

axiomTM



The 30 Year Horizon

<i>Manuel Bronstein</i>	<i>William Burge</i>	<i>Timothy Daly</i>
<i>James Davenport</i>	<i>Michael Dewar</i>	<i>Martin Dunstan</i>
<i>Albrecht Fortenbacher</i>	<i>Patrizia Gianni</i>	<i>Johannes Grabmeier</i>
<i>Jocelyn Guidry</i>	<i>Richard Jenks</i>	<i>Larry Lambe</i>
<i>Michael Monagan</i>	<i>Scott Morrison</i>	<i>William Sit</i>
<i>Jonathan Steinbach</i>	<i>Robert Sutor</i>	<i>Barry Trager</i>
<i>Stephen Watt</i>	<i>Jim Wen</i>	<i>Clifton Williamson</i>

Volume 5: Axiom Interpreter

Contents

1	Credits	1
1.0.1	defvar \$credits	1
2	The Interpreter	5
3	The Fundamental Data Structures	7
3.1	The global variables	7
3.1.1	defvar \$current-directory	7
3.1.2	defvar \$current-directory	7
3.1.3	defvar \$defaultMsgDatabaseName	8
3.1.4	defvar \$defaultMsgDatabaseName	8
3.1.5	defvar \$directory-list	8
3.1.6	defvar \$directory-list	8
3.1.7	defvar \$InitialModemapFrame	9
3.1.8	defvar \$InitialModemapFrame	9
3.1.9	defvar \$library-directory-list	9
3.1.10	defvar \$library-directory-list	9
3.1.11	defvar \$msgDatabaseName	9
3.1.12	defvar \$msgDatabaseName	10
3.1.13	defvar \$openServerIfTrue	10
3.1.14	defvar \$openServerIfTrue	10
3.1.15	defvar \$relative-directory-list	10
3.1.16	defvar \$relative-directory-list	11
3.1.17	defvar \$relative-library-directory-list	11
3.1.18	defvar \$relative-library-directory-list	11
3.1.19	defvar \$spadroot	11
3.1.20	defvar \$spadroot	12
3.1.21	defvar \$SpadServer	12
3.1.22	defvar \$SpadServer	12
3.1.23	defvar \$SpadServerName	12
3.1.24	defvar \$SpadServerName	13

4	Starting Axiom	15
4.1	Variables Used	15
4.2	Data Structures	15
4.3	Functions	15
4.3.1	Set the restart hook	15
4.3.2	restart function (The restart function)	16
4.3.3	defun Non-interactive restarts	18
4.3.4	defun The startup banner messages	19
4.3.5	defun Make a vector of filler characters	20
4.3.6	Starts the interpreter but do not read in profiles	20
4.3.7	defvar \$quitTag	20
4.3.8	defun runspad	20
4.3.9	defun Reset the stack limits	21
5	Handling Terminal Input	23
5.1	Streams	23
5.1.1	defvar \$curinstream	23
5.1.2	defvar \$curoutstream	23
5.1.3	defvar \$errorinstream	23
5.1.4	defvar \$erroroutstream	24
5.1.5	defvar \$*eof*	24
5.1.6	defvar \$*whitespace*	24
5.1.7	defvar \$InteractiveMode	24
5.1.8	defvar \$boot	25
5.1.9	Top-level read-parse-eval-print loop	25
5.1.10	defun ncIntLoop	25
5.1.11	defvar \$intTopLevel	26
5.1.12	defvar \$intRestart	26
5.1.13	defun intloop	26
5.1.14	defvar \$ncMsgList	27
5.1.15	defun SpadInterpretStream	27
5.1.16	defvar \$promptMsg	28
5.1.17	defvar \$newcompErrorCount	28
5.1.18	defvar \$nopus	28
5.2	The Read-Eval-Print Loop	29
5.2.1	defun intloopReadConsole	29
5.3	Helper Functions	31
5.3.1	Get the value of an environment variable	31
5.3.2	defvar \$intCoerceFailure	31
5.3.3	defvar \$intSpadReader	32
5.3.4	defun InterpExecuteSpadSystemCommand	32
5.3.5	defun ExecuteInterpSystemCommand	32
5.3.6	defun Handle Synonyms	33
5.3.7	defun Synonym File Reader	33
5.3.8	defun init-memory-config	34
5.3.9	Set spadroot to be the AXIOM shell variable	35

5.3.10	Does the string start with this prefix?	35
5.3.11	defun Interpret a line of lisp code	36
5.3.12	Get the current directory	36
5.3.13	Prepend the absolute path to a filename	36
5.3.14	Make the initial modemap frame	36
5.3.15	defun ncloopEscaped	37
5.3.16	defun intloopProcessString	37
5.3.17	defun ncloopParse	37
5.3.18	defun next	38
5.3.19	defun next1	38
5.3.20	defun incString	39
5.3.21	Call the garbage collector	39
5.3.22	defun reroot	40
5.3.23	defun setCurrentLine	41
5.3.24	Show the Axiom prompt	42
5.3.25	defvar \$frameAlist	42
5.3.26	defvar \$frameNumber	43
5.3.27	defvar \$currentFrameNum	43
5.3.28	defvar \$EndServerSession	43
5.3.29	defvar \$NeedToSignalSessionManager	43
5.3.30	defvar \$sockBufferLength	44
5.3.31	READ-LINE in an Axiom server system	44
5.3.32	defun protectedEVAL	46
5.3.33	defvar \$QuietCommand	47
5.3.34	defun executeQuietCommand	47
5.3.35	defun parseAndInterpret	48
5.3.36	defun ncParseAndInterpretString	48
5.3.37	defun parseFromString	48
5.3.38	defvar \$interpOnly	49
5.3.39	defvar \$minivectorNames	49
5.3.40	defvar \$domPvar	49
5.3.41	defun processInteractive	49
5.3.42	defvar \$ProcessInteractiveValue	52
5.3.43	defvar \$HTCompanionWindowID	52
5.3.44	defun processInteractive1	52
5.3.45	defun interpretTopLevel	53
5.3.46	defvar \$genValue	53
5.3.47	defun Type analyzes and evaluates expression x, returns object	54
5.3.48	defun Dispatcher for the type analysis routines	54
5.3.49	defun interpret2	55
5.3.50	defun Result Output Printing	56
5.3.51	defun printStatisticsSummary	58
5.3.52	defun printStorage	58
5.3.53	defun printTypeAndTime	58
5.3.54	defun printTypeAndTimeNormal	59
5.3.55	defun printTypeAndTimeSaturn	60

5.3.56	defun printAsTeX	61
5.3.57	defun sameUnionBranch	61
5.3.58	defun msgText	62
5.3.59	defun Right-justify the Type output	62
5.3.60	defun Destructively fix quotes in strings	63
5.3.61	Include a file into the stream	63
5.3.62	defun intloopInclude0	63
5.3.63	defun intloopProcess	64
5.3.64	defun intloopSpadProcess	65
5.3.65	defun intloopSpadProcess,interp	66
5.3.66	defun phParse	66
5.3.67	defun intSayKeyedMsg	67
5.3.68	defun packageTran	67
5.3.69	defun phIntReportMsgs	67
5.3.70	defun phInterpret	68
5.3.71	defun intInterpretPform	68
5.3.72	defun zeroOneTran	69
5.3.73	defun ncConversationPhase	69
5.3.74	defun ncConversationPhase,wrapup	69
5.3.75	defun ncError	70
5.3.76	defun intloopEchoParse	70
5.3.77	defun ncloopPrintLines	71
5.3.78	defun mkLineList	71
5.3.79	defun nonBlank	72
5.3.80	defun ncloopDQlines	72
5.3.81	defun poGlobalLinePosn	73
5.3.82	defun streamChop	73
5.3.83	defun ncloopInclude0	74
5.3.84	defun incStream	74
5.3.85	defun incRenumber	75
5.3.86	defun incZip	75
5.3.87	defun incZip1	75
5.3.88	defun incIgen	76
5.3.89	defun incIgen1	76
5.3.90	defun incRenumberLine	76
5.3.91	defun incRenumberItem	77
5.3.92	defun incHandleMessage	77
5.3.93	defun incLude	78
5.3.94	defmacro Rest	78
5.3.95	defvar \$Top	78
5.3.96	defvar \$IfSkipToEnd	78
5.3.97	defvar \$IfKeepPart	79
5.3.98	defvar \$IfSkipPart	79
5.3.99	defvar \$ElseifSkipToEnd	79
5.3.100	defvar \$ElseifKeepPart	79
5.3.101	defvar \$ElseifSkipPart	79

5.3.102 defvar \$ElseSkipToEnd	80
5.3.103 defvar \$ElseKeepPart	80
5.3.104 defvar \$Top?	80
5.3.105 defvar \$If?	80
5.3.106 defvar \$Elseif?	81
5.3.107 defvar \$Else?	81
5.3.108 defvar \$SkipEnd?	81
5.3.109 defvar \$KeepPart?	82
5.3.110 defvar \$SkipPart?	82
5.3.111 defvar \$Skipping?	82
5.3.112 defun incLude1	82
5.3.113 defun xlPrematureEOF	87
5.3.114 defun xlMsg	87
5.3.115 defun xlOK	87
5.3.116 defun xlOK1	88
5.3.117 defun incAppend	88
5.3.118 defun incAppend1	88
5.3.119 defun incLine	89
5.3.120 defun incLine1	89
5.3.121 defun inclmsgPrematureEOF	89
5.3.122 defun theorigin	89
5.3.123 defun porigin	90
5.3.124 defun ifCond	90
5.3.125 defun xlSkip	90
5.3.126 defun xlSay	91
5.3.127 defun inclmsgSay	91
5.3.128 defun theid	91
5.3.129 defun xlNoSuchFile	92
5.3.130 defun inclmsgNoSuchFile	92
5.3.131 defun thefname	92
5.3.132 defun pfname	92
5.3.133 defun xlCannotRead	93
5.3.134 defun inclmsgCannotRead	93
5.3.135 defun xlFileCycle	93
5.3.136 defun inclmsgFileCycle	93
5.3.137 defun xlConActive	94
5.3.138 defun inclmsgConActive	95
5.3.139 defun xlConStill	95
5.3.140 defun inclmsgConStill	95
5.3.141 defun xlConsole	95
5.3.142 defun inclmsgConsole	96
5.3.143 defun xlSkippingFin	96
5.3.144 defun inclmsgFinSkipped	96
5.3.145 defun xlPrematureFin	96
5.3.146 defun inclmsgPrematureFin	97
5.3.147 defun assertCond	97

5.3.148 defun xIfSyntax	97
5.3.149 defun inclmsgIfSyntax	98
5.3.150 defun xIfBug	98
5.3.151 defun inclmsgIfBug	99
5.3.152 defun xlCmdBug	99
5.3.153 defun inclmsgCmdBug	99
5.3.154 defvar \$incCommands	99
5.3.155 defvar \$pfMacros	100
5.3.156 defun incClassify	100
5.3.157 defun incCommand?	101
5.3.158 defun incPrefix?	102
5.3.159 defun incCommandTail	102
5.3.160 defun incDrop	103
5.3.161 defun inclFname	103
5.3.162 defun incFileInput	103
5.3.163 defun incConsoleInput	103
5.3.164 defun incNConsoles	104
5.3.165 defun incActive?	104
5.3.166 defun incRgen	104
5.3.167 defun Delay	105
5.3.168 defvar \$StreamNil	105
5.3.169 defvar \$StreamNil	105
5.3.170 defun incRgen1	105
6 The Token Scanner	107
6.0.171 defvar \$space	107
6.0.172 defvar \$escape	107
6.0.173 defvar \$stringchar	107
6.0.174 defvar \$pluscomment	108
6.0.175 defvar \$minuscomment	108
6.0.176 defvar \$radixchar	108
6.0.177 defvar \$dot	108
6.0.178 defvar \$exponent1	109
6.0.179 defvar \$exponent2	109
6.0.180 defvar \$closeparen	109
6.0.181 defvar \$closeangle	109
6.0.182 defvar \$question	110
6.0.183 defvar \$scanKeyWords	110
6.0.184 defvar \$infgeneric	112
6.0.185 defun lineoftoks	113
6.0.186 defun nextline	115
6.0.187 defun scanIgnoreLine	115
6.0.188 defun constoken	116
6.0.189 defun scanToken	116
6.0.190 defun lfid	117
6.0.191 defun startsComment?	118

6.0.192 defun scanComment	118
6.0.193 defun lfcomment	119
6.0.194 defun startsNegComment?	119
6.0.195 defun scanNegComment	119
6.0.196 defun lfnegcomment	120
6.0.197 defun punctuation?	120
6.0.198 defun scanPunct	120
6.0.199 defun subMatch	121
6.0.200 defun substringMatch	121
6.0.201 defun scanKeyTr	122
6.0.202 defun keyword	123
6.0.203 defun keyword?	123
6.0.204 defun scanPossFloat	123
6.0.205 defun digit?	124
6.0.206 defun lfkey	124
6.0.207 defun spleI	124
6.0.208 defun spleI1	125
6.0.209 defun scanEsc	125
6.0.210 defvar \$scanCloser	127
6.0.211 defun scanCloser?	128
6.0.212 defun scanWord	128
6.0.213 defun scanExponent	128
6.0.214 defun lffloat	130
6.0.215 defmacro idChar?	130
6.0.216 defun scanW	130
6.0.217 defun posend	131
6.0.218 defun scanSpace	131
6.0.219 defun lfspace	132
6.0.220 defun scanString	132
6.0.221 defun lfstring	133
6.0.222 defun scanS	133
6.0.223 defun scanTransform	134
6.0.224 defun scanNumber	134
6.0.225 defun rdigit?	135
6.0.226 defun lfinteger	136
6.0.227 defun lfrinteger	136
6.0.228 defun scanCheckRadix	136
6.0.229 defun scanEscape	137
6.0.230 defun scanError	137
6.0.231 defun lferror	138
6.0.232 defvar \$scanKeyTable	138
6.0.233 defun scanKeyTableCons	138
6.0.234 defvar \$scanDict	139
6.0.235 defun scanDictCons	139
6.0.236 defun scanInsert	140
6.0.237 defvar \$scanPun	141

6.0.238 defun scanPunCons	141
7 Input Stream Parser	143
7.0.239 defun Input Stream Parser	143
7.0.240 defun npItem	144
7.0.241 defun npItem1	144
7.0.242 defun npFirstTok	145
7.0.243 defun Push one item onto \$stack	145
7.0.244 defun Pop one item off \$stack	146
7.0.245 defun Pop the second item off \$stack	146
7.0.246 defun Pop the third item off \$stack	146
7.0.247 defun npQualDef	147
7.0.248 defun Advance over a keyword	147
7.0.249 defun Advance the input stream	147
7.0.250 defun npComma	148
7.0.251 defun npTuple	148
7.0.252 defun npCommaBackSet	148
7.0.253 defun npQualifiedDefinition	149
7.0.254 defun npQualified	149
7.0.255 defun npDefinitionOrStatement	149
7.0.256 defun npBackTrack	150
7.0.257 defun npGives	150
7.0.258 defun npLambda	150
7.0.259 defun npType	151
7.0.260 defun npMatch	152
7.0.261 defun npSuch	152
7.0.262 defun npWith	152
7.0.263 defun npCompMissing	153
7.0.264 defun npMissing	153
7.0.265 defun npRestore	154
7.0.266 defun Peek for keyword s, no advance of token stream	154
7.0.267 defun npCategoryL	154
7.0.268 defun npCategory	155
7.0.269 defun npSCategory	155
7.0.270 defun npSignature	156
7.0.271 defun npSigItemList	156
7.0.272 defun npListing	157
7.0.273 defun Always produces a list, fn is applied to it	157
7.0.274 defun npSigItem	158
7.0.275 defun npTypeVariable	158
7.0.276 defun npSignatureDefinee	158
7.0.277 defun npTypeVariablelist	159
7.0.278 defun npSigDecl	159
7.0.279 defun npPrimary	159
7.0.280 defun npPrimary2	160
7.0.281 defun npADD	160

7.0.282 defun npAdd	161
7.0.283 defun npAtom2	161
7.0.284 defun npInfixOperator	162
7.0.285 defun npInfixOp	163
7.0.286 defun npPrefixColon	163
7.0.287 defun npApplication	164
7.0.288 defun npDotted	164
7.0.289 defun npAnyNo	164
7.0.290 defun npSelector	165
7.0.291 defun npApplication2	165
7.0.292 defun npPrimary1	166
7.0.293 defun npMacro	166
7.0.294 defun npMdef	166
7.0.295 defun npMDEF	167
7.0.296 defun npMDEFinition	167
7.0.297 defun npFix	168
7.0.298 defun npLet	168
7.0.299 defun npLetQualified	168
7.0.300 defun npDefinition	169
7.0.301 defun npDefinitionItem	169
7.0.302 defun npTyping	170
7.0.303 defun npDefaultItemlist	170
7.0.304 defun npSDefaultItem	171
7.0.305 defun npDefaultItem	171
7.0.306 defun npDefaultDecl	172
7.0.307 defun npStatement	172
7.0.308 defun npExport	173
7.0.309 defun npLocalItemlist	173
7.0.310 defun npSLocalItem	174
7.0.311 defun npLocalItem	174
7.0.312 defun npLocalDecl	174
7.0.313 defun npLocal	175
7.0.314 defun npFree	175
7.0.315 defun npInline	176
7.0.316 defun npIterate	176
7.0.317 defun npBreak	176
7.0.318 defun npLoop	177
7.0.319 defun npIterators	177
7.0.320 defun npIterator	178
7.0.321 defun npSuchThat	178
7.0.322 defun Apply argument 0 or more times	179
7.0.323 defun npWhile	179
7.0.324 defun npForIn	179
7.0.325 defun npReturn	180
7.0.326 defun npVoid	181
7.0.327 defun npExpress	181

7.0.328 defun npExpress1	181
7.0.329 defun npConditionalStatement	182
7.0.330 defun npImport	182
7.0.331 defun npQualTypelist	182
7.0.332 defun npSQualTypelist	183
7.0.333 defun npQualType	183
7.0.334 defun npAndOr	183
7.0.335 defun npEncAp	184
7.0.336 defun npEncl	184
7.0.337 defun npAtom1	185
7.0.338 defun npPDefinition	185
7.0.339 defun npDollar	185
7.0.340 defun npConstTok	186
7.0.341 defun npBDefinition	187
7.0.342 defun npBracketed	187
7.0.343 defun npParened	187
7.0.344 defun npBracked	188
7.0.345 defun npBraced	188
7.0.346 defun npAngleBared	188
7.0.347 defun npDefn	189
7.0.348 defun npDef	189
7.0.349 defun npBPileDefinition	190
7.0.350 defun npPileBracketed	190
7.0.351 defun npPileDefinitionlist	191
7.0.352 defun npListAndRecover	191
7.0.353 defun npRecoverTrap	192
7.0.354 defun npMoveTo	193
7.0.355 defun syIgnoredFromTo	193
7.0.356 defun syGeneralErrorHere	194
7.0.357 defun sySpecificErrorHere	194
7.0.358 defun sySpecificErrorAtToken	194
7.0.359 defun npDefinitionlist	195
7.0.360 defun npSemiListing	195
7.0.361 defun npSemiBackSet	195
7.0.362 defun npRule	195
7.0.363 defun npSingleRule	196
7.0.364 defun npDefTail	196
7.0.365 defun npDefaultValue	196
7.0.366 defun npWConditional	197
7.0.367 defun npConditional	197
7.0.368 defun npElse	198
7.0.369 defun npBacksetElse	199
7.0.370 defun npLogical	199
7.0.371 defun npDisjand	199
7.0.372 defun npDiscrim	199
7.0.373 defun npQuiver	200

7.0.374 defun npRelation	200
7.0.375 defun npSynthetic	200
7.0.376 defun npBy	201
7.0.377 defun	201
7.0.378 defun npSegment	202
7.0.379 defun npArith	202
7.0.380 defun npSum	203
7.0.381 defun npTerm	203
7.0.382 defun npRemainder	203
7.0.383 defun npProduct	204
7.0.384 defun npPower	204
7.0.385 defun npAmpersandFrom	204
7.0.386 defun npFromdom	204
7.0.387 defun npFromdom1	205
7.0.388 defun npAmpersand	206
7.0.389 defun npName	206
7.0.390 defvar \$npPParg	206
7.0.391 defun npId	206
7.0.392 defun npSymbolVariable	207
7.0.393 defun npRightAssoc	208
7.0.394 defun $p \circ p \circ p \circ p = (((p \circ p) \circ p) \circ p)$	208
7.0.395 defun npInfGeneric	209
7.0.396 defun npDDInfKey	210
7.0.397 defun npInfKey	210
7.0.398 defun npPushId	211
7.0.399 defvar \$npPParg	211
7.0.400 defun npPP	211
7.0.401 defun npPPff	212
7.0.402 defun npPPg	212
7.0.403 defun npPPf	213
7.0.404 defun npEnclosed	213
7.0.405 defun npState	214
7.0.406 defun npTrap	214
7.0.407 defun npTrapForm	214
7.0.408 defun npVariable	215
7.0.409 defun npVariablelist	215
7.0.410 defun npVariableName	215
7.0.411 defun npDecl	216
7.0.412 defun npParenthesized	216
7.0.413 defun npParenthesize	217
7.0.414 defun npMissingMate	217
7.0.415 defun npExit	217
7.0.416 defun npPileExit	218
7.0.417 defun npAssign	218
7.0.418 defun npAssignment	219
7.0.419 defun npAssignVariable	219

7.0.420	defun npColon	219
7.0.421	defun npTagged	220
7.0.422	defun npTypedForm1	220
7.0.423	defun npTypified	220
7.0.424	defun npTypeStyle	221
7.0.425	defun npPretend	221
7.0.426	defun npColonQuery	221
7.0.427	defun npCoerceTo	222
7.0.428	defun npTypedForm	222
7.0.429	defun npRestrict	222
7.0.430	defun npListofFun	223
7.1	Macro handling	223
7.1.1	defun phMacro	223
7.1.2	defun macroExpanded	224
7.1.3	defun macExpand	224
7.1.4	defun macApplication	225
7.1.5	defun mac0MLambdaApply	225
7.1.6	defun mac0ExpandBody	226
7.1.7	defun mac0InfiniteExpansion	227
7.1.8	defun mac0InfiniteExpansion,name	228
7.1.9	defun mac0GetName	228
7.1.10	defun macId	229
7.1.11	defun mac0Get	230
7.1.12	defun macWhere	230
7.1.13	defun macWhere,mac	230
7.1.14	defun macLambda	230
7.1.15	defun macLambda,mac	231
7.1.16	defun Add appropriate definition the a Macro pform	231
7.1.17	defun Add a macro to the global pfMacros list	232
7.1.18	defun macSubstituteOuter	232
7.1.19	defun mac0SubstituteOuter	233
7.1.20	defun macLambdaParameterHandling	233
7.1.21	defun macSubstituteId	234
8	Pftrees	235
8.1	Abstract Syntax Trees Overview	235
8.2	Structure handlers	237
8.2.1	defun pfGlobalLinePosn	237
8.2.2	defun pfCharPosn	237
8.2.3	defun pfLinePosn	237
8.2.4	defun pfFileName	238
8.2.5	defun pfCopyWithPos	238
8.2.6	defun pfMapParts	238
8.2.7	defun pf0ApplicationArgs	239
8.2.8	defun pf0FlattenSyntacticTuple	239
8.2.9	defun pfSourcePosition	240

8.2.10	defun Convert a Sequence node to a list	240
8.2.11	defun pfSpread	241
8.2.12	defun Deconstruct nodes to lists	241
8.2.13	defun pfCheckMacroOut	242
8.2.14	defun pfCheckArg	243
8.2.15	defun pfCheckId	243
8.2.16	defun pfFlattenApp	243
8.2.17	defun pfCollect1?	244
8.2.18	defun pfCollectVariable1	244
8.2.19	defun pfPushMacroBody	245
8.2.20	defun pfSourceStok	245
8.2.21	defun pfTransformArg	246
8.2.22	defun pfTaggedToTyped1	246
8.2.23	defun pfSuch	246
8.3	Special Nodes	247
8.3.1	defun Create a Listof node	247
8.3.2	defun pfNothing	247
8.3.3	defun Is this a Nothing node?	247
8.4	Leaves	248
8.4.1	defun Create a Document node	248
8.4.2	defun Construct an Id node	248
8.4.3	defun Is this an Id node?	248
8.4.4	defun Construct an Id leaf node	248
8.4.5	defun Return the Id part	249
8.4.6	defun Construct a Leaf node	249
8.4.7	defun Is this a leaf node?	249
8.4.8	defun Return the token position of a leaf node	250
8.4.9	defun Return the Leaf Token	250
8.4.10	defun Is this a Literal node?	250
8.4.11	defun Create a LiteralClass node	250
8.4.12	defun Return the LiteralString	251
8.4.13	defun Return the parts of a tree node	251
8.4.14	defun Return the argument unchanged	251
8.4.15	defun pfPushBody	251
8.4.16	defun An S-expression which people can read.	252
8.4.17	defun Create a human readable S-expression	252
8.4.18	defun Construct a Symbol or Expression node	253
8.4.19	defun Construct a Symbol leaf node	253
8.4.20	defun Is this a Symbol node?	254
8.4.21	defun Return the Symbol part	254
8.5	Trees	254
8.5.1	defun Construct a tree node	254
8.5.2	defun Construct an Add node	254
8.5.3	defun Construct an And node	255
8.5.4	defun pfAttribute	255
8.5.5	defun Return an Application node	255

8.5.6	defun Return the Arg part of an Application node	256
8.5.7	defun Return the Op part of an Application node	256
8.5.8	defun Is this an And node?	256
8.5.9	defun Return the Left part of an And node	256
8.5.10	defun Return the Right part of an And node	257
8.5.11	defun Flatten a list of lists	257
8.5.12	defun Is this an Application node?	257
8.5.13	defun Create an Assign node	257
8.5.14	defun Is this an Assign node?	258
8.5.15	defun Return the parts of an LhsItem of an Assign node	258
8.5.16	defun Return the LhsItem of an Assign node	258
8.5.17	defun Return the RHS of an Assign node	258
8.5.18	defun Construct an application node for a brace	259
8.5.19	defun Construct an Application node for brace-bars	259
8.5.20	defun Construct an Application node for a bracket	259
8.5.21	defun Construct an Application node for bracket-bars	259
8.5.22	defun Create a Break node	260
8.5.23	defun Is this a Break node?	260
8.5.24	defun Return the From part of a Break node	260
8.5.25	defun Construct a Coerceto node	261
8.5.26	defun Is this a CoerceTo node?	261
8.5.27	defun Return the Expression part of a CoerceTo node	261
8.5.28	defun Return the Type part of a CoerceTo node	261
8.5.29	defun Return the Body of a Collect node	262
8.5.30	defun Return the Iterators of a Collect node	262
8.5.31	defun Create a Collect node	262
8.5.32	defun Is this a Collect node?	262
8.5.33	defun pfDefinition	263
8.5.34	defun Return the Lhs of a Definition node	263
8.5.35	defun Return the Rhs of a Definition node	263
8.5.36	defun Is this a Definition node?	263
8.5.37	defun Return the parts of a Definition node	264
8.5.38	defun Create a Do node	264
8.5.39	defun Is this a Do node?	264
8.5.40	defun Return the Body of a Do node	264
8.5.41	defun Construct a Sequence node	265
8.5.42	defun Construct an Exit node	265
8.5.43	defun Is this an Exit node?	265
8.5.44	defun Return the Cond part of an Exit	265
8.5.45	defun Return the Expression part of an Exit	266
8.5.46	defun Create an Export node	266
8.5.47	defun Construct an Expression leaf node	266
8.5.48	defun pfFirst	266
8.5.49	defun Create an Application Fix node	267
8.5.50	defun Create a Free node	267
8.5.51	defun Is this a Free node?	267

8.5.52	defun Return the parts of the Items of a Free node	268
8.5.53	defun Return the Items of a Free node	268
8.5.54	defun Construct a Forin node	268
8.5.55	defun Is this a ForIn node?	268
8.5.56	defun Return all the parts of the LHS of a ForIn node	269
8.5.57	defun Return the LHS part of a ForIn node	269
8.5.58	defun Return the Whole part of a ForIn node	269
8.5.59	defun pfFromDom	269
8.5.60	defun Construct a Fromdom node	270
8.5.61	defun Is this a Fromdom mode?	270
8.5.62	defun Return the What part of a Fromdom node	270
8.5.63	defun Return the Domain part of a Fromdom node	271
8.5.64	defun Construct a Hide node	271
8.5.65	defun pfIf	271
8.5.66	defun Is this an If node?	271
8.5.67	defun Return the Cond part of an If	272
8.5.68	defun Return the Then part of an If	272
8.5.69	defun pfIfThenOnly	272
8.5.70	defun Return the Else part of an If	272
8.5.71	defun Construct an Import node	273
8.5.72	defun Construct an Iterate node	273
8.5.73	defun Is this an Iterate node?	273
8.5.74	defun Handle an infix application	273
8.5.75	defun Create an Inline node	274
8.5.76	defun pfLam	274
8.5.77	defun pfLambda	275
8.5.78	defun Return the Body part of a Lambda node	275
8.5.79	defun Return the Rets part of a Lambda node	275
8.5.80	defun Is this a Lambda node?	275
8.5.81	defun Return the Args part of a Lambda node	276
8.5.82	defun Return the Args of a Lambda Node	276
8.5.83	defun Construct a Local node	276
8.5.84	defun Is this a Local node?	276
8.5.85	defun Return the parts of Items of a Local node	277
8.5.86	defun Return the Items of a Local node	277
8.5.87	defun Construct a Loop node	277
8.5.88	defun pfLoop1	277
8.5.89	defun Is this a Loop node?	278
8.5.90	defun Return the Iterators of a Loop node	278
8.5.91	defun pf0LoopIterators	278
8.5.92	defun pfLp	278
8.5.93	defun Create a Macro node	279
8.5.94	defun Is this a Macro node?	279
8.5.95	defun Return the Lhs of a Macro node	279
8.5.96	defun Return the Rhs of a Macro node	279
8.5.97	defun Construct an MLambda node	280

8.5.98 defun Is this an MLambda node?	280
8.5.99 defun Return the Args of an MLambda	280
8.5.100 defun Return the parts of an MLambda argument	280
8.5.101 defun pfMLambdaBody	281
8.5.102 defun Is this a Not node?	281
8.5.103 defun Return the Arg part of a Not node	281
8.5.104 defun Construct a NoValue node	281
8.5.105 defun Is this a Novalue node?	282
8.5.106 defun Return the Expr part of a Novalue node	282
8.5.107 defun Construct an Or node	282
8.5.108 defun Is this an Or node?	282
8.5.109 defun Return the Left part of an Or node	283
8.5.110 defun Return the Right part of an Or node	283
8.5.111 defun Return the part of a parenthesised expression	283
8.5.112 defun pfPretend	283
8.5.113 defun Is this a Pretend node?	284
8.5.114 defun Return the Expression part of a Pretend node	284
8.5.115 defun Return the Type part of a Pretend node	284
8.5.116 defun Construct a QualType node	284
8.5.117 defun Construct a Restrict node	285
8.5.118 defun Is this a Restrict node?	285
8.5.119 defun Return the Expr part of a Restrict node	285
8.5.120 defun Return the Type part of a Restrict node	285
8.5.121 defun Construct a RetractTo node	286
8.5.122 defun Construct a Return node	286
8.5.123 defun Is this a Return node?	286
8.5.124 defun Return the Expr part of a Return node	286
8.5.125 defun pfReturnNoName	287
8.5.126 defun Construct a ReturnTyped node	287
8.5.127 defun Construct a Rule node	287
8.5.128 defun Return the Lhs of a Rule node	288
8.5.129 defun Return the Rhs of a Rule node	288
8.5.130 defun Is this a Rule node?	288
8.5.131 defun pfSecond	288
8.5.132 defun Construct a Sequence node	289
8.5.133 defun Return the Args of a Sequence node	289
8.5.134 defun Is this a Sequence node?	289
8.5.135 defun Return the parts of the Args of a Sequence node	289
8.5.136 defun Create a Suchthat node	290
8.5.137 defun Is this a SuchThat node?	290
8.5.138 defun Return the Cond part of a SuchThat node	290
8.5.139 defun Create a Tagged node	290
8.5.140 defun Is this a Tagged node?	291
8.5.141 defun Return the Expression portion of a Tagged node	291
8.5.142 defun Return the Tag of a Tagged node	291
8.5.143 defun pfTaggedToTyped	291

8.5.144 defun pfTweakIf	292
8.5.145 defun Construct a Typed node	292
8.5.146 defun Is this a Typed node?	293
8.5.147 defun Return the Type of a Typed node	293
8.5.148 defun Return the Id of a Typed node	293
8.5.149 defun Construct a Typing node	293
8.5.150 defun Return a Tuple node	294
8.5.151 defun Return a Tuple from a List	294
8.5.152 defun Is this a Tuple node?	294
8.5.153 defun Return the Parts of a Tuple node	295
8.5.154 defun Return the parts of a Tuple	295
8.5.155 defun Return a list from a Sequence node	295
8.5.156 defun The comment is attached to all signatutres	295
8.5.157 defun Construct a WDeclare node	296
8.5.158 defun Construct a Where node	296
8.5.159 defun Is this a Where node?	296
8.5.160 defun Return the parts of the Context of a Where node	297
8.5.161 defun Return the Context of a Where node	297
8.5.162 defun Return the Expr part of a Where node	297
8.5.163 defun Construct a While node	297
8.5.164 defun Is this a While node?	298
8.5.165 defun Return the Cond part of a While node	298
8.5.166 defun Construct a With node	298
8.5.167 defun Create a Wrong node	298
8.5.168 defun Is this a Wrong node?	299
9 Pftree to s-expression translation	301
9.0.169 defun Pftree to s-expression translation	301
9.0.170 defun Pftree to s-expression translation inner function	302
9.0.171 defun Convert a Literal to an S-expression	306
9.0.172 defun Convert a float to an S-expression	307
9.0.173 defun Change an Application node to an S-expression	307
9.0.174 defun Convert a SuchThat node to an S-expression	309
9.0.175 defun pfOp2Sex	310
9.0.176 defun pmDontQuote?	311
9.0.177 defun hasOptArgs?	311
9.0.178 defun Convert a Sequence node to an S-expression	312
9.0.179 defun pfSequence2Sex0	312
9.0.180 defun Convert a loop node to an S-expression	313
9.0.181 defun Change a Collect node to an S-expression	316
9.0.182 defun Convert a Definition node to an S-expression	317
9.0.183 defun Convert a Lambda node to an S-expression	318
9.0.184 defun pfCollectArgTran	319
9.0.185 defun Convert a Lambda node to an S-expression	319
9.0.186 defun Convert a Rule node to an S-expression	320
9.0.187 defun Convert the Lhs of a Rule to an S-expression	320

9.0.188 defun	Convert the Rhs of a Rule to an S-expression	321
9.0.189 defun	Convert a Rule predicate to an S-expression	321
9.0.190 defun	patternVarsOf	323
9.0.191 defun	patternVarsOf1	323
9.0.192 defun	pvarPredTran	324
9.0.193 defun	Convert the Lhs of a Rule node to an S-expression	324
9.0.194 defvar	\$dotdot	325
9.0.195 defun	Translate ops into internal symbols	325
10	Keyed Message Handling	327
10.0.196 defvar	\$cacheMessages	328
10.0.197 defvar	\$msgAlist	328
10.0.198 defvar	\$msgDatabaseName	328
10.0.199 defvar	\$testingErrorPrefix	329
10.0.200 defvar	\$texFormatting	329
10.0.201 defvar	\$*msghash*	329
10.0.202 defvar	\$msgdbPrims	329
10.0.203 defvar	\$msgdbPunct	329
10.0.204 defvar	\$msgdbNoBlanksBeforeGroup	330
10.0.205 defvar	\$msgdbNoBlanksAfterGroup	330
10.0.206 defun	Fetch a message from the message database	330
10.0.207 defun	Cache messages read from message database	331
10.0.208 defun	getKeyedMsg	331
10.0.209 defun	Say a message using a keyed lookup	331
10.0.210 defun	Handle msg formatting and print to file	332
10.0.211 defun	Break a message into words	332
10.0.212 defun	Write a msg into spadmsg.listing file	333
10.0.213 defun	sayMSG	333
11	Stream Utilities	335
11.0.214 defun	npNull	335
11.0.215 defun	StreamNull	335
12	Code Piles	337
12.0.216 defun	insertpile	337
12.0.217 defun	pilePlusComment	338
12.0.218 defun	pilePlusComments	338
12.0.219 defun	pileTree	339
12.0.220 defun	pileColumn	339
12.0.221 defun	pileForests	339
12.0.222 defun	pileForest	340
12.0.223 defun	pileForest1	340
12.0.224 defun	eqpileTree	341
12.0.225 defun	pileCtree	342
12.0.226 defun	pileCforest	342
12.0.227 defun	enPile	342

12.0.22	defun firstTokPosn	343
12.0.22	defun lastTokPosn	343
12.0.23	defun separatePiles	343
13	Dequeue Functions	345
13.0.23	defun dqUnit	345
13.0.23	defun dqConcat	345
13.0.23	defun dqAppend	346
13.0.24	defun dqToList	346
14	Message Handling	347
14.1	The Line Object	347
14.1.1	defun Line object creation	347
14.1.2	defun Line element 0; Extra blanks	347
14.1.3	defun Line element 1; String	347
14.1.4	defun Line element 2; Global number	348
14.1.5	defun Line element 2; Set Global number	348
14.1.6	defun Line element 3; Local number	348
14.1.7	defun Line element 4; Place of origin	348
14.1.8	defun Line element 4: Is it a filename?	349
14.1.9	defun Line element 4: Is it a filename?	349
14.1.10	defun Line element 4; Get filename	349
14.2	Messages	349
14.2.1	defun msgCreate	349
14.2.2	defun getMsgPosTagOb	350
14.2.3	defun getMsgKey	350
14.2.4	defun getMsgArgL	351
14.2.5	defun getMsgPrefix	351
14.2.6	defun setMsgPrefix	351
14.2.7	defun getMsgText	351
14.2.8	defun setMsgText	351
14.2.9	defun getMsgPrefix?	352
14.2.10	defun getMsgTag	352
14.2.11	defun getMsgTag?	352
14.2.12	defun line?	353
14.2.13	defun leader?	353
14.2.14	defun toScreen?	353
14.2.15	defun ncSoftError	353
14.2.16	defun ncHardError	354
14.2.17	defun desiredMsg	354
14.2.18	defun processKeyedError	355
14.2.19	defun msgOutputter	355
14.2.20	defun listOutputter	356
14.2.21	defun getStFromMsg	356
14.2.22	defvar \$preLength	357
14.2.23	defun getPreStL	357

14.2.24 defun getPosStL	358
14.2.25 defun ppos	359
14.2.26 defun remFile	359
14.2.27 defun showMsgPos?	359
14.2.28 defvar \$imPrGuys	360
14.2.29 defun msgImPr?	360
14.2.30 defun getMsgCatAttr	360
14.2.31 defun getMsgPos	361
14.2.32 defun getMsgFTTag?	361
14.2.33 defun decideHowMuch	361
14.2.34 defun poNopos?	362
14.2.35 defun poPosImmediate?	362
14.2.36 defun poFileName	362
14.2.37 defun poGetLineObject	363
14.2.38 defun poLinePosn	363
14.2.39 defun listDecideHowMuch	363
14.2.40 defun remLine	364
14.2.41 defun getMsgKey?	364
14.2.42 defun getMsgLitSym	364
14.2.43 defun tabbing	364
14.2.44 defvar \$toWhereGuys	365
14.2.45 defun getMsgToWhere	365
14.2.46 defun toFile?	365
14.2.47 defun alreadyOpened?	365
14.2.48 defun setMsgForcedAttrList	366
14.2.49 defun setMsgForcedAttr	366
14.2.50 defvar \$attrCats	366
14.2.51 defun whichCat	367
14.2.52 defun setMsgCatlessAttr	367
14.2.53 defun putDatabaseStuff	367
14.2.54 defun getMsgInfoFromKey	368
14.2.55 defun setMsgUnforcedAttrList	368
14.2.56 defun setMsgUnforcedAttr	369
14.2.57 defvar \$imPrTagGuys	369
14.2.58 defun initImPr	369
14.2.59 defun initToWhere	370
14.2.60 defun ncBug	370
14.2.61 defun processMsgList	371
14.2.62 defun erMsgSort	371
14.2.63 defun erMsgCompare	372
14.2.64 defun compareposns	372
14.2.65 defun erMsgSep	372
14.2.66 defun makeMsgFromLine	373
14.2.67 defun rep	373
14.2.68 defun getLinePos	374
14.2.69 defun getLineText	374

14.2.70 defun queueUpErrors	374
14.2.71 defun thisPosIsLess	376
14.2.72 defun thisPosIsEqual	376
14.2.73 defun redundant	376
14.2.74 defvar \$repGuys	377
14.2.75 defun msgNoRep?	377
14.2.76 defun sameMsg?	378
14.2.77 defun processChPosesForOneLine	378
14.2.78 defun poCharPosn	379
14.2.79 defun makeLeaderMsg	379
14.2.80 defun posPointers	380
14.2.81 defun getMsgPos2	380
14.2.82 defun insertPos	381
14.2.83 defun putFTText	381
14.2.84 defun From	382
14.2.85 defun To	382
14.2.86 defun FromTo	382
15 The Interpreter Syntax	385
15.1 syntax assignment	385
15.2 syntax blocks	388
15.3 system clef	390
15.4 syntax collection	391
15.5 syntax for	393
15.6 syntax if	397
15.7 syntax iterate	399
15.8 syntax leave	400
15.9 syntax parallel	401
15.10 syntax repeat	404
15.11 syntax suchthat	408
15.12 syntax syntax	409
15.13 syntax while	409
16 Abstract Syntax Trees (ptrees)	413
16.0.1 defun Construct a leaf token	413
16.0.2 defun Return a part of a node	414
16.0.3 defun Compare a part of a node	414
16.0.4 defun pfNoPosition?	414
16.0.5 defun poNoPosition?	415
16.0.6 defun tokType	415
16.0.7 defun tokPart	415
16.0.8 defun tokPosn	415
16.0.9 defun pfNoPosition	416
16.0.10 defun poNoPosition	416

17 Attributed Structures	417
17.0.11 defun ncTag	417
17.0.12 defun ncAlist	417
17.0.13 defun ncEltQ	418
17.0.14 defun ncPutQ	418
18 System Command Handling	421
18.1 Variables Used	423
18.1.1 defvar \$systemCommands	423
18.1.2 defvar \$syscommands	424
18.1.3 defvar \$noParseCommands	424
18.2 Functions	425
18.2.1 defun handleNoParseCommands	425
18.2.2 defun Handle a top level command	426
18.2.3 defun Split block into option block	427
18.2.4 defun Tokenize a system command	427
18.2.5 defun Handle system commands	428
18.2.6 defun Select commands matching this user level	428
18.2.7 defun No command begins with this string	429
18.2.8 defun No option begins with this string	429
18.2.9 defvar \$oldline	429
18.2.10 defun No command/option begins with this string	429
18.2.11 defun Option not available at this user level	430
18.2.12 defun Command not available at this user level	430
18.2.13 defun Command not available error message	430
18.2.14 defun satisfiesUserLevel	431
18.2.15 defun hasOption	431
18.2.16 defun terminateSystemCommand	432
18.2.17 defun Terminate a system command	432
18.2.18 defun commandAmbiguityError	432
18.2.19 defun getParserMacroNames	433
18.2.20 defun clearParserMacro	433
18.2.21 defun displayMacro	433
18.2.22 defun displayWorkspaceNames	434
18.2.23 defun getWorkspaceNames	435
18.2.24 defun fixObjectForPrinting	436
18.2.25 defun displayProperties,sayFunctionDeps	436
18.2.26 defun displayValue	439
18.2.27 defun displayType	440
18.2.28 defun getAndSay	441
18.2.29 defun displayProperties	441
18.2.30 defun displayParserMacro	444
18.2.31 defun displayCondition	445
18.2.32 defun interpFunctionDepAlists	445
18.2.33 defun displayModemap	446
18.2.34 defun displayMode	446

18.2.35 defun Split into tokens delimited by spaces	447
18.2.36 defun Convert string tokens to their proper type	447
18.2.37 defun Is the argument string an integer?	448
18.2.38 defun Handle parsed system commands	448
18.2.39 defun Parse a system command	449
18.2.40 defun Get first word in a string	449
18.2.41 defun Unabbreviate keywords in commands	449
18.2.42 defun The command is ambiguous error	450
18.2.43 defun Remove the spaces surrounding a string	451
18.2.44 defun Remove the lisp command prefix	451
18.2.45 defun Handle the)lisp command	452
18.2.46 defun The)boot command is no longer supported	452
18.2.47 defun Handle the)system command	452
18.2.48 defun Handle the)synonym command	453
18.2.49 defun Handle the synonym system command	453
18.2.50 defun printSynonyms	454
18.2.51 defun Print a list of each matching synonym	454
18.2.52 defvar \$tokenCommands	455
18.2.53 defvar \$InitialCommandSynonymAlist	456
18.2.54 defun Print the current version information	456
18.2.55 defvar \$CommandSynonymAlist	458
18.2.56 defun nclloopCommand	458
18.2.57 defun nclloopPrefix?	459
18.2.58 defun selectOptionLC	459
18.2.59 defun selectOption	459
19)abbreviations help page Command	461
19.1 abbreviations help page man page	461
19.2 Functions	463
19.2.1 defun abbreviations	463
19.2.2 defun abbreviationsSpad2Cmd	463
19.2.3 defun listConstructorAbbreviations	464
20)boot help page Command	467
20.1 boot help page man page	467
20.2 Functions	468
21)browse help page Command	469
21.1 browse help page man page	469
21.2 Overview	469
21.3 Browsers, MathML, and Fonts	470
21.4 The axServer/multiServ loop	471
21.5 The)browse command	472
21.6 Variables Used	473
21.7 Functions	473
21.8 The server support code	473

22)cd help page Command	475
22.1 cd help page man page	475
22.2 Variables Used	476
22.3 Functions	476
23)clear help page Command	477
23.1 clear help page man page	477
23.2 Variables Used	479
23.2.1 defvar \$clearOptions	479
23.3 Functions	479
23.3.1 defun clear	479
23.3.2 defvar \$clearExcept	479
23.3.3 defun clearSpad2Cmd	480
23.3.4 defun clearCmdSortedCaches	481
23.3.5 defvar \$functionTable	481
23.3.6 defun clearCmdCompletely	482
23.3.7 defun clearCmdAll	483
23.3.8 defun clearMacroTable	484
23.3.9 defun clearCmdExcept	484
23.3.10 defun clearCmdParts	484
24)close help page Command	487
24.1 close help page man page	487
24.2 Functions	488
24.2.1 defun queryClients	488
24.2.2 defun close	488
25)compile help page Command	491
25.1 compile help page man page	491
25.2 Functions	493
25.2.1 defvar \$/editfile	493
26)copyright help page Command	495
26.1 copyright help page man page	495
26.2 Functions	500
26.2.1 defun copyright	500
26.2.2 defun trademark	501
27)credits help page Command	503
27.1 credits help page man page	503
27.2 Variables Used	503
27.3 Functions	503
27.3.1 defun credits	503

28)describe help page Command	505
28.1 describe help page man page	505
28.1.1 defvar \$describeOptions	506
28.2 Functions	506
28.2.1 defun Print comment strings from algebra libraries	506
28.2.2 defun describeSpad2Cmd	506
28.2.3 defun cleanline	507
28.2.4 defun flatten	509
29)display help page Command	511
29.1 display help page man page	511
29.1.1 defvar \$displayOptions	513
29.2 Functions	513
29.2.1 defun display	513
29.2.2 displaySpad2Cmd	513
29.2.3 defun abbQuery	514
29.2.4 defun displayOperations	515
29.2.5 defun yesanswer	515
29.2.6 defun displayMacros	516
29.2.7 defun sayExample	517
29.2.8 defun cleanupLine	518
30)edit help page Command	521
30.1 edit help page man page	521
30.2 Functions	522
30.2.1 defun edit	522
30.2.2 defun editSpad2Cmd	522
30.2.3 defun Implement the)edit command	523
30.2.4 defun updateSourceFiles	524
31)fin help page Command	525
31.1 fin help page man page	525
31.1.1 defun Exit from the interpreter to lisp	526
31.2 Functions	526
32)frame help page Command	527
32.1 frame help page man page	527
32.2 Variables Used	529
32.2.1 Primary variables	529
32.2.2 Used variables	530
32.3 Data Structures	530
32.3.1 Frames and the Interpreter Frame Ring	530
32.4 Accessor Functions	530
32.4.1 0th Frame Component – frameName	530
32.4.2 defun frameName	530
32.4.3 1st Frame Component – frameInteractive	531

32.4.4	2nd Frame Component – frameIOIndex	531
32.4.5	3rd Frame Component – frameHiFiAccess	531
32.4.6	4th Frame Component – frameHistList	531
32.4.7	5th Frame Component – frameHistListLen	532
32.4.8	6th Frame Component – frameHistListAct	532
32.4.9	7th Frame Component – frameHistRecord	532
32.4.10	8th Frame Component – frameHistoryTable	532
32.4.11	9th Frame Component – frameExposureData	533
32.5	Functions	533
32.5.1	Initializing the Interpreter Frame Ring	533
32.5.2	Creating a List of all of the Frame Names	534
32.5.3	Get Named Frame Environment (aka Interactive)	534
32.5.4	Create a new, empty Interpreter Frame	534
32.5.5	Collecting up the Environment into a Frame	535
32.5.6	Update from the Current Frame	536
32.5.7	Find a Frame in the Frame Ring by Name	537
32.5.8	Update the Current Interpreter Frame	537
32.5.9	Move to the next Interpreter Frame in Ring	538
32.5.10	Change to the Named Interpreter Frame	538
32.5.11	Move to the previous Interpreter Frame in Ring	539
32.5.12	Add a New Interpreter Frame	539
32.5.13	Close an Interpreter Frame	540
32.5.14	Display the Frame Names	541
32.5.15	Import items from another frame	541
32.5.16	The top level frame command	543
32.5.17	The top level frame command handler	544
32.6	Frame File Messages	545
33)help help page Command	547
33.1	help help page man page	547
33.2	Functions	550
33.2.1	The top level help command	550
33.2.2	The top level help command handler	550
33.2.3	defun newHelpSpad2Cmd	550
34)history help page Command	553
34.1	history help page man page	553
34.2	Initialized history variables	556
34.2.1	defvar \$oldHistoryFileName	556
34.2.2	defvar \$historyFileType	557
34.2.3	defvar \$historyDirectory	557
34.2.4	defvar \$useInternalHistoryTable	557
34.3	Data Structures	557
34.4	Functions	557
34.4.1	defun makeHistFileName	557
34.4.2	defun oldHistFileName	558

34.4.3	defun histFileName	558
34.4.4	defun histInputFileName	558
34.4.5	defun initHist	559
34.4.6	defun initHistList	559
34.4.7	The top level history command	560
34.4.8	The top level history command handler	560
34.4.9	defun setHistoryCore	562
34.4.10	defvar \$underbar	564
34.4.11	defun writeInputLines	565
34.4.12	defun resetInCoreHist	566
34.4.13	defun changeHistListLen	567
34.4.14	defun updateHist	567
34.4.15	defun updateInCoreHist	568
34.4.16	defun putHist	568
34.4.17	defun recordNewValue	569
34.4.18	defun recordNewValue0	569
34.4.19	defun recordOldValue	570
34.4.20	defun recordOldValue0	570
34.4.21	defun undoInCore	570
34.4.22	defun undoChanges	571
34.4.23	defun undoFromFile	572
34.4.24	defun saveHistory	573
34.4.25	defun restoreHistory	575
34.4.26	defun setIOindex	577
34.4.27	defun showInput	577
34.4.28	defun showInOut	578
34.4.29	defun fetchOutput	578
34.4.30	Read the history file using index n	579
34.4.31	Write information of the current step to history file	580
34.4.32	Disable history if an error occurred	581
34.4.33	defun writeHistModesAndValues	581
34.5	Lisplib output transformations	582
34.5.1	defun spadwrite0	582
34.5.2	defun Random write to a stream	582
34.5.3	defun spadwrite	583
34.5.4	defun spadread	583
34.5.5	defun Random read a key from a stream	583
34.5.6	defun unwritable?	584
34.5.7	defun writifyComplain	584
34.5.8	defun safeWritify	585
34.5.9	defun writify,writifyInner	585
34.5.10	defun writify	588
34.5.11	defun spadClosure?	589
34.5.12	defun dewritify,is?	589
34.5.13	defvar \$NonNullStream	589
34.5.14	defvar \$NullStream	590

34.5.15 defun dewritify,dewritifyInner	590
34.5.16 defun dewritify	593
34.5.17 defun ScanOrPairVec,ScanOrInner	594
34.5.18 defun ScanOrPairVec	594
34.5.19 defun gensymInt	595
34.5.20 defun charDigitVal	595
34.5.21 defun histFileErase	596
34.6 History File Messages	596
35)include help page Command	599
35.1 include help page man page	599
35.2 Functions	599
35.2.1 defun ncloopInclude1	599
35.2.2 Returns the first non-blank substring of the given string	600
35.2.3 Open the include file and read it in	600
35.2.4 Return the include filename	600
35.2.5 Return the next token	601
36)library help page Command	603
36.1 library help page man page	603
37)lisp help page Command	605
37.1 lisp help page man page	605
37.2 Functions	606
38)load help page Command	607
38.1 load help page man page	607
38.1.1 defun The)load command (obsolete)	607
39)ltrace help page Command	609
39.1 ltrace help page man page	609
39.1.1 defun The top level)ltrace function	610
39.2 Variables Used	610
39.3 Functions	610
40)pquit help page Command	611
40.1 pquit help page man page	611
40.2 Functions	612
40.2.1 The top level pquit command	612
40.2.2 The top level pquit command handler	612
41)quit help page Command	615
41.1 quit help page man page	615
41.2 Functions	616
41.2.1 The top level quit command	616
41.2.2 The top level quit command handler	616
41.2.3 Leave the Axiom interpreter	617

42)read help page Command	619
42.1 read help page man page	619
42.1.1 defun The)read command	620
42.1.2 defun Implement the)read command	620
42.1.3 defun /read	622
43)savesystem help page Command	623
43.1 savesystem help page man page	623
43.1.1 defun The)savesystem command	624
44)set help page Command	625
44.1 set help page man page	625
44.2 Overview	626
44.3 Variables Used	627
44.4 Functions	627
44.4.1 Initialize the set variables	627
44.4.2 Reset the workspace variables	628
44.4.3 Display the set option information	629
44.4.4 Display the set variable settings	631
44.4.5 Translate options values to t or nil	632
44.4.6 Translate t or nil to option values	633
44.5 The list structure	633
44.6 breakmode	634
44.6.1 defvar \$BreakMode	635
44.7 debug	635
44.8 debug lambda type	636
44.8.1 defvar \$lambdatype	636
44.9 debug dalymode	636
44.9.1 defvar \$dalymode	637
44.10compile	637
44.11compile output	638
44.12Variables Used	638
44.13Functions	638
44.13.1 The set output command handler	638
44.13.2 Describe the set output library arguments	639
44.13.3 defvar \$output-library	639
44.13.4 Open the output library	640
44.14compile input	640
44.15Variables Used	641
44.16Functions	641
44.16.1 The set input library command handler	641
44.16.2 Describe the set input library arguments	642
44.16.3 Add the input library to the list	642
44.16.4 defvar \$input-libraries	642
44.16.5 Drop an input library from the list	643
44.17expose	643

44.18	Variables Used	644
44.18.1	defvar \$globalExposureGroupAlist	644
44.18.2	defvar \$localExposureDataDefault	670
44.18.3	defvar \$localExposureData	670
44.19	Functions	670
44.19.1	The top level set expose command handler	670
44.19.2	The top level set expose add command handler	671
44.19.3	Expose a group	672
44.19.4	The top level set expose add constructor handler	674
44.19.5	The top level set expose drop handler	675
44.19.6	The top level set expose drop group handler	676
44.19.7	The top level set expose drop constructor handler	677
44.19.8	Display exposed groups	678
44.19.9	Display exposed constructors	678
44.19.10	Display hidden constructors	679
44.20	functions	679
44.21	functions cache	680
44.22	Variables Used	681
44.22.1	defvar \$cacheAlist	681
44.23	Functions	681
44.23.1	The top level set functions cache handler	681
44.23.2	defvar \$compileDontDefineFunctions	685
44.24	functions recurrence	685
44.24.1	defvar \$compileRecurrence	686
44.25	fortran	686
44.25.1	ints2floats	687
44.25.2	defvar \$fortInts2Floats	687
44.25.3	fortindent	688
44.25.4	defvar \$fortIndent	688
44.25.5	fortlength	689
44.25.6	defvar \$fortLength	689
44.25.7	typedecs	689
44.25.8	defvar \$printFortranDecs	690
44.25.9	defaulttype	690
44.25.10	defvar \$defaultFortranType	690
44.25.11	precision	691
44.25.12	defvar \$fortranPrecision	691
44.25.13	intrinsic	692
44.25.14	defvar \$useIntrinsicFunctions	692
44.25.15	explength	692
44.25.16	defvar \$maximumFortranExpressionLength	693
44.25.17	segment	693
44.25.18	defvar \$fortranSegment	694
44.25.19	optlevel	694
44.25.20	defvar \$fortranOptimizationLevel	694
44.25.21	startindex	695

44.25.22	defvar \$fortranArrayStartingIndex	695
44.25.23	calling	695
44.25.24	defvar \$fortranTmpDir	696
44.25.25	The top level set fortran calling tempfile handler	697
44.25.26	Validate the output directory	698
44.25.27	Describe the set fortran calling tempfile	698
44.25.28	defvar \$fortranDirectory	699
44.25.29	defun setFortDir	699
44.25.30	defun describeSetFortDir	700
44.25.31	defvar \$fortranLibraries	701
44.25.32	defun setLinkerArgs	702
44.25.33	defun describeSetLinkerArgs	702
44.26	kernel	703
44.26.1	kernelwarn	703
44.26.2	defun protectedSymbolsWarning	704
44.26.3	defun describeProtectedSymbolsWarning	704
44.26.4	kernelprotect	705
44.26.5	defun protectSymbols	705
44.26.6	defun describeProtectSymbols	706
44.27	hyperdoc	706
44.27.1	fullscreen	707
44.27.2	defvar \$fullScreenSysVars	707
44.27.3	mathwidth	708
44.27.4	defvar \$historyDisplayWidth	708
44.28	help	708
44.28.1	fullscreen	709
44.28.2	defvar \$useFullScreenHelp	709
44.29	history	710
44.29.1	defvar \$HiFiAccess	710
44.30	messages	710
44.30.1	any	712
44.30.2	defvar \$printAnyIfTrue	712
44.30.3	autoload	713
44.30.4	defvar \$printLoadMsgs	713
44.30.5	bottomup	713
44.30.6	defvar \$reportBottomUpFlag	714
44.30.7	coercion	714
44.30.8	defvar \$reportCoerceIfTrue	714
44.30.9	dropmap	715
44.30.10	defvar \$displayDroppedMap	715
44.30.11	expose	716
44.30.12	defvar \$giveExposureWarning	716
44.30.13	file	716
44.30.14	defvar \$printMsgsToFile	717
44.30.15	frame	717
44.30.16	defvar \$frameMessages	718

44.30.1	highlighting	718
44.30.1	defvar \$highlightAllowed	718
44.30.1	instant	719
44.30.2	defvar \$reportInstantiations	719
44.30.2	instead	720
44.30.2	defvar \$reportEachInstantiation—	720
44.30.2	interponly	720
44.30.2	defvar \$reportInterpOnly	721
44.30.2	naglink	721
44.30.2	defvar \$nagMessages	722
44.30.2	number	722
44.30.2	defvar \$displayMsgNumber	722
44.30.2	prompt	723
44.30.3	defvar \$inputPromptType	723
44.30.3	selection	724
44.30.3	set	724
44.30.3	defvar \$displaySetValue	725
44.30.3	startup	725
44.30.3	defvar \$displayStartMsgs	726
44.30.3	summary	726
44.30.3	defvar \$printStatisticsSummaryIfTrue	726
44.30.3	testing	727
44.30.3	defvar \$testingSystem	727
44.30.4	time	728
44.30.4	defvar \$printTimeIfTrue	728
44.30.4	type	729
44.30.4	defvar \$printTypeIfTrue	729
44.30.4	void	729
44.30.4	defvar \$printVoidIfTrue	730
44.31	naglink	730
44.31.1	host	731
44.31.2	defvar \$nagHost	731
44.31.3	defun setNagHost	732
44.31.4	defun describeSetNagHost	732
44.31.5	persistence	732
44.31.6	defvar \$fortPersistence	733
44.31.7	defun setFortPers	733
44.31.8	defun describeFortPersistence	734
44.31.9	messages	735
44.31.1	double	735
44.31.1	defvar \$nagEnforceDouble	735
44.32	output	736
44.32.1	abbreviate	737
44.32.2	defvar \$abbreviateTypes	737
44.32.3	algebra	738
44.32.4	defvar \$algebraFormat	738

44.32.5	defvar \$algebraOutputFile	739
44.32.6	defvar \$algebraOutputStream	739
44.32.7	defun setOutputAlgebra	740
44.32.8	defun describeSetOutputAlgebra	742
44.32.9	characters	743
44.32.10	defun setOutputCharacters	743
44.32.11	fortran	745
44.32.12	defvar \$fortranFormat	746
44.32.13	defvar \$fortranOutputFile	746
44.32.14	defun setOutputFortran	747
44.32.15	defun describeSetOutputFortran	749
44.32.16	fraction	750
44.32.17	defvar \$fractionDisplayType	750
44.32.18	length	751
44.32.19	defvar \$margin	751
44.32.20	defvar \$linelength	751
44.32.21	mathml	752
44.32.22	defvar \$mathmlFormat	752
44.32.23	defvar \$mathmlOutputFile	753
44.32.24	defun setOutputMathml	753
44.32.25	defun describeSetOutputMathml	755
44.32.26	html	756
44.32.27	defvar \$htmlFormat	757
44.32.28	defvar \$htmlOutputFile	757
44.32.29	defun setOutputHtml	758
44.32.30	defun describeSetOutputHtml	760
44.32.31	bpenmath	761
44.32.32	defvar \$openMathFormat	762
44.32.33	defvar \$openMathOutputFile	762
44.32.34	defun setOutputOpenMath	763
44.32.35	defun describeSetOutputOpenMath	765
44.32.36	script	766
44.32.37	defvar \$formulaFormat	766
44.32.38	defvar \$formulaOutputFile	766
44.32.39	defun setOutputFormula	767
44.32.40	defun describeSetOutputFormula	769
44.32.41	scripts	770
44.32.42	defvar \$linearFormatScripts	771
44.32.43	showeditor	771
44.32.44	defvar \$useEditorForShowOutput	771
44.32.45	tex	772
44.32.46	defvar \$texFormat	773
44.32.47	defvar \$texOutputFile	773
44.32.48	defun setOutputTex	773
44.32.49	defun describeSetOutputTex	776
44.33	quit	776

44.33.1 defvar \$quitCommandType	777
44.34 streams	777
44.34.1 calculate	778
44.34.2 defvar \$streamCount	778
44.34.3 defun setStreamsCalculate	779
44.34.4 defun describeSetStreamsCalculate	779
44.34.5 showall	780
44.34.6 defvar \$streamsShowAll	780
44.35 system	780
44.35.1 functioncode	781
44.35.2 defvar \$reportCompilation	781
44.35.3 optimization	782
44.35.4 defvar \$reportOptimization	782
44.35.5 prettyprint	783
44.35.6 defvar \$prettyprint	783
44.36 userlevel	784
44.36.1 defvar \$UserLevel	784
44.36.2 defvar \$setOptionNames	785
44.37 Set code	785
44.37.1 defun set	785
44.37.2 defun set1	786
45)show help page Command	791
45.1 show help page man page	791
45.1.1 defun The)show command	792
45.1.2 defun The internal)show command	792
45.1.3 defun reportOperations	793
45.1.4 defun reportOpsFromLisplib0	795
45.1.5 defun reportOpsFromLisplib1	795
45.1.6 defun reportOpsFromLisplib	796
45.1.7 defun displayOperationsFromLisplib	798
45.1.8 defun reportOpsFromUnitDirectly0	799
45.1.9 defun reportOpsFromUnitDirectly	799
45.1.10 defun reportOpsFromUnitDirectly1	801
45.1.11 defun sayShowWarning	802
46)spool help page Command	803
46.1 spool help page man page	803
47)summary help page Command	805
47.1 summary help page man page	805
47.1.1 defun summary	806

48)synonym help page Command	807
48.1 synonym help page man page	807
48.1.1 defun The)synonym command	808
48.1.2 defun The)synonym command implementation	808
48.1.3 defun Return a sublist of applicable synonyms	809
48.1.4 defun Get the system command from the input line	809
48.1.5 defun Remove system keyword	810
48.1.6 defun processSynonymLine	811
49)system help page Command	813
49.1 system help page man page	813
50)trace help page Command	815
50.1 trace help page man page	815
50.1.1 The trace global variables	819
50.1.2 defvar \$traceNoisely	820
50.1.3 defvar \$reportSpadTrace	820
50.1.4 defvar \$optionAlist	820
50.1.5 defvar \$tracedMapSignatures	820
50.1.6 defvar \$traceOptionList	820
50.1.7 defun trace	821
50.1.8 defun traceSpad2Cmd	821
50.1.9 defun trace1	822
50.1.10 defun getTraceOptions	826
50.1.11 defun saveMapSig	827
50.1.12 defun getMapSig	827
50.1.13 defun getTraceOption,hn	827
50.1.14 defun getTraceOption	828
50.1.15 defun traceOptionError	831
50.1.16 defun resetTimers	832
50.1.17 defun resetSpacers	832
50.1.18 defun resetCounters	832
50.1.19 defun ptimers	833
50.1.20 defun pspacers	833
50.1.21 defun pcounters	834
50.1.22 defun transOnlyOption	834
50.1.23 defun stackTraceOptionError	835
50.1.24 defun removeOption	835
50.1.25 defun domainToGenvar	835
50.1.26 defun genDomainTraceName	836
50.1.27 defun untrace	836
50.1.28 defun transTraceItem	837
50.1.29 defun removeTracedMapSigs	838
50.1.30 defun coerceTraceArgs2E	838
50.1.31 defun coerceSpadArgs2E	839
50.1.32 defun subTypes	840

50.1.33 defun coerceTraceFunValue2E	841
50.1.34 defun coerceSpadFunValue2E	842
50.1.35 defun isListOfIdentifiers	842
50.1.36 defun isListOfIdentifiersOrStrings	843
50.1.37 defun getMapSubNames	843
50.1.38 defun getPreviousMapSubNames	844
50.1.39 defun lassocSub	845
50.1.40 defun rassocSub	845
50.1.41 defun isUncompiledMap	845
50.1.42 defun isInterpOnlyMap	846
50.1.43 defun augmentTraceNames	846
50.1.44 defun isSubForRedundantMapName	847
50.1.45 defun untraceMapSubNames	847
50.1.46 defun funfind,LAM	848
50.1.47 defmacro funfind	848
50.1.48 defun isDomainOrPackage	849
50.1.49 defun isTraceGensym	849
50.1.50 defun spadTrace,g	849
50.1.51 defun spadTrace,isTraceable	849
50.1.52 defun spadTrace	850
50.1.53 defun traceDomainLocalOps	854
50.1.54 defun untraceDomainLocalOps	854
50.1.55 defun traceDomainConstructor	854
50.1.56 defun untraceDomainConstructor,keepTraced?	856
50.1.57 defun untraceDomainConstructor	857
50.1.58 defun flattenOperationAlist	857
50.1.59 defun mapLetPrint	858
50.1.60 defun letPrint	859
50.1.61 defun Identifier beginning with a sharpsign-number?	860
50.1.62 defun Identifier beginning with a sharpsign?	860
50.1.63 defun isgenvar	860
50.1.64 defun letPrint2	861
50.1.65 defun letPrint3	862
50.1.66 defun getAliasIfTracedMapParameter	863
50.1.67 defun getBpiNameIfTracedMap	864
50.1.68 defun hasPair	865
50.1.69 defun shortenForPrinting	865
50.1.70 defun spadTraceAlias	865
50.1.71 defun getOption	866
50.1.72 defun reportSpadTrace	866
50.1.73 defun orderBySlotNumber	867
50.1.74 defun /tracereply	868
50.1.75 defun spadReply,printName	868
50.1.76 defun spadReply	869
50.1.77 defun spadUntrace	869
50.1.78 defun prTraceNames,fn	871

50.1.79 defun prTraceNames	872
50.1.80 defvar \$constructors	872
50.1.81 defun traceReply	873
50.1.82 defun addTraceItem	876
50.1.83 defun ?t	876
50.1.84 defun tracelet	877
50.1.85 defun breaklet	878
50.1.86 defun stupidIsSpadFunction	880
50.1.87 defun break	880
50.1.88 defun compileBoot	880
51)undo help page Command	883
51.1 undo help page man page	883
51.2 Data Structures	884
51.3 Functions	885
51.3.1 Initial Undo Variables	885
51.3.2 defvar \$undoFlag	885
51.3.3 defvar \$frameRecord	885
51.3.4 defvar \$previousBindings	885
51.3.5 defvar \$reportUndo	886
51.3.6 defun undo	886
51.3.7 defun recordFrame	887
51.3.8 defun diffAlist	888
51.3.9 defun reportUndo	891
51.3.10 defun clearFrame	893
51.3.11 Undo previous n commands	893
51.3.12 defun undoSteps	894
51.3.13 defun undoSingleStep	895
51.3.14 defun undoLocalModemapHack	897
51.3.15 Remove undo lines from history write	897
52)what help page Command	901
52.1 what help page man page	901
52.1.1 defvar \$whatOptions	903
52.1.2 defun what	903
52.1.3 defun whatSpad2Cmd,fixpat	903
52.1.4 defun whatSpad2Cmd	904
52.1.5 defun Show keywords for)what command	905
52.1.6 defun The)what commands implementation	905
52.1.7 defun Find all names contained in a pattern	906
52.1.8 defun Find function of names contained in pattern	907
52.1.9 defun satisfiesRegularExpressions	907
52.1.10 defun filterAndFormatConstructors	908
52.1.11 defun whatConstructors	909
52.1.12 Display all operation names containing the fragment	909

53)with help page Command	911
53.1 with help page man page	911
53.1.1 defun with	911
54)workfiles help page Command	913
54.1 workfiles help page man page	913
54.1.1 defun workfiles	913
54.1.2 defun workfilesSpad2Cmd	913
55)zsystemdevelopment help page Command	917
55.1 zsystemdevelopment help page man page	917
55.1.1 defun zsystemdevelopment	917
55.1.2 defun zsystemDevelopmentSpad2Cmd	917
55.1.3 defun zsystemdevelopment1	918
56 Handling input files	921
56.0.4 defun Handle .axiom.input file	921
56.0.5 defun /rq	921
56.0.6 defun /rf	922
56.0.7 defvar \$boot-line-stack	922
56.0.8 defvar \$in-stream	922
56.0.9 defvar \$out-stream	922
56.0.10 defvar \$file-closed	923
56.0.11 defvar \$echo-meta	923
56.0.12 defvar \$noSubsumption	923
56.0.13 defvar \$envHashTable	923
56.0.14 defun Dynamically add bindings to the environment	923
56.0.15 defun Fetch a property list for a symbol from CategoryFrame	924
56.0.16 defun Search for a binding in the environment list	925
56.0.17 defun Search for a binding in the current environment	925
56.0.18 defun searchTailEnv	926
57 File Parsing	927
57.0.19 defun Bind a variable in the interactive environment	927
57.0.20 defvar \$line-handler	927
57.0.21 defvar \$spad-errors	927
57.0.22 defvar \$xtokenreader	928
57.0.23 defun Initialize the spad reader	928
57.0.24 defun ioclear	929
57.0.25 defun Set boot-line-stack to nil	929
58 Handling output	931
58.1 Special Character Tables	931
58.1.1 defvar \$defaultSpecialCharacters	931
58.1.2 defvar \$plainSpecialCharacters0	932
58.1.3 defvar \$plainSpecialCharacters1	932

58.1.4	defvar \$plainSpecialCharacters2	933
58.1.5	defvar \$plainSpecialCharacters3	933
58.1.6	defvar \$plainRTspecialCharacters	934
58.1.7	defvar \$RTspecialCharacters	934
58.1.8	defvar \$specialCharacters	935
58.1.9	defvar \$specialCharacterAlist	935
58.1.10	defun Look up a special character code for a symbol	936
59	Stream and File Handling	937
59.0.11	defun make-instream	937
59.0.12	defun make-outstream	937
59.0.13	defun make-appendstream	938
59.0.14	defun defiostream	938
59.0.15	defun shut	938
59.0.16	defun eofp	939
59.0.17	defun makeStream	939
59.0.18	defun Construct a new input file name	939
59.0.19	defun getDirectoryList	940
59.0.20	defun probeName	940
59.0.21	defun makeFullNamestring	941
59.0.22	defun Replace a file by erase and rename	941
60	The Spad Server Mechanism	943
60.0.23	defun openserver	943
61	Axiom Build-time Functions	945
61.0.24	defun spad-save	945
62	Exposure Groups	947
63	Databases	949
63.1	Database structure	949
63.1.1	kaf File Format	949
63.1.2	Database Files	950
63.1.3	defstruct \$database	952
63.1.4	defvar \$*defaultdomain-list*	953
63.1.5	defvar \$*operation-hash*	953
63.1.6	defvar \$*hasCategory-hash*	953
63.1.7	defvar \$*miss*	954
63.1.8	Database streams	954
63.1.9	defvar \$*compressvector*	954
63.1.10	defvar \$*compressVectorLength*	954
63.1.11	defvar \$*compress-stream*	955
63.1.12	defvar \$*compress-stream-stamp*	955
63.1.13	defvar \$*interp-stream*	955
63.1.14	defvar \$*interp-stream-stamp*	955

63.1.15 defvar <code>*\$operation-stream*</code>	955
63.1.16 defvar <code>*\$operation-stream-stamp*</code>	956
63.1.17 defvar <code>*\$browse-stream*</code>	956
63.1.18 defvar <code>*\$browse-stream-stamp*</code>	956
63.1.19 defvar <code>*\$category-stream*</code>	956
63.1.20 defvar <code>*\$category-stream-stamp*</code>	957
63.1.21 defvar <code>*\$allconstructors*</code>	957
63.1.22 defvar <code>*\$allOperations*</code>	957
63.1.23 defun Reset all hash tables before saving system	957
63.1.24 defun Preload algebra into saved system	958
63.1.25 defun Open the interp database	960
63.1.26 defun Open the browse database	962
63.1.27 defun Open the category database	963
63.1.28 defun Open the operations database	964
63.1.29 defun Add operations from newly compiled code	964
63.1.30 defun Show all database attributes of a constructor	965
63.1.31 defun Set a value for a constructor key in the database	966
63.1.32 defun Delete a value for a constructor key in the database	967
63.1.33 defun Get constructor information for a database key	967
63.1.34 defun The <code>)library</code> top level command	971
63.1.35 defun Read a local filename and update the hash tables	971
63.1.36 defun Update the database from an <code>nrllib</code> <code>index.kaf</code> file	973
63.1.37 defun Make new databases	975
63.1.38 defun Construct the proper database full pathname	979
63.1.39 <code>compress.daase</code>	979
63.1.40 defun Set up compression vectors for the databases	979
63.1.41 defvar <code>*\$attributes*</code>	980
63.1.42 defun Write out the compress database	980
63.1.43 defun Compress an expression using the compress vector	982
63.1.44 defun Uncompress an expression using the compress vector	982
63.1.45 Building the <code>interp.daase</code> from hash tables	983
63.1.46 defun Write the interp database	987
63.1.47 Building the <code>browse.daase</code> from hash tables	988
63.1.48 defun Write the browse database	989
63.1.49 Building the <code>category.daase</code> from hash tables	990
63.1.50 defun Write the category database	990
63.1.51 Building the <code>operation.daase</code> from hash tables	991
63.1.52 defun Write the operations database	991
63.1.53 Database support operations	991
63.1.54 defun Data preloaded into the image at build time	991
63.1.55 defun Return all constructors	992
63.1.56 defun Return all operations	992

65 Special Lisp Functions	995
65.1 Axiom control structure macros	995
65.1.1 defun put	995
65.1.2 defmacro while	995
65.1.3 defmacro whileWithResult	996
65.2 Filename Handling	996
65.2.1 defun namestring	996
65.2.2 defun pathnameName	996
65.2.3 defun pathnameType	996
65.2.4 defun pathnameTypeId	997
65.2.5 defun mergePathnames	997
65.2.6 defun pathnameDirectory	998
65.2.7 defun Axiom pathnames	998
65.2.8 defun makePathname	998
65.2.9 defun Delete a file	999
65.2.10 defun wrap	999
65.2.11 defun lotsof	999
65.2.12 defmacro startsId?	1000
65.2.13 defun hput	1000
65.2.14 defmacro hget	1000
65.2.15 defun hkeys	1000
65.2.16 defun digitp	1001
65.2.17 defun pname	1001
65.2.18 defun size	1001
65.2.19 defun strpos	1002
65.2.20 defun strposl	1002
65.2.21 defun qenum	1002
65.2.22 defmacro identp	1003
65.2.23 defun concat	1003
65.2.24 defun functionp	1003
65.2.25 defun brightprint	1004
65.2.26 defun brightprint-0	1004
65.2.27 defun member	1004
65.2.28 defun messageprint	1004
65.2.29 defun messageprint-1	1005
65.2.30 defun messageprint-2	1005
65.2.31 defun sayBrightly1	1005
65.2.32 defmacro assq	1006
66 Common Lisp Algebra Support	1007
66.1 SingleInteger	1007
66.1.1 defun qsquotient	1007
66.1.2 defun qsremainder	1008
66.1.3 defmacro qsdifference	1008
66.1.4 defmacro qslessp	1008
66.1.5 defmacro qsadd1	1008

66.1.6	defmacro qssub1	1009
66.1.7	defmacro qsminus	1009
66.1.8	defmacro qsplus	1009
66.1.9	defmacro qstimes	1009
66.1.10	defmacro qsabsval	1010
66.1.11	defmacro qsoddp	1010
66.1.12	defmacro qszerop	1010
66.1.13	defmacro qsmax	1010
66.1.14	defmacro qsmin	1011
66.2	Boolean	1011
66.2.1	defun The Boolean = function support	1011
66.3	IndexedBits	1011
66.3.1	defmacro truth-to-bit	1011
66.3.2	defun IndexedBits new function support	1011
66.3.3	defmacro bit-to-truth	1012
66.3.4	defmacro bvec-elt	1012
66.3.5	defmacro bvec-setelt	1012
66.3.6	defmacro bvec-size	1012
66.3.7	defun IndexedBits concat function support	1012
66.3.8	defun IndexedBits copy function support	1013
66.3.9	defun IndexedBits = function support	1013
66.3.10	defun IndexedBits < function support	1013
66.3.11	defun IndexedBits And function support	1013
66.3.12	defun IndexedBits Or function support	1014
66.3.13	defun IndexedBits xor function support	1014
66.3.14	defun IndexedBits nand function support	1014
66.3.15	defun IndexedBits nor function support	1014
66.3.16	defun IndexedBits not function support	1015
66.4	KeyedAccessFile	1015
66.4.1	defun KeyedAccessFile defstream function support	1015
66.4.2	defun KeyedAccessFile defstream function support	1015
66.5	Table	1016
66.5.1	defun Table InnerTable support	1016
66.6	Plot3d	1016
66.6.1	defvar \$numericFailure	1016
66.6.2	defvar \$oldBreakMode	1017
66.6.3	defmacro trapNumericErrors	1017
66.7	DoubleFloatVector	1017
66.7.1	defmacro dlen	1017
66.7.2	defmacro make-double-vector	1018
66.7.3	defmacro make-double-vector1	1018
66.7.4	defmacro delt	1018
66.7.5	defmacro dsetelt	1018
66.8	ComplexDoubleFloatVector	1019
66.8.1	defmacro make-cdouble-vector	1019
66.8.2	defmacro cdelt	1019

66.8.3	defmacro cdsetelt	1019
66.8.4	defmacro cdlen	1020
66.9	DoubleFloatMatrix	1020
66.9.1	defmacro make-double-matrix	1020
66.9.2	defmacro make-double-matrix1	1020
66.9.3	defmacro daref2	1021
66.9.4	defmacro dsetaref2	1021
66.9.5	defmacro danrows	1021
66.9.6	defmacro dancols	1021
66.10	ComplexDoubleFloatMatrix	1022
66.10.1	defmacro make-cdouble-matrix	1022
66.10.2	defmacro cdaref2	1022
66.10.3	defmacro cdsetaref2	1022
66.10.4	defmacro cdanrows	1023
66.10.5	defmacro cdancols	1023
66.11	Integer	1023
66.11.1	defun Integer divide function support	1023
66.11.2	defun Integer quo function support	1024
66.11.3	defun Integer quo function support	1024
66.11.4	defun Integer random function support	1024
66.12	IndexCard	1025
66.12.1	defun IndexCard origin function support	1025
66.12.2	defun IndexCard origin function support	1025
66.12.3	defun IndexCard elt function support	1025
66.13	OperationsQuery	1026
66.13.1	defun OperationQuery getDatabase function support	1026
66.14	Database	1027
66.14.1	defun Database elt function support	1027
66.15	FileName	1027
66.15.1	defun FileName filename function implementation	1027
66.15.2	defun FileName filename support function	1027
66.15.3	defun FileName directory function implementation	1028
66.15.4	defun FileName directory function support	1028
66.15.5	defun FileName name function implementation	1028
66.15.6	defun FileName extension function implementation	1028
66.15.7	defun FileName exists? function implementation	1029
66.15.8	defun FileName readable? function implementation	1029
66.15.9	defun FileName writeable? function implementation	1029
66.15.10	defun FileName writeable? function support	1029
66.15.11	defun FileName new function implementation	1030
66.16	DoubleFloat	1030
66.16.1	defmacro DFLessThan	1030
66.16.2	defmacro DFUnaryMinus	1031
66.16.3	defmacro DFMinusp	1031
66.16.4	defmacro DFZerop	1031
66.16.5	defmacro DFAdd	1031

66.16.6	defmacro DFSubtract	1032
66.16.7	defmacro DFMultiply	1032
66.16.8	defmacro DFIntegerMultiply	1032
66.16.9	defmacro DFMax	1032
66.16.10	defmacro DFMin	1033
66.16.11	defmacro DFEql	1033
66.16.12	defmacro DFDivide	1033
66.16.13	defmacro DFIntegerDivide	1033
66.16.14	defmacro DFSqrt	1034
66.16.15	defmacro DFLogE	1034
66.16.16	defmacro DFLog	1034
66.16.17	defmacro DFIntegerExpt	1034
66.16.18	defmacro DFExpt	1035
66.16.19	defmacro DFExp	1035
66.16.20	defmacro DFSin	1035
66.16.21	defmacro DFCos	1035
66.16.22	defmacro DFTan	1036
66.16.23	defmacro DFAsin	1036
66.16.24	defmacro DFacos	1036
66.16.25	defmacro DFatan	1036
66.16.26	defmacro DFatan2	1037
66.16.27	defmacro DFSinh	1037
66.16.28	defmacro DFCosh	1037
66.16.29	defmacro DFTanh	1038
66.16.30	defmacro DFAsinh	1038
66.16.31	defmacro DFacosh	1038
66.16.32	defmacro DFatanh	1039
66.16.33	defun Machine specific float numerator	1039
66.16.34	defun Machine specific float denominator	1039
66.16.35	defun Machine specific float sign	1040
66.16.36	defun Machine specific float bit length	1040
66.16.37	defun Decode floating-point values	1040
66.16.38	defun The cotangent routine	1040
66.16.39	defun The inverse cotangent function	1041
66.16.40	defun The secant function	1041
66.16.41	defun The inverse secant function	1041
66.16.42	defun The cosecant function	1042
66.16.43	defun The inverse cosecant function	1042
66.16.44	defun The hyperbolic cosecant function	1042
66.16.45	defun The hyperbolic cotangent function	1043
66.16.46	defun The hyperbolic secant function	1043
66.16.47	defun The inverse hyperbolic cosecant function	1043
66.16.48	defun The inverse hyperbolic cotangent function	1043
66.16.49	defun The inverse hyperbolic secant function	1044

67 NRLIB code.lisp support code	1045
67.0.50 defun makeByteWordVec2	1045
67.0.51 defmacro spadConstant	1045
68 Monitoring execution	1047
68.0.52 defvar \$*monitor-domains*	1053
68.0.53 defvar \$*monitor-nrlibs*	1053
68.0.54 defvar \$*monitor-table*	1054
68.0.55 defstruct \$monitor-data	1054
68.0.56 defstruct \$libstream	1054
68.0.57 defun Initialize the monitor statistics hashtable	1054
68.0.58 defun End the monitoring process, we cannot restart	1055
68.0.59 defun Return a list of the monitor-data structures	1055
68.0.60 defun Add a function to be monitored	1056
68.0.61 defun Remove a function being monitored	1056
68.0.62 defun Enable all (or optionally one) function for monitoring	1056
68.0.63 defun Disable all (optionally one) function for monitoring	1057
68.0.64 defun Reset the table count for the table (or a function)	1057
68.0.65 defun Incr the count of fn by 1	1058
68.0.66 defun Decr the count of fn by 1	1058
68.0.67 defun Return the monitor information for a function	1059
68.0.68 defun Hang a monitor call on all of the defuns in a file	1059
68.0.69 defun Return a list of the functions with zero count fields	1059
68.0.70 defun Return a list of functions with non-zero counts	1060
68.0.71 defun Write out a list of symbols or structures to a file	1060
68.0.72 defun Save the *monitor-table* in loadable form	1061
68.0.73 defun restore a checkpointed file	1061
68.0.74 defun Printing help documentation	1062
68.0.75 Monitoring algebra files	1064
68.0.76 defun Monitoring algebra code.lisp files	1064
68.0.77 defun Monitor autoloaded files	1064
68.0.78 defun Monitor an nrlib	1065
68.0.79 defun Given a monitor-data item, extract the nrlib name	1065
68.0.80 defun Is this an exposed algebra function?	1066
68.0.81 defun Monitor exposed domains	1066
68.0.82 defun Generate a report of the monitored domains	1067
68.0.83 defun Parse an)abbrev expression for the domain name	1068
68.0.84 defun Given a spad file, report all nrlibs it creates	1068
68.0.85 defun Print percent of functions tested	1069
68.0.86 defun Find all monitored symbols containing the string	1069
69 The Interpreter	1071

70 The Global Variables	1101
70.1 Star Global Variables	1101
70.1.1 *eof*	1101
70.1.2 *features*	1101
70.1.3 *package*	1101
70.1.4 *standard-input*	1102
70.1.5 *standard-output*	1102
70.1.6 *top-level-hook*	1102
70.2 Dollar Global Variables	1104
70.2.1 \$boot	1105
70.2.2 coerceFailure	1105
70.2.3 \$currentLine	1105
70.2.4 \$displayStartMsgs	1105
70.2.5 \$e	1105
70.2.6 \$erMsgToss	1105
70.2.7 \$fn	1105
70.2.8 \$frameRecord	1105
70.2.9 \$HiFiAccess	1106
70.2.10 \$HistList	1106
70.2.11 \$HistListAct	1106
70.2.12 \$HistListLen	1106
70.2.13 \$HistRecord	1106
70.2.14 \$historyFileType	1107
70.2.15 \$internalHistoryTable	1107
70.2.16 \$interpreterFrameName	1107
70.2.17 \$interpreterFrameRing	1107
70.2.18 \$InteractiveFrame	1107
70.2.19 \$intRestart	1107
70.2.20 \$intTopLevel	1107
70.2.21 \$IOindex	1108
70.2.22 \$lastPos	1108
70.2.23 \$libQuiet	1108
70.2.24 \$msgDatabaseName	1108
70.2.25 \$ncMsgList	1108
70.2.26 \$newcompErrorCount	1108
70.2.27 \$newspad	1108
70.2.28 \$nopus	1108
70.2.29 \$oldHistoryFileName	1109
70.2.30 \$okToExecuteMachineCode	1109
70.2.31 \$options	1109
70.2.32 \$previousBindings	1109
70.2.33 \$PrintCompilerMessageIfTrue	1109
70.2.34 \$reportUndo	1109
70.2.35 \$spad	1109
70.2.36 \$SpadServer	1110
70.2.37 \$SpadServerName	1110

70.2.38 \$systemCommandFunction	1110
70.2.39 top_level	1110
70.2.40 \$quitTag	1110
70.2.41 \$useInternalHistoryTable	1110
70.2.42 \$undoFlag	1110
71 Index	1113

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we've broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We've also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Chapter 1

Credits

Axiom has a very long history and many people have contributed to the effort, some in large ways and some in small ways. Any and all effort deserves recognition. There is no other criteria than contribution of effort. We would like to acknowledge and thank the following people:

1.0.1 defvar \$credits

— initvars —

```
(defvar credits '(
  "An alphabetical listing of contributors to AXIOM:"
  "Cyril Alberga      Roy Adler      Christian Aistleitner"
  "Richard Anderson  George Andrews  S.J. Atkins"
  "Henry Baker       Martin Baker    Stephen Balzac"
  "Yuriy Baransky    David R. Barton Gerald Baumgartner"
  "Gilbert Baumslag  Michael Becker  Nelson H. F. Beebe"
  "Jay Belanger      David Bindel    Fred Blair"
  "Vladimir Bondarenko Mark Botch      Alexandre Bouyer"
  "Peter A. Broadbery Martin Brock     Manuel Bronstein"
  "Stephen Buchwald  Florian Bundschuh Luanne Burns"
  "William Burge"
  "Quentin Carpent   Robert Caviness Bruce Char"
  "Ondrej Certik     Cheekai Chin    David V. Chudnovsky"
  "Gregory V. Chudnovsky Josh Cohen       Christophe Conil"
  "Don Coppersmith   George Corliss  Robert Corless"
  "Gary Cornell      Meino Cramer    Claire Di Crescenzo"
  "David Cyganski"
  "Nathaniel Daly    Timothy Daly Sr. Timothy Daly Jr."
  "James H. Davenport Didier Deshommes Michael Dewar"
  "Jean Della Dora   Gabriel Dos Reis Claire DiCrescendo"
```

"Sam Dooley	Lionel Ducos	Lee Duhem"
"Martin Dunstan	Brian Dupee	Dominique Duval"
"Robert Edwards	Heow Eide-Goodman	Lars Erickson"
"Richard Fateman	Bertfried Fauser	Stuart Feldman"
"John Fletcher	Brian Ford	Albrecht Fortenbacher"
"George Frances	Constantine Frangos	Timothy Freeman"
"Korrinn Fu"		
"Marc Gaetano	Rudiger Gebauer	Kathy Gerber"
"Patricia Gianni	Samantha Goldrich	Holger Gollan"
"Teresa Gomez-Diaz	Laureano Gonzalez-Vega	Stephen Gortler"
"Johannes Grabmeier	Matt Grayson	Klaus Ebbe Grue"
"James Griesmer	Vladimir Grinberg	Oswald Gschnitzer"
"Jocelyn Guidry"		
"Gaetan Hache	Steve Hague	Satoshi Hamaguchi"
"Mike Hansen	Richard Harke	Bill Hart"
"Vilya Harvey	Martin Hassner	Arthur S. Hathaway"
"Dan Hatton	Waldek Hebisch	Karl Hegbloom"
"Ralf Hemmecke	Henderson	Antoine Hersen"
"Gernot Hueber"		
"Pietro Iglio"		
"Alejandro Jakubi	Richard Jenks"	
"Kai Kaminski	Grant Keady	Tony Kennedy"
"Ted Kosan	Paul Kosinski	Klaus Kusche"
"Bernhard Kutzler"		
"Tim Lahey	Larry Lambe	Kaj Laurson"
"Franz Lehner	Frederic Lehubey	Michel Levaud"
"Howard Levy	Liu Xiaojun	Rudiger Loos"
"Michael Lucks	Richard Luczak"	
"Camm Maguire	Francois Maltey	Alasdair McAndrew"
"Bob McElrath	Michael McGettrick	Ian Meikle"
"David Mentre	Victor S. Miller	Gerard Milmeister"
"Mohammed Mobarak	H. Michael Moeller	Michael Monagan"
"Marc Moreno-Maza	Scott Morrison	Joel Moses"
"Mark Murray"		
"William Naylor	Patrice Naudin	C. Andrew Neff"
"John Nelder	Godfrey Nolan	Arthur Norman"
"Jinzhong Niu"		
"Michael O'Connor	Summat Oemrawsingh	Kostas Oikonomou"
"Humberto Ortiz-Zuazaga"		
"Julian A. Padget	Bill Page	David Parnas"
"Susan Pelzel	Michel Petitot	Didier Pinchon"
"Ayal Pinkus	Jose Alfredo Portes"	
"Claude Quitte"		
"Arthur C. Ralfs	Norman Ramsey	Anatoly Raportirenko"
"Albert D. Rich	Michael Richardson	Renaud Rioboo"
"Jean Rivlin	Nicolas Robidoux	Simon Robinson"
"Raymond Rogers	Michael Rothstein	Martin Rubey"
"Philip Santas	Alfred Scheerhorn	William Schelter"
"Gerhard Schneider	Martin Schoenert	Marshall Schor"
"Frithjof Schulze	Fritz Schwarz	Steven Segletes"

"Nick Simicich	William Sit	Elena Smirnova"
"Jonathan Steinbach	Fabio Stumbo	Christine Sundaresan"
"Robert Sutor	Moss E. Sweedler	Eugene Surowitz"
"Max Tegmark	James Thatcher	Balbir Thomas"
"Mike Thomas	Dylan Thurston	Steve Toleque"
"Barry Trager	Themos T. Tsikas"	
"Gregory Vanuxem"		
"Bernhard Wall	Stephen Watt	Jaap Weel"
"Juergen Weiss	M. Weller	Mark Wegman"
"James Wen	Thorsten Werther	Michael Wester"
"John M. Wiley	Berhard Will	Clifton J. Williamson"
"Stephen Wilson	Shmuel Winograd	Robert Wisbauer"
"Sandra Wityak	Waldemar Wiwianka	Knut Wolf"
"Clifford Yapp	David Yun"	
"Vadim Zhytnikov	Richard Zippel	Evelyn Zoernack"
"Bruno Zuercher	Dan Zwillinge"	
))		

Chapter 2

The Interpreter

The Axiom interpreter is a large common lisp program. It has several forms of interaction and run from terminal in a standalone fashion, run under the control of a session handler program, run as a web server, or run in a unix pipe.

Chapter 3

The Fundamental Data Structures

Axiom currently depends on a lot of global variables. These are generally listed here along with explanations.

3.1 The global variables

3.1.1 `defvar $current-directory`

The `$current-directory` variable is set to the current directory at startup. This is used by the `)cd` function and some of the compile routines. This is the result of the (p36) `get-current-directory` function. This variable is used to set `*default-pathname-defaults*`. The (p40) `reroot` function resets it to `$spadroot`.

An example of a runtime value is:

```
$current-directory = "/research/test/"
```

3.1.2 `defvar $current-directory`

— initvars —

```
(defvar $current-directory nil)
```

—————

3.1.3 defvar \$defaultMsgDatabaseName

The `$defaultMsgDatabaseName` variable contains the location of the international message database. This can be changed to use a translated version of the messages. It defaults to the United States English version. The relative pathname used as the default is hardcoded in the (p40) reroot function. This value is prefixed with the `$spadroot` to make the path absolute.

In general, all Axiom message text should be stored in this file to enable internationalization of messages.

An example of a runtime value is:

```
|$defaultMsgDatabaseName| =
  #p"/research/test/mnt/ubuntu/doc/messages/s2-us.messages"
```

3.1.4 defvar \$defaultMsgDatabaseName

— initvars —

```
(defvar |$defaultMsgDatabaseName| nil)
```

—————

3.1.5 defvar \$directory-list

The `$directory-list` is a runtime list of absolute pathnames. This list is generated by the (p40) reroot function from the list of relative paths held in the variable `$relative-directory-list`. Each entry will be prefixed by `$spadroot`.

An example of a runtime value is:

```
$directory-list =
  ("/research/test/mnt/ubuntu/../../src/input/"
   "/research/test/mnt/ubuntu/doc/messages/"
   "/research/test/mnt/ubuntu/../../src/algebra/"
   "/research/test/mnt/ubuntu/../../src/interp/"
   "/research/test/mnt/ubuntu/doc/spadhelp/")
```

3.1.6 defvar \$directory-list

— initvars —

```
(defvar $directory-list nil)
```

3.1.7 defvar \$InitialModemapFrame

The `$InitialModemapFrame` is used as the initial value.

See the function “makeInitialModemapFrame” (5.3.14 p 36).

An example of a runtime value is:

```
$InitialModemapFrame = '((nil))
```

3.1.8 defvar \$InitialModemapFrame

— initvars —

```
(defvar |$InitialModemapFrame| '((nil)))
```

3.1.9 defvar \$library-directory-list

The `$library-directory-list` variable is the system-wide search path for library files. It is set up in the (p40) `reroot` function by prepending the `$spadroot` variable to the `$relative-library-directory-list` variable.

An example of a runtime value is:

```
$library-directory-list = ("/research/test/mnt/ubuntu/algebra/")
```

3.1.10 defvar \$library-directory-list

— initvars —

```
(defvar $library-directory-list '("/algebra/"))
```

3.1.11 defvar \$msgDatabaseName

The `$msgDatabaseName` is a locally shared variable among the message database routines.

An example of a runtime value is:

```
|$msgDatabaseName| = nil
```

3.1.12 defvar \$msgDatabaseName

— initvars —

```
(defvar |$msgDatabaseName| nil)
```

3.1.13 defvar \$openServerIfTrue

The `$openServerIfTrue` It appears to control whether the interpreter will be used as an open server, probably for OpenMath use.

If an open server is not requested then this variable to NIL

See the function “openserver” (60.0.23 p 943).

An example of a runtime value is:

```
$openServerIfTrue = nil
```

3.1.14 defvar \$openServerIfTrue

— initvars —

```
(defvar $openServerIfTrue nil)
```

3.1.15 defvar \$relative-directory-list

The `$relative-directory-list` variable contains a hand-generated list of directories used in the Axiom system. The relative directory list specifies a search path for files for the current directory structure. It has been changed from the NAG distribution back to the original form.

This list is used by the (p40) reroot function to generate the absolute list of paths held in the variable `$directory-list`. Each entry will be prefixed by `$spadroot`.

An example of a runtime value is:

```
$relative-directory-list =
  ("../../../../src/input/"
   "/doc/messages/"
   "../../../../src/algebra/"
   "../../../../src/interp/"
   "/doc/spadhelp/")
```

3.1.16 defvar \$relative-directory-list

— initvars —

```
(defvar $relative-directory-list
  '("../../../../src/input/"
    "/doc/messages/"
    "../../../../src/algebra/"
    "../../../../src/interp/" ; for lisp files (helps fd)
    "/doc/spadhelp/" ))
```

3.1.17 defvar \$relative-library-directory-list

The `$relative-library-directory-list` is a hand-generated list of directories containing algebra. The (p40) `reroot` function will prefix every path in this list with the value of the `$spadroot` variable to construct the `$library-directory-list` variable.

An example of a runtime value is:

```
$relative-library-directory-list = ("/algebra/")
```

3.1.18 defvar \$relative-library-directory-list

— initvars —

```
(defvar $relative-library-directory-list '("/algebra/"))
```

3.1.19 defvar \$spadroot

The `$spadroot` variable is the internal name for the AXIOM shell variable. It is set in `reroot` to the value of the argument. The value is expected to be a directory name. The (p35)

initroot function uses this variable if the AXIOM shell variable is not set. The (p36) make-absolute-filename function uses this path as a prefix to all of the relative filenames to make them absolute.

An example of a runtime value is:

```
$Spadroot = "/research/test/mnt/ubuntu"
```

3.1.20 defvar \$Spadroot

— initvars —

```
(defvar $Spadroot nil)
```

—————

3.1.21 defvar \$SpadServer

The \$SpadServer determines whether Axiom acts as a remote server.

See the function “openserver” (60.0.23 p 943).

An example of a runtime value is:

```
$SpadServer = nil
```

3.1.22 defvar \$SpadServer

— initvars —

```
(defvar $SpadServer nil "t means Axiom acts as a remote server")
```

—————

3.1.23 defvar \$SpadServerName

The \$SpadServerName defines the name of the spad server socket. In unix these exist in the tmp directory as names.

See the function “openserver” (60.0.23 p 943).

An example of a runtime value is:

```
$SpadServerName = "/tmp/.d"
```

3.1.24 defvar \$SpadServerName

— initvars —

```
(defvar $SpadServerName "/tmp/.d" "the name of the spad server socket")
```

—————

Chapter 4

Starting Axiom

Axiom starts by invoking a function value of the lisp symbol `*top-level-hook*`. The function invocation path to from this point until the prompt is approximates (skipping initializations):

```
lisp -> restart
      -> |spad|
      -> |runspad|
      -> |ncTopLevel|
      -> |ncIntLoop|
      -> |intloop|
      -> |SpadInterpretStream|
      -> |intloopReadConsole|
```

The `—intloopReadConsole—` function does tail-recursive calls to itself (don't break this) and never exits.

4.1 Variables Used

4.2 Data Structures

4.3 Functions

4.3.1 Set the restart hook

When a lisp image containing code is reloaded there is a hook to allow a function to be called. In our case it is the restart function which is the entry to the Axiom interpreter.

— **defun set-restart-hook 0** —

```
(defun set-restart-hook ())
```

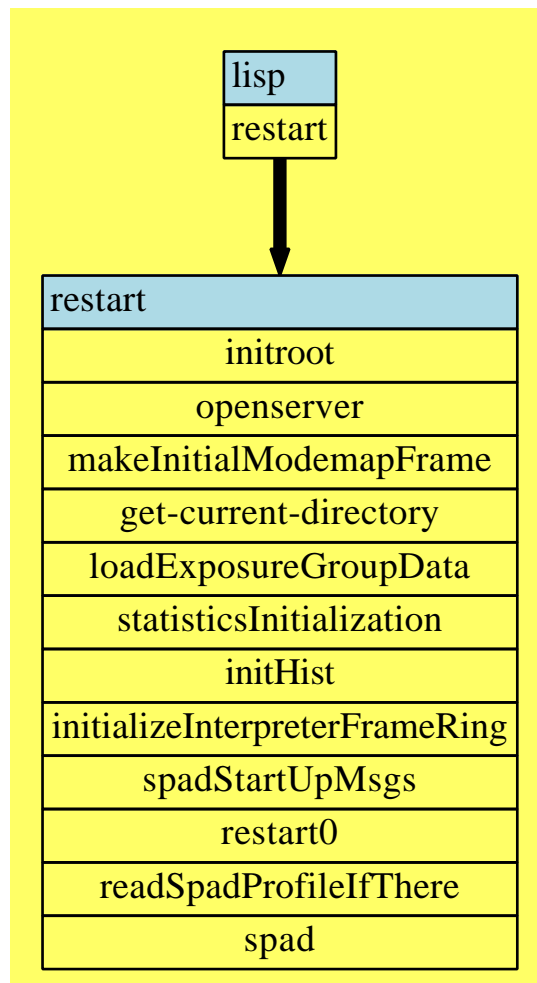


```

"Set the restart hook"
#+KCL (setq system::*top-level-hook* 'restart)
#+Lucid (setq boot::restart-hook 'restart)
'restart
)

```

4.3.2 restart function (The restart function)



The restart function is the real root of the world. It sets up memory if we are working in a GCL/akcl version of the system.

The `compiler::*compile-verbose*` flag has been set to nil globally. We do not want to know

about the microsteps of GCL's compile facility.

The `compiler::*suppress-compiler-warnings*` flag has been set to `t`. We do not care that certain generated variables are not used.

The `compiler::*suppress-compiler-notes*` flag has been set to `t`. We do not care that tail recursion occurs.

It sets the current package to be the "BOOT" package which is the standard package in which the interpreter runs.

The "initroot" (5.3.9 p 35) function sets global variables that depend on the AXIOM shell variable. These are needed to find basic files like `s2-us.msgs`, which contains the error message text.

The "openserver" (60.0.23 p 943) function tried to set up the socket connection used for things like hyperdoc. The `$openServerIfTrue` variable starts true, which implies trying to start a server.

The `$I0index` variable is the number associated with the input prompt. Every successful expression evaluated increments this number until a `)clear all` resets it. Here we set it to the initial value.

Axiom has multiple frames that contain independent information about a computation. There can be several frames at any one time and you can shift back and forth between the frames. By default, the system starts in "frame0" (try the `)frame names` command). See the Frame Mechanism chapter (32.3.1 page 530).

The `$InteractiveFrame` variable contains the state information related to the current frame, which includes things like the last value, the value of all of the variables, etc.

The "printLoadMsgs" (44.30.4 p 713) variable controls whether load messages will be output as library routines are loaded. We disable this by default. It can be changed by using `)set message autoload`.

The "current-directory" (3.1.2 p 7) variable is set to the current directory. This is used by the `)cd` function and some of the compile routines.

The "statisticsInitialization" (?? p ??) function initializes variables used to collect statistics. Currently, only the garbage collector information is initialized.

```
[init-memory-config p34]
[initroot p35]
[openserver p943]
[makeInitialModemapFrame p36]
[get-current-directory p36]
[statisticsInitialization p??]
[initHist p559]
[initializeInterpreterFrameRing p533]
[spadStartUpMsgs p19]
[restart0 p18]
[readSpadProfileIfThere p921]
[spad p20]
```

```

[$openServerIfTrue p10]
[$SpadServername p??]
[$SpadServer p12]
[$IOindex p??]
[$InteractiveFrame p??]
[$printLoadMsgs p713]
[$current-directory p7]
[$displayStartMsgs p726]
[$currentLine p??]

```

— **defun restart** —

```

(defun restart ()
  (declare (special $openServerIfTrue $SpadServerName |$SpadServer|
    |$IOindex| |$InteractiveFrame| |$printLoadMsgs| $current-directory
    |$displayStartMsgs| |$currentLine|))
  #+:akcl
    (init-memory-config :cons 500 :fixnum 200 :symbol 500 :package 8
      :array 400 :string 500 :cfun 100 :cpages 3000 :rpages 1000 :hole 2000)
  #+:akcl (setq compiler::*compile-verbose* nil)
  #+:akcl (setq compiler::*suppress-compiler-warnings* t)
  #+:akcl (setq compiler::*suppress-compiler-notes* t)
    (in-package "BOOT")
    (initroot)
  #+:akcl
    (when (and $openServerIfTrue (zerop (openserver $SpadServerName)))
      (setq $openServerIfTrue nil)
      (setq |$SpadServer| t))
    (setq |$IOindex| 1)
    (setq |$InteractiveFrame| (|makeInitialModemapFrame|))
    (setq |$printLoadMsgs| nil)
    (setq $current-directory (get-current-directory))
    (setq *default-pathname-defaults* (pathname $current-directory))
    (|statisticsInitialization|)
    (|initHist|)
    (|initializeInterpreterFrameRing|)
    (when |$displayStartMsgs| (|spadStartUpMsgs|))
    (setq |$currentLine| nil)
    (restart0)
    (|readSpadProfileIfThere|)
    (|spad|))

```

4.3.3 defun Non-interactive restarts

```

[compressopen p??]
[interpopen p??]

```

```
[operationopen p??]
[categoryopen p??]
[browseopen p??]
[getEnv p??]
```

— **defun restart0** —

```
(defun restart0 ()
  (compressopen)      ;; set up the compression tables
  (interpopen)        ;; open up the interpreter database
  (operationopen)      ;; all of the operations known to the system
  (categoryopen)      ;; answer hasCategory question
  (browseopen))
```

4.3.4 defun The startup banner messages

```
[fillerSpaces p20]
[specialChar p936]
[sayKeyedMsg p331]
[sayMSG p333]
[msgAlist p328]
[$opSysName p??]
[$linelength p751]
[*yearweek* p??]
[*build-version* p??]
```

— **defun spadStartUpMsgs** —

```
(defun |spadStartUpMsgs| ()
  (let (bar)
    (declare (special |msgAlist| |$opSysName| $linelength *yearweek*
                      *build-version*))
    (when (> $linelength 60)
      (setq bar (|fillerSpaces| $linelength (|specialChar| '|hbar|)))
      (|sayKeyedMsg| 'S2GL0001 (list *build-version* *yearweek*))
      (|sayMSG| bar)
      (|sayKeyedMsg| 'S2GL0018C nil)
      (|sayKeyedMsg| 'S2GL0018D nil)
      (|sayKeyedMsg| 'S2GL0003B (list |$opSysName|))
      (say " Visit http://axiom-developer.org for more information")
      (|sayMSG| bar)
      (setq |msgAlist| nil)
      (|sayMSG| '| |))))
```

4.3.5 defun Make a vector of filler characters

```
[ifcar p??]
```

— defun fillerSpaces —

```
(defun |fillerSpaces| (&rest arglist &aux charPart n)
  (dsetq (n . charPart) arglist)
  (if (<= n 0)
      ""
      (make-string n :initial-element (character (or (ifcar charPart) " "))))
```

—————

4.3.6 Starts the interpreter but do not read in profiles

```
[setOutputAlgebra p740]
[runspad p20]
[$PrintCompilerMessageIfTrue p??]
```

— defun spad —

```
(defun |spad| ()
  "Starts the interpreter but do not read in profiles"
  (let (|$PrintCompilerMessageIfTrue|)
    (declare (special |$PrintCompilerMessageIfTrue|))
    (setq |$PrintCompilerMessageIfTrue| nil)
    (|setOutputAlgebra| '|%initialize%|)
    (|runspad|)
    '|EndOfSpad|))
```

—————

4.3.7 defvar \$quitTag

— initvars —

```
(defvar |$quitTag| system::*quit-tag*)
```

—————

4.3.8 defun runspad

```
[quitTag p20]
[coerceFailure p??]
```

```
[top-level p??]
[seq p??]
[exit p??]
[resetStackLimits p21]
[ncTopLevel p25]
[$quitTag p20]
```

— **defun runspad** —

```
(defun |runspad| ()
  (prog (mode)
    (declare (special |$quitTag|))
    (return
      (seq
        (progn
          (setq mode '|restart|)
          (do ()
            ((null (eq mode '|restart|)) nil)
            (seq
              (exit
                (progn
                  (|resetStackLimits|)
                  (catch |$quitTag|
                    (catch '|coerceFailure|
                      (setq mode (catch '|top_level| (|ncTopLevel|))))))))))))))
```

—————

4.3.9 defun Reset the stack limits

```
[reset-stack-limits p??]
```

— **defun resetStackLimits 0** —

```
(defun |resetStackLimits| ()
  "Reset the stack limits"
  (system:reset-stack-limits))
```

—————

Chapter 5

Handling Terminal Input

5.1 Streams

5.1.1 defvar \$curinstream

The curinstream variable is set to the value of the ***standard-input*** common lisp variable in ncIntLoop. While not using the “dollar” convention this variable is still “global”.

— **initvars** —

```
(defvar curinstream (make-synonym-stream '*standard-input*))
```

—————

5.1.2 defvar \$curoutstream

The curoutstream variable is set to the value of the ***standard-output*** common lisp variable in ncIntLoop. While not using the “dollar” convention this variable is still “global”.

— **initvars** —

```
(defvar curoutstream (make-synonym-stream '*standard-output*))
```

—————

5.1.3 defvar \$errorinstream

— **initvars** —


```
(defvar errorinstream (make-synonym-stream '*terminal-io*))
```

5.1.4 defvar \$erroroutstream

— initvars —

```
(defvar erroroutstream (make-synonym-stream '*terminal-io*))
```

5.1.5 defvar \$*eof*

— initvars —

```
(defvar *eof* nil)
```

5.1.6 defvar \$*whitespace*

— initvars —

```
(defvar *whitespace*
'(#\Space #\Newline #\Tab #\Page #\Linefeed #\Return #\Backspace)
"A list of characters used by string-trim considered as whitespace")
```

5.1.7 defvar \$InteractiveMode

— initvars —

```
(defvar |$InteractiveMode| t)
```

5.1.8 defvar \$boot

— initvars —

```
(defvar $boot nil)
```

—————

5.1.9 Top-level read-parse-eval-print loop

Top-level read-parse-eval-print loop for the interpreter. Uses the Bill Burge's parser. [ncInt-Loop p25]

```
[ $e p?? ]
[ $spad p20 ]
[ $newspad p?? ]
[ $boot p25 ]
[ $InteractiveMode p24 ]
[ $InteractiveFrame p?? ]
[ *eof* p24 ]
[ in-stream p922 ]
```

— defun ncTopLevel —

```
(defun |ncTopLevel| ()
  "Top-level read-parse-eval-print loop"
  (let (|$e| $spad $newspad $boot |$InteractiveMode| *eof* in-stream)
    (declare (special |$e| $spad $newspad $boot |$InteractiveMode| *eof*
                      in-stream |$InteractiveFrame|))
    (setq in-stream curinstream)
    (setq *eof* nil)
    (setq |$InteractiveMode| t)
    (setq $boot nil)
    (setq $newspad t)
    (setq $spad t)
    (setq |$e| |$InteractiveFrame|)
    (|ncIntLoop|)))
```

—————

5.1.10 defun ncIntLoop

```
[intloop p26]
[curinstream p23]
[curoutstream p23]
```

— **defun ncIntLoop** —

```
(defun |ncIntLoop| ()
  (let ((curinstream *standard-output*)
        (curoutstream *standard-input*))
    (declare (special curinstream curoutstream))
    (|intloop|)))
```

—————

5.1.11 **defvar \$intTopLevel**

— **initvars** —

```
(defvar |$intTopLevel| '|top_level|)
```

—————

5.1.12 **defvar \$intRestart**

— **initvars** —

```
(defvar |$intRestart| '|restart|)
```

—————

5.1.13 **defun intloop**

Note that the `SpadInterpretStream` function uses a list of three strings as an argument. The values in the list seem to have no use and can eventually be removed. [`intTopLevel` p26]

[`SpadInterpretStream` p27]

[`resetStackLimits` p21]

[`$intTopLevel` p26]

[`$intRestart` p26]

— **defun intloop** —

```
(defun |intloop| ()
  (prog (mode)
    (declare (special |$intTopLevel| |$intRestart|)))
```

```

(return
  (progn
    (setq mode |$intRestart|)
    ((lambda ()
      (loop
        (cond
          ((not (equal mode |$intRestart|))
           (return nil))
          (t
           (progn
             (|resetStackLimits|)
             (setq mode
              (catch |$intTopLevel|
                (|SpadInterpretStream| 1
                 (list 'tim 'daly '? t))))))))))))))

```

5.1.14 defvar \$ncMsgList

— initvars —

```
(defvar |$ncMsgList| nil)
```

5.1.15 defun SpadInterpretStream

The SpadInterpretStream function takes three arguments

str This is passed as an argument to intloopReadConsole

source This is the name of a source file but appears not to be used. It is set to the list (tim daly ?).

interactive? If this is false then various messages are suppressed and input does not use piles. If this is true then the library loading routines might output messages and piles are expected on input (as from a file).

The system commands are handled by the function kept in the “hook” variable `$systemCommandFunction` which has the default function `InterpExecuteSpadSystemCommand`. Thus, when a system command is entered this function is called.

The `$promptMsg` variable is set to the constant S2CTP023. This constant points to a message in `src/doc/msgsg/s2-us.msgsg`. This message does nothing but print the argument value.

5.1.16 defvar \$promptMsg

— initvars —

```
(defvar |$promptMsg| 'S2CTP023)
```

—————

5.1.17 defvar \$newcompErrorCount

— initvars —

```
(defvar |$newcompErrorCount| 0)
```

—————

5.1.18 defvar \$npos

— initvars —

```
(defvar |$npos| (list '|noposition|))
```

—————

```
[mkprompt p42]
[intloopReadConsole p29]
[intloopInclude p63]
[$promptMsg p28]
[$systemCommandFunction p??]
[$ncMsgList p27]
[$erMsgToss p??]
[$lastPos p??]
[$inclAssertions p??]
[$okToExecuteMachineCode p??]
[$newcompErrorCount p28]
[$libQuiet p??]
[$fn p??]
[$npos p28]
```

— defun SpadInterpretStream —

```

(defun |SpadInterpretStream| (str source interactive?)
  (let (|$promptMsg| |$systemCommandFunction|
        |$ncMsgList| |$erMsgToss| |$lastPos| |$inclAssertions|
        |$okToExecuteMachineCode| |$newcompErrorCount|
        |$libQuiet| |$fn|)
    (declare (special |$promptMsg|
                      |$systemCommandFunction| |$ncMsgList| |$erMsgToss| |$lastPos|
                      |$inclAssertions| |$okToExecuteMachineCode| |$newcompErrorCount|
                      |$libQuiet| |$fn| |$npos|))
    (setq |$fn| source)
    (setq |$libQuiet| (null interactive?))
    (setq |$newcompErrorCount| 0)
    (setq |$okToExecuteMachineCode| t)
    (setq |$inclAssertions| (list 'aix '|CommonLisp|))
    (setq |$lastPos| |$npos|)
    (setq |$erMsgToss| nil)
    (setq |$ncMsgList| nil)
    (setq |$systemCommandFunction| #'|InterpExecuteSpadSystemCommand|)
    (setq |$promptMsg| 's2ctp023)
    (if interactive?
        (progn
          (princ (mkprompt))
          (|intloopReadConsole| "" str))
        (|intloopInclude| source 0))))

```

5.2 The Read-Eval-Print Loop

5.2.1 defun intloopReadConsole

Note that this function relies on the fact that lisp can do tail-recursion. The function recursively invokes itself.

The `serverReadLine` function is a special readline function that handles communication with the session manager code, which is a separate process running in parallel.

We read a line from standard input.

- If it is a null line then we exit Axiom.
- If it is a zero length line we prompt and recurse
- If `$dalymode` and open-paren we execute lisp code, prompt and recurse The `$dalymode` will interpret any input that begins with an open-paren as a lisp expression rather than Axiom input. This is useful for debugging purposes when most of the input lines will be lisp. Setting `$dalymode` non-nil will certainly break user expectations and is to be used with caution.

- If it is “)fi” or “)fin” we drop into lisp. Use the (restart) function to return to the interpreter loop.
- If it starts with “)” we process the command, prompt, and recurse
- If it is a command then we remember the current line, process the command, prompt, and recurse.
- If the input has a trailing underscore (Axiom line-continuation) then we cut off the continuation character and pass the truncated string to ourselves, prompt, and recurse
- otherwise we process the input, prompt, and recurse.

Notice that all but two paths (a null input or a “)fi” or a “)fin”) will end up as a recursive call to ourselves. [top-level p??]

```
[serverReadLine p44]
[leaveScratchpad p617]
[mkprompt p42]
[intloopReadConsole p29]
[intloopPrefix? p35]
[intnplisp p36]
[setCurrentLine p41]
[ncloopCommand p458]
[concat p1003]
[ncloopEscaped p37]
[intloopProcessString p37]
[$dalymode p637]
```

— defun intloopReadConsole —

```
(defun |intloopReadConsole| (b n)
  (declare (special $dalymode))
  (let (c d pfx input)
    (setq input (|serverReadLine| *standard-input*))
    (when (null (stringp input)) (|leaveScratchpad|))
    (when (eql (length input) 0)
      (princ (mkprompt))
      (|intloopReadConsole| "" n))
    (when (and $dalymode (|intloopPrefix?| "(" input))
      (|intnplisp| input)
      (princ (mkprompt))
      (|intloopReadConsole| "" n))
    (setq pfx (|intloopPrefix?| ")fi" input))
    (when (and pfx (or (string= pfx ")fi") (string= pfx ")fin")))
      (throw '|top_level| nil))
    (when (and (equal b "") (setq d (|intloopPrefix?| ")" input)))
      (|setCurrentLine| d)
      (setq c (|ncloopCommand| d n))
      (princ (mkprompt)))
```

```
(|intloopReadConsole| "" c))
(setq input (concat b input))
(when (|ncloopEscaped| input)
  (|intloopReadConsole| (subseq input 0 (- (length input) 1)) n))
(setq c (|intloopProcessString| input n))
(princ (mkprompt))
(|intloopReadConsole| "" c)))
```

5.3 Helper Functions

5.3.1 Get the value of an environment variable

[getenv p??]

— defun getenviron 0 —

```
(defun getenviron (var)
  "Get the value of an environment variable"
  #+allegro (sys::getenv (string var))
  #+clisp (ext:getenv (string var))
  #+(or cmu scl)
  (cdr
   (assoc (string var) ext:*environment-list* :test #'equalp :key #'string))
  #+(or kcl akcl gcl) (si::getenv (string var))
  #+lispworks (lw:environment-variable (string var))
  #+lucid (lcl:environment-variable (string var))
  #+mcl (ccl::getenv var)
  #+sbcl (sb-ext:posix-getenv var)
  )
```

5.3.2 defvar \$intCoerceFailure

— initvars —

```
(defvar |$intCoerceFailure| ' |coerceFailure|)
```

5.3.3 defvar \$intSpadReader

```

— initvars —

(defvar |$intSpadReader| 'SPAD_READER)

```

5.3.4 defun InterpExecuteSpadSystemCommand

```

[intCoerceFailure p31]
[intSpadReader p32]
[ExecuteInterpSystemCommand p32]
[$intSpadReader p32]
[$intCoerceFailure p31]

— defun InterpExecuteSpadSystemCommand —

(defun |InterpExecuteSpadSystemCommand| (string)
  (declare (special |$intSpadReader| |$intCoerceFailure|))
  (catch |$intCoerceFailure|
    (catch |$intSpadReader|
      (|ExecuteInterpSystemCommand| string))))

```

5.3.5 defun ExecuteInterpSystemCommand

```

[intProcessSynonyms p33]
[substring p??]
[doSystemCommand p426]
[$currentLine p??]

— defun ExecuteInterpSystemCommand —

(defun |ExecuteInterpSystemCommand| (string)
  (let (|$currentLine|)
    (declare (special |$currentLine|))
    (setq string (|intProcessSynonyms| string))
    (setq |$currentLine| string)
    (setq string (substring string 1 nil))
    (unless (equal string "") (|doSystemCommand| string))))

```

5.3.6 defun Handle Synonyms

```
[processSynonyms p33]
[line p??]
```

— **defun intProcessSynonyms** —

```
(defun |intProcessSynonyms| (str)
  (let ((line str))
    (declare (special line))
    (|processSynonyms|
     line))
```

—————

5.3.7 defun Synonym File Reader

```
[strpos p1002]
[substring p??]
[string2id-n p??]
[lassoc p??]
[nequal p??]
[strconc p??]
[size p1001]
[concat p1003]
[rplacstr p??]
[processSynonyms p33]
[$CommandSynonymAlist p458]
[line p??]
```

— **defun processSynonyms** —

```
(defun |processSynonyms| ()
  (let (fill p aline synstr syn to opt fun cl chr)
    (declare (special |$CommandSynonymAlist| line))
    (setq p (strpos ")") line 0 nil))
  (setq fill "")
  (cond
   (p
    (setq aline (substring line p nil))
    (when (> p 0) (setq fill (substring line 0 p))))
   (t
    (setq p 0)
    (setq aline line)))
  (setq to (strpos " " aline 1 nil))
  (cond (to (setq to (1- to))))
  (setq synstr (substring aline 1 to))
```

```

(setq syn (string2id-n synstr 1))
(when (setq fun (lassoc syn |$CommandSynonymAlist|))
  (setq to (strpos ")" fun 1 nil))
  (cond
    ((and to (nequal to (1- (size fun))))
      (setq opt (strconc " " (substring fun to nil)))
      (setq fun (substring fun 0 (1- to ))))
    (t (setq opt " ")))
  (when (> (size synstr) (size fun))
    (do ((G167173 (size synstr)) (i (size fun) (1+ i)))
      ((> i G167173) nil)
      (setq fun (concat fun " "))))
  (setq cl (strconc fill (rplacstr aline 1 (size synstr) fun) opt))
  (setq line cl)
  (setq chr (elt line (1+ p)))
  (|processSynonyms|))))

```

5.3.8 defun init-memory-config

Austin-Kyoto Common Lisp (AKCL), now known as Gnu Common Lisp (GCL) requires some changes to the default memory setup to run Axiom efficiently. This function performs those setup commands. [allocate p??]

[allocate-contiguous-pages p??]

[allocate-relocatable-pages p??]

[set-hole-size p??]

— defun init-memory-config 0 —

```

(defun init-memory-config (&key
  (cons 500)
  (fixnum 200)
  (symbol 500)
  (package 8)
  (array 400)
  (string 500)
  (cfun 100)
  (cpages 3000)
  (rpages 1000)
  (hole 2000) )
  ;; initialize AKCL memory allocation parameters
  #+:AKCL
  (progn
    (system:allocate 'cons cons)
    (system:allocate 'fixnum fixnum)
    (system:allocate 'symbol symbol)
    (system:allocate 'package package)

```

```

(system:allocate 'array array)
(system:allocate 'string string)
(system:allocate 'cfun cfun)
(system:allocate-contiguous-pages cpages)
(system:allocate-relocatable-pages rpages)
(system:set-hole-size hole))
#-:AKCL
nil)

```

5.3.9 Set spadroot to be the AXIOM shell variable

Sets up the system to use the **AXIOM** shell variable if we can and default to the **\$spadroot** variable (which was the value of the **AXIOM** shell variable at build time) if we can't.

```

[reroot p40]
[getenv p31]
[$spadroot p12]

```

— defun initroot —

```

(defun initroot (&optional (newroot (getenv "AXIOM")))
  "Set spadroot to be the AXIOM shell variable"
  (declare (special $spadroot))
  (reroot (or newroot $spadroot (error "setenv AXIOM or (setq $spadroot)"))))

```

5.3.10 Does the string start with this prefix?

If the prefix string is the same as the whole string initial characters –R(ignoring spaces in the whole string) then we return the whole string minus any leading spaces.

— defun intloopPrefix? 0 —

```

(defun |intloopPrefix?| (prefix whole)
  "Does the string start with this prefix?"
  (let ((newprefix (string-left-trim '(#\space) prefix))
        (newwhole (string-left-trim '(#\space) whole)))
    (when (<= (length newprefix) (length newwhole))
      (when (string= newprefix newwhole :end2 (length prefix))
        newwhole))))

```

5.3.11 defun Interpret a line of lisp code

This is used to handle)lisp top level commands [nplisp p452]
[\$currentLine p??]

— defun intnplisp —

```
(defun |intnplisp| (s)
  (declare (special |$currentLine|))
  (setq |$currentLine| s)
  (|nplisp| |$currentLine|))
```

5.3.12 Get the current directory

— defun get-current-directory 0 —

```
(defun get-current-directory ()
  "Get the current directory"
  (namestring (truename "")))
```

5.3.13 Prepend the absolute path to a filename

Prefix a filename with the **AXIOM** shell variable. [\$spadroot p12]

— defun make-absolute-filename 0 —

```
(defun make-absolute-filename (name)
  "Prepend the absolute path to a filename"
  (declare (special $spadroot))
  (concatenate 'string $spadroot name))
```

5.3.14 Make the initial modemap frame

[copy p??]
[\$InitialModemapFrame p9]

— defun makeInitialModemapFrame 0 —

```
(defun |makeInitialModemapFrame| ()
  "Make the initial modemap frame"
  (declare (special |$InitialModemapFrame|))
  (copy |$InitialModemapFrame|))
```

5.3.15 defun ncloopEscaped

The ncloopEscaped function will return true if the last non-blank character of a line is an underscore, the Axiom line-continuation character. Otherwise, it returns nil.

— **defun ncloopEscaped 0** —

```
(defun |ncloopEscaped| (x)
  (let ((l (length x)))
    (dotimes (i l)
      (when (char= (char x (- l i 1)) #\_) (return t))
      (unless (char= (char x (- l i 1)) #\space) (return nil))))))
```

5.3.16 defun intloopProcessString

```
[setCurrentLine p41]
[intloopProcess p64]
[next p38]
[incString p39]
```

— **defun intloopProcessString** —

```
(defun |intloopProcessString| (s n)
  (|setCurrentLine| s)
  (|intloopProcess| n t
    (|next| #'|ncloopParse|
      (|next| #'|lineoftoks| (|incString| s)))))
```

5.3.17 defun ncloopParse

```
[ncloopDQlines p72]
[npParse p143]
```

[dqToList p346]

— **defun ncloopParse** —

```
(defun |ncloopParse| (s)
  (let (cudr lines stream dq t1)
    (setq t1 (car s))
    (setq dq (car t1))
    (setq stream (cadr t1))
    (setq t1 (|ncloopDQlines| dq stream))
    (setq lines (car t1))
    (setq cudr (cadr t1))
    (cons (list (list lines (|npParse| (|dqToList| dq)))) (cdr s))))
```

—————

5.3.18 **defun next**

[Delay p105]

[next1 p38]

— **defun next** —

```
(defun |next| (f s)
  (|Delay| #'|next1| (list f s)))
```

—————

5.3.19 **defun next1**

[StreamNull p335]

[incAppend p88]

[next p38]

— **defun next1** —

```
(defun |next1| (&rest z)
  (let (h s f)
    (setq f (car z))
    (setq s (cadr z))
    (cond
      ((|StreamNull| s) |StreamNil|)
      (t
       (setq h (apply f (list s)))
       (|incAppend| (car h) (|next| f (cdr h)))))))
```

5.3.20 defun incString

```
[incRenum p75]
[incLude p78]
[Top p78]
```

— defun incString —

```
(defun |incString| (s)
  (declare (special |Top|))
  (incRenum| (incLude| 0 (list s) 0 (list "strings") (list |Top|))))
```

5.3.21 Call the garbage collector

Call the garbage collector on various platforms.

— defun reclaim 0 —

```
#+abcl
(defun reclaim () "Call the garbage collector" (ext::gc))
#+allegro
(defun reclaim () "Call the garbage collector" (excl::gc t))
#+CCL
(defun reclaim () "Call the garbage collector" (gc))
#+clisp
(defun reclaim ()
  "Call the garbage collector"
  (#+lisp=cl ext::gc #-lisp=cl lisp::gc))
#+(or :cmulisp :cmu)
(defun reclaim () "Call the garbage collector" (ext:gc))
#+cormanlisp
(defun reclaim () "Call the garbage collector" (cl::gc))
#+(OR IBCL KCL GCL)
(defun reclaim () "Call the garbage collector" (si::gbc t))
#+lispworks
(defun reclaim () "Call the garbage collector" (hcl::normal-gc))
#+Lucid
(defun reclaim () "Call the garbage collector" (lcl::gc))
#+sbcl
(defun reclaim () "Call the garbage collector" (sb-ext::gc))
```

5.3.22 defun reroot

The reroot function is used to reset the important variables used by the system. In particular, these variables are sensitive to the **AXIOM** shell variable. That variable is renamed internally to be **\$spadroot**. The **reroot** function will change the system to use a new root directory and will have the same effect as changing the **AXIOM** shell variable and rerunning the system from scratch. Note that we have changed from the NAG distribution back to the original form. If you need the NAG version you can push **:tpd** on the ***features*** variable before compiling this file. A correct call looks like:

```
(in-package "BOOT")
(reroot "/spad/mnt/${SYS}")
```

where the **\${SYS}** variable is the same one set at build time.

For the example call:

```
(REROOT "/research/test/mnt/ubuntu")
```

the variables are set as:

```
$spadroot = "/research/test/mnt/ubuntu"

$relative-directory-list =
  ("../../../../src/input/"
   "/doc/messages/"
   "../../../../src/algebra/"
   "../../../../src/interp/"
   "/doc/spadhelp/")

$directory-list =
  ("/research/test/mnt/ubuntu/../../../../src/input/"
   "/research/test/mnt/ubuntu/doc/messages/"
   "/research/test/mnt/ubuntu/../../../../src/algebra/"
   "/research/test/mnt/ubuntu/../../../../src/interp/"
   "/research/test/mnt/ubuntu/doc/spadhelp/")

$relative-library-directory-list = ("/algebra/")

$library-directory-list = ("/research/test/mnt/ubuntu/algebra/")

|$defaultMsgDatabaseName| = #p"/research/test/mnt/ubuntu/doc/messages/s2-us.messages"

|$msgDatabaseName| = nil

$current-directory = "/research/test/"

[make-absolute-filename p36]
[$spadroot p12]
```

```

[$directory-list p8]
[$relative-directory-list p11]
[$library-directory-list p9]
[$relative-library-directory-list p11]
[$defaultMsgDatabaseName p8]
[$msgDatabaseName p328]
[$current-directory p7]

```

— **defun reroot** —

```

(defun reroot (dir)
  (declare (special $spadroot $directory-list $relative-directory-list
    $library-directory-list $relative-library-directory-list
    |$defaultMsgDatabaseName| |$msgDatabaseName| $current-directory))
  (setq $spadroot dir)
  (setq $directory-list
    (mapcar #'make-absolute-filename $relative-directory-list))
  (setq $library-directory-list
    (mapcar #'make-absolute-filename $relative-library-directory-list))
  (setq |$defaultMsgDatabaseName|
    (pathname (make-absolute-filename "/doc/msgs/s2-us.msgs")))
  (setq |$msgDatabaseName| ())
  (setq $current-directory $spadroot))

```

—————

5.3.23 defun setCurrentLine

Remember the current line. The cases are:

- If there is no \$currentLine set it to the input
- Is the current line a string and the input a string? Make them into a list
- Is \$currentLine not a cons cell? Make it one.
- Is the input a string? Cons it on the end of the list.
- Otherwise stick it on the end of the list

Note I suspect the last two cases do not occur in practice since they result in a dotted pair if the input is not a cons. However, this is what the current code does so I won't change it.

[\$currentLine p??]

— **defun setCurrentLine 0** —

```
(defun |setCurrentLine| (s)
  (declare (special |$currentLine|))
  (cond
    ((null |$currentLine|) (setq |$currentLine| s))
    ((and (stringp |$currentLine|) (stringp s))
     (setq |$currentLine| (list |$currentLine| s)))
    ((not (consp |$currentLine|)) (setq |$currentLine| (cons |$currentLine| s)))
    ((stringp s) (rplacd (last |$currentLine|) (cons s nil)))
    (t (rplacd (last |$currentLine|) s)))
  |$currentLine|)
```

5.3.24 Show the Axiom prompt

```
[concat p1003]
[substring p??]
[currenttime p??]
[$inputPromptType p723]
[$IOindex p??]
[$interpreterFrameName p??]
```

— defun mkprompt —

```
(defun mkprompt ()
  "Show the Axiom prompt"
  (declare (special |$inputPromptType| |$IOindex| |$interpreterFrameName|))
  (case |$inputPromptType|
    (|none| "")
    (|plain| "-> ")
    (|step| (concat "(" (princ-to-string |$IOindex|) ") -> "))
    (|frame|
     (concat (princ-to-string |$interpreterFrameName|) " ("
              (princ-to-string |$IOindex|) ") -> "))
    (t (concat (princ-to-string |$interpreterFrameName|) " ["
                (substring (currenttime) 8 nil) "]" ["
                (princ-to-string |$IOindex|) "]" -> "))))
```

5.3.25 defvar \$frameAlist

— initvars —

```
(defvar |$frameAlist| nil)
```

5.3.26 `defvar $frameNumber`

— initvars —

```
(defvar |$frameNumber| 0)
```

5.3.27 `defvar $currentFrameNum`

— initvars —

```
(defvar |$currentFrameNum| 0)
```

5.3.28 `defvar $EndServerSession`

— initvars —

```
(defvar |$EndServerSession| nil)
```

5.3.29 `defvar $NeedToSignalSessionManager`

— initvars —

```
(defvar |$NeedToSignalSessionManager| nil)
```

5.3.30 defvar \$sockBufferLength

— initvars —

```
(defvar |$sockBufferLength| 9217)
```

—

5.3.31 READ-LINE in an Axiom server system

```
[coerceFailure p??]
[top-level p??]
[spad-reader p??]
[read-line p??]
[addNewInterpreterFrame p539]
[sockSendInt p??]
[sockSendString p??]
[mkprompt p42]
[sockGetInt p??]
[lassoc p??]
[changeToNamedInterpreterFrame p538]
[sockGetString p??]
[unescapeStringsInForm p63]
[protectedEVAL p46]
[executeQuietCommand p47]
[parseAndInterpret p48]
[serverReadLine is-console (vol9)]
[serverSwitch p??]
[$KillLispSystem p??]
[$NonSmanSession p??]
[$SpadCommand p??]
[$QuietSpadCommand p??]
[$MenuServer p??]
[$sockBufferLength p44]
[$LispCommand p??]
[$EndServerSession p43]
[$EndSession p??]
[$SwitchFrames p??]
[$CreateFrameAnswer p??]
[$currentFrameNum p43]
[$frameNumber p43]
[$frameAlist p42]
[$CreateFrame p??]
[$CallInterp p??]
```

```

[$EndOfOutput p??]
[$SessionManager p??]
[$NeedToSignalSessionManager p43]
[$EndServerSession p43]
[$SpadServer p12]
[*eof* p24]
[in-stream p922]

```

— defun serverReadLine —

```

(defun |serverReadLine| (stream)
  "used in place of READ-LINE in a Axiom server system."
  (let (in-stream *eof* 1 framename currentframe form stringbuf line action)
    (declare (special in-stream *eof* |$SpadServer| |$EndServerSession|
      |$NeedToSignalSessionManager| |$SessionManager| |$EndOfOutput| | |
      |$CallInterp| |$CreateFrame| |$frameAlist| |$frameNumber|
      |$currentFrameNum| |$CreateFrameAnswer| |$SwitchFrames| |$EndSession|
      |$EndServerSession| |$LispCommand| |$sockBufferLength| |$MenuServer|
      |$QuietSpadCommand| |$SpadCommand| |$NonSmanSession| |$KillLispSystem|))
    (force-output)
    (if (or (null |$SpadServer|) (null (is-console stream)))
        (|read-line| stream)
        (progn
          (setq in-stream stream)
          (setq *eof* nil)
          (setq line
            (do ()
              ((null (and (null |$EndServerSession|) (null *eof*))) nil)
              (when |$NeedToSignalSessionManager|
                (|sockSendInt| |$SessionManager| |$EndOfOutput|))
              (setq |$NeedToSignalSessionManager| nil)
              (setq action (|serverSwitch|))
              (cond
                ((= action |$CallInterp|)
                 (setq l (|read-line| stream))
                 (setq |$NeedToSignalSessionManager| t)
                 (return l))
                ((= action |$CreateFrame|)
                 (setq framename (gentemp "frame"))
                 (|addNewInterpreterFrame| framename)
                 (setq |$frameAlist|
                   (cons (cons |$frameNumber| framename) |$frameAlist|))
                 (setq |$currentFrameNum| |$frameNumber|)
                 (|sockSendInt| |$SessionManager| |$CreateFrameAnswer|)
                 (|sockSendInt| |$SessionManager| |$frameNumber|)
                 (setq |$frameNumber| (1+ |$frameNumber|))
                 (|sockSendString| |$SessionManager| (mkprompt)))
                ((= action |$SwitchFrames|)
                 (setq |$currentFrameNum| (|sockGetInt| |$SessionManager|))

```

```

    (setq currentframe (lassoc |$currentFrameNum| |$frameAlist|))
    (|changeToNamedInterpreterFrame| currentframe))
  ((= action |$EndSession|)
   (setq |$EndServerSession| t))
  ((= action |$LispCommand|)
   (setq |$NeedToSignalSessionManager| t)
   (setq stringbuf (make-string |$sockBufferLength|))
   (|sockGetString| |$MenuServer| stringbuf |$sockBufferLength|)
   (setq form (|unescapeStringsInForm| (read-from-string stringbuf)))
   (|protectedEVAL| form))
  ((= action |$QuietSpadCommand|)
   (setq |$NeedToSignalSessionManager| t)
   (|executeQuietCommand|))
  ((= action |$SpadCommand|)
   (setq |$NeedToSignalSessionManager| t)
   (setq stringbuf (make-string 512))
   (|sockGetString| |$MenuServer| stringbuf 512)
   (catch '|coerceFailure|
    (catch '|top_level|
     (catch '|spad_reader|
      (|parseAndInterpret| stringbuf))))))
  (princ (mkprompt))
  (finish-output))
  ((= action |$NonSmanSession|) (setq |$SpadServer| nil))
  ((= action |$KillLispSystem|) (bye))
  (t nil))))
(cond
 (line line)
 (t '||))))))

```

5.3.32 defun protectedEVAL

[resetStackLimits p21]
 [sendHTErrorSignal p??]

— defun protectedEVAL —

```

(defun |protectedEVAL| (x)
  (let (val (error t))
    (unwind-protect
     (progn
      (setq val (eval x))
      (setq error nil))
     (when error
      (|resetStackLimits|))

```

```
(|sendHTErrorSignal|)))
(unless error val)))
```

5.3.33 defvar \$QuietCommand

— initvars —

```
(defvar |$QuietCommand| nil "If true, produce no top level output")
```

5.3.34 defun executeQuietCommand

When \$QuietCommand is true Spad will not produce any output from a top level command

```
[spad-reader p??]
[coerceFailure p??]
[toplevel p??]
[spadreader p??]
[make-string p??]
[sockGetString p??]
[parseAndInterpret p48]
[$MenuServer p??]
[$QuietCommand p47]
```

— defun executeQuietCommand —

```
(defun |executeQuietCommand| ()
  (let (|$QuietCommand| stringBuffer)
    (declare (special |$QuietCommand| |$MenuServer|))
    (setq |$QuietCommand| t)
    (setq stringBuffer (make-string 512))
    (|sockGetString| |$MenuServer| stringBuffer 512)
    (catch '|coerceFailure|
      (catch '|top_level|
        (catch '|spad_reader (|parseAndInterpret| stringBuffer))))))
```

5.3.35 defun parseAndInterpret

```
[ncParseAndInterpretString p48]
[$InteractiveMode p24]
[$boot p25]
[$spad p20]
[$e p??]
[$InteractiveFrame p??]
```

— defun parseAndInterpret —

```
(defun |parseAndInterpret| (str)
  (let (|$InteractiveMode| $boot $spad |$e|)
    (declare (special |$InteractiveMode| $boot $spad |$e|
                      |$InteractiveFrame|))
    (setq |$InteractiveMode| t)
    (setq $boot nil)
    (setq $spad t)
    (setq |$e| |$InteractiveFrame|)
    (|ncParseAndInterpretString| str)))
```

—————

5.3.36 defun ncParseAndInterpretString

```
[processInteractive p49]
[packageTran p67]
[parseFromString p48]
```

— defun ncParseAndInterpretString —

```
(defun |ncParseAndInterpretString| (s)
  (|processInteractive| (|packageTran| (|parseFromString| s)) nil))
```

—————

5.3.37 defun parseFromString

```
[next p38]
[ncloopParse p37]
[lineoftoks p113]
[incString p39]
[StreamNull p335]
[pf2Sex p301]
```

[macroExpanded p224]

— **defun parseFromString** —

```
(defun |parseFromString| (s)
  (setq s (|next| #'|ncloopParse| (|next| #'|lineoftoks| (|incString| s))))
  (unless (|StreamNull| s) (|pf2Sex| (|macroExpanded| (cadar s)))))
```

—————

5.3.38 **defvar \$interpOnly**

— **initvars** —

```
(defvar |$interpOnly| nil)
```

—————

5.3.39 **defvar \$minivectorNames**

— **initvars** —

```
(defvar |$minivectorNames| nil)
```

—————

5.3.40 **defvar \$domPvar**

— **initvars** —

```
(defvar |$domPvar| nil)
```

—————

5.3.41 **defun processInteractive**

Parser Output --> Interpreter

Top-level dispatcher for the interpreter. It sets local variables and then calls `processInteractive1` to do most of the work. This function receives the output from the parser. `[initializeTimedNames p??]`

```
[pairp p??]
[qcar p??]
[processInteractive1 p52]
[reportInstantiations p719]
[clrhash p??]
[writeHistModesAndValues p581]
[updateHist p567]
[$op p??]
[$Coerce p??]
[$compErrorMessageStack p??]
[$freeVars p??]
[$mapList p??]
[$compilingMap p??]
[$compilingLoop p??]
[$interpOnly p49]
[$whereCacheList p??]
[$timeGlobalName p??]
[$StreamFrame p??]
[$declaredMode p??]
[$localVars p??]
[$analyzingMapList p??]
[$lastLineInSEQ p??]
[$instantCoerceCount p??]
[$instantCanCoerceCount p??]
[$instantMmCondCount p??]
[$fortVar p??]
[$minivector p??]
[$minivectorCode p??]
[$minivectorNames p49]
[$domPvar p49]
[$inRetract p??]
[$instantRecord p??]
[$reportInstantiations p719]
[$ProcessInteractiveValue p52]
[$defaultFortVar p??]
[$interpreterTimedNames p??]
[$interpreterTimedClasses p??]
```

— **defun processInteractive** —

```
(defun |processInteractive| (form posnForm)
  (let (|$op| |$Coerce| |$compErrorMessageStack| |$freeVars|
        |$mapList| |$compilingMap| |$compilingLoop|
```

```

    |$interpOnly| |$whereCacheList| |$timeGlobalName|
    |$StreamFrame| |$declaredMode| |$localVars|
    |$analyzingMapList| |$lastLineInSEQ|
    |$instantCoerceCount| |$instantCanCoerceCount|
    |$instantMmCondCount| |$fortVar| |$minivector|
    |$minivectorCode| |$minivectorNames| |$domPvar|
    |$inRetract| object)
(declare (special |$op| |$Coerce| |$compErrorMessageStack|
    |$freeVars| |$mapList| |$compilingMap|
    |$compilingLoop| |$interpOnly| |$whereCacheList|
    |$timeGlobalName| |$StreamFrame| |$declaredMode|
    |$localVars| |$analyzingMapList| |$lastLineInSEQ|
    |$instantCoerceCount| |$instantCanCoerceCount|
    |$instantMmCondCount| |$fortVar| |$minivector|
    |$minivectorCode| |$minivectorNames| |$domPvar|
    |$inRetract| |$instantRecord| |$reportInstantiations|
    |$ProcessInteractiveValue| |$defaultFortVar|
    |$interpreterTimedNames| |$interpreterTimedClasses|))
(|initializeTimedNames| |$interpreterTimedNames| |$interpreterTimedClasses|)
(if (pairp form) ; compute name of operator
    (setq |$op| (qcar form))
    (setq |$op| form))
(setq |$Coerce| nil)
(setq |$compErrorMessageStack| nil)
(setq |$freeVars| nil)
(setq |$mapList| nil) ; list of maps being type analyzed
(setq |$compilingMap| nil) ; true when compiling a map
(setq |$compilingLoop| nil) ; true when compiling a loop body
(setq |$interpOnly| nil) ; true when in interp only mode
(setq |$whereCacheList| nil) ; maps compiled because of where
(setq |$timeGlobalName| '$compTimeSum|); see incrementTimeSum
(setq |$StreamFrame| nil) ; used in printing streams
(setq |$declaredMode| nil) ; weak type propagation for symbols
(setq |$localVars| nil) ; list of local variables in function
(setq |$analyzingMapList| nil) ; names of maps currently being analyzed
(setq |$lastLineInSEQ| t) ; see evalIF and friends
(setq |$instantCoerceCount| 0)
(setq |$instantCanCoerceCount| 0)
(setq |$instantMmCondCount| 0)
(setq |$defaultFortVar| 'x) ; default FORTRAN variable name
(setq |$fortVar| |$defaultFortVar|) ; variable name for FORTRAN output
(setq |$minivector| nil)
(setq |$minivectorCode| nil)
(setq |$minivectorNames| nil)
(setq |$domPvar| nil)
(setq |$inRetract| nil)
(setq object (|processInteractive| form posnForm))
(unless |$ProcessInteractiveValue|
    (when |$reportInstantiations|
        (|reportInstantiations|)

```

```

      (clrhash |$instantRecord|))
      (|writeHistModesAndValues|)
      (|updateHist|))
      object))

```

5.3.42 defvar \$ProcessInteractiveValue

— initvars —

```

(defvar |$ProcessInteractiveValue| nil "If true, no output or record")

```

5.3.43 defvar \$HTCompanionWindowID

— initvars —

```

(defvar |$HTCompanionWindowID| nil)

```

5.3.44 defun processInteractive1

This calls the analysis and output printing routines [recordFrame p887]

```

[startTimingProcess p??]
[interpretTopLevel p53]
[stopTimingProcess p??]
[recordAndPrint p56]
[objValUnwrap p??]
[objMode p??]
[$e p??]
[$ProcessInteractiveValue p52]
[$InteractiveFrame p??]

```

— defun processInteractive1 —

```

(defun |processInteractive1| (form posnForm)
  (let (|$e| object)

```

```
(declare (special |$e| |$ProcessInteractiveValue| |$InteractiveFrame|))
(setq |$e| |$InteractiveFrame|)
(|recordFrame| '|system|)
(|startTimingProcess| '|analysis|)
(setq object (|interpretTopLevel| form posnForm))
(|stopTimingProcess| '|analysis|)
(|startTimingProcess| '|print|)
(unless |$ProcessInteractiveValue|
  (|recordAndPrint| (|objValUnwrap| object) (|objModel| object)))
(|recordFrame| '|normal|)
(|stopTimingProcess| '|print|)
object))
```

5.3.45 defun interpretTopLevel

```
[interpreter p??]
[interpret p54]
[stopTimingProcess p??]
[peekTimedName p??]
[interpretTopLevel p53]
[$timedNameStack p??]
```

— defun interpretTopLevel —

```
(defun |interpretTopLevel| (x posnForm)
  (let (savedTimerStack c)
    (declare (special |$timedNameStack|))
    (setq savedTimerStack (copy |$timedNameStack|))
    (setq c (catch '|interpreter| (|interpret| x posnForm)))
    (do ()
      ((equal savedTimerStack |$timedNameStack|) nil)
      (|stopTimingProcess| (|peekTimedName|)))
    (if (eq c '|tryAgain|)
      (|interpretTopLevel| x posnForm)
      c)))
```

5.3.46 defvar \$genValue

If the `$genValue` variable is true then evaluate generated code, otherwise leave code unevaluated. If `$genValue` is false then we are compiling. This variable is only defined and used

locally.

— **initvars** —

```
(defvar |$genValue| nil "evaluate generated code if true")
```

—————

5.3.47 **defun** Type analyzes and evaluates expression x, returns object

```
[pairp p??]  
[interpret1 p54]  
[$env p??]  
[$eval p??]  
[$genValue p53]
```

— **defun interpret** —

```
(defun |interpret| (&rest arg &aux restarts x)  
  (dsetq (x . restarts) arg)  
  (let (|$env| |$eval| |$genValue| posnForm)  
    (declare (special |$env| |$eval| |$genValue|))  
    (if (pairp restarts)  
        (setq posnForm (car restarts))  
        (setq posnForm restarts))  
    (setq |$env| (list (list nil)))  
    (setq |$eval| t) ; generate code -- don't just type analyze  
    (setq |$genValue| t) ; evaluate all generated code  
    (|interpret1| x nil posnForm)))
```

—————

5.3.48 **defun** Dispatcher for the type analysis routines

This is the dispatcher for the type analysis routines. It type analyzes and evaluates the expression x in the rootMode (if non-nil) which may be `$EmptyMode`. It returns an object if evaluating, and a modeset otherwise. It creates the attributed tree. [mkAtreeWithSrcPos p??]

```
[putTarget p??]  
[bottomUp p??]  
[getArgValue p??]  
[objNew p??]  
[getValue p??]  
[interpret2 p55]
```

```
[keyedSystemError p??]
[$genValue p53]
[$eval p??]
```

— **defun interpret1** —

```
(defun |interpret1| (x rootMode posnForm)
  (let (node modeSet newRootMode argVal val)
    (declare (special |$genValue| |$eval|))
    (setq node (|mkAtreeWithSrcPos| x posnForm))
    (when rootMode (|putTarget| node rootMode))
    (setq modeSet (|bottomUp| node))
    (if (null |$eval|)
        modeSet
        (progn
          (if (null rootMode)
              (setq newRootMode (car modeSet))
              (setq newRootMode rootMode))
          (setq argVal (|getArgValue| node newRootMode))
          (cond
            ((and argVal (null |$genValue|))
             (|objNew| argVal newRootMode))
            ((and argVal (setq val (|getValue| node)))
             (|interpret2| val newRootMode posnForm)))
          (t
           (|keyedSystemError| 'S2IS0053 (list x)))))))
```

5.3.49 defun interpret2

This is the late interpretCoerce. I removed the call to coerceInteractive, so it only does the JENKS cases ALBI [objVal p??]

```
[objMode p??]
[pairp p??]
[member p1004]
[objNew p??]
[systemErrorHere p??]
[coerceInteractive p??]
[throwKeyedMsgCannotCoerceWithValue p??]
[$EmptyMode p??]
[$ThrowAwayMode p??]
```

— **defun interpret2** —

```
(defun |interpret2| (object m1 posnForm)
```



```

(declare (ignore posnForm))
(let (x m op ans)
(declare (special |$EmptyMode| |$ThrowAwayMode|))
(cond
  ((equal m1 |$ThrowAwayMode|) object)
  (t
   (setq x (|objVal| object))
   (setq m (|objMode| object))
   (cond
    ((equal m |$EmptyMode|)
     (cond
      ((and (pairp x)
            (progn (setq op (qcar x)) t)
                  (|member| op '(map stream))))
      (|objNew| x m1))
    ((equal m1 |$EmptyMode|)
     (|objNew| x m))
    (t
     (|systemErrorHere| "interpret2")))))
(m1
 (if (setq ans (|coerceInteractive| object m1))
     ans
     (|throwKeyedMsgCannotCoerceWithValue| x m m1)))
(t object))))))

```

5.3.50 defun Result Output Printing

Prints out the value `x` which is of type `m`, and records the changes in environment `$e` into `$InteractiveFrame` `$printAnyIfTrue` is documented in `setvart.boot`. It is controlled with the `)se me any` command. `[nequal p??]`

```

[output p??]
[putHist p568]
[objNewWrap p??]
[printTypeAndTime p58]
[printStorage p58]
[printStatisticsSummary p58]
[mkCompanionPage p??]
[recordAndPrintTest p??]
[$outputMode p??]
[$mkTestOutputType p??]
[$runTestFlag p??]
[$e p??]
[$mkTestFlag p??]
[$HTCompanionWindowID p52]

```

```

[QuietCommand p47]
[printStatsSummaryIfTrue p726]
[printTypeIfTrue p729]
[printStorageIfTrue p??]
[printTimeIfTrue p728]
[Void p??]
[algebraOutputStream p739]
[collectOutput p??]
[EmptyMode p??]
[printVoidIfTrue p730]
[outputMode p??]
[printAnyIfTrue p712]

```

— **defun recordAndPrint** —

```

(defun |recordAndPrint| (x md)
  (let (|$outputMode| xp mdp mode)
    (declare (special |$outputMode| |$mkTestOutputType| |$runTestFlag| |$e|
                      |$mkTestFlag| |$HTCompanionWindowID| |$QuietCommand|
                      |$printStatsSummaryIfTrue| |$printTypeIfTrue|
                      |$printStorageIfTrue| |$printTimeIfTrue| |$Void|
                      |$algebraOutputStream| |$collectOutput| |$EmptyMode|
                      |$printVoidIfTrue| |$outputMode| |$printAnyIfTrue|))

    (cond
      ((and (equal md '(|Any|)) |$printAnyIfTrue|)
       (setq mdp (car x))
       (setq xp (cdr x)))
      (t
       (setq mdp md)
       (setq xp x)))
    (setq |$outputMode| md)
    (if (equal md |$EmptyMode|)
        (setq mode (|quadSch|))
        (setq mode md))
    (when (or (nequal md |$Void|) |$printVoidIfTrue|)
      (unless |$collectOutput| (terpri |$algebraOutputStream|))
      (unless |$QuietCommand| (|output| xp mdp)))
    (|putHist| '% '|value| (|objNewWrap| x md) |$e|)
    (when (or |$printTimeIfTrue| |$printTypeIfTrue|)
      (|printTypeAndTime| xp mdp))
    (when |$printStorageIfTrue| (|printStorage|))
    (when |$printStatsSummaryIfTrue| (|printStatsSummary|))
    (when (integerp |$HTCompanionWindowID|) (|mkCompanionPage| md))
    (cond
      (|$mkTestFlag| (|recordAndPrintTest| md))
      (|$runTestFlag|
       (setq |$mkTestOutputType| md)
       '|done|)
      (t '|done|))))

```

5.3.51 defun printStatisticsSummary

```
[sayKeyedMsg p331]
[statisticsSummary p??]
[$collectOutput p??]
```

— defun printStatisticsSummary —

```
(defun |printStatisticsSummary| ()
  (declare (special |$collectOutput|))
  (unless |$collectOutput|
    (|sayKeyedMsg| 'S2GL0017 (list (|statisticsSummary|)))))
```

5.3.52 defun printStorage

```
[makeLongSpaceString p??]
[$interpreterTimedClasses p??]
[$collectOutput p??]
[$interpreterTimedNames p??]
```

— defun printStorage —

```
(defun |printStorage| ()
  (declare (special |$interpreterTimedClasses| |$collectOutput|
    |$interpreterTimedNames|))
  (unless |$collectOutput|
    (|sayKeyedMsg| 'S2GL0016
      (list
        (|makeLongSpaceString|
          |$interpreterTimedNames|
          |$interpreterTimedClasses|))))))
```

5.3.53 defun printTypeAndTime

```
[printTypeAndTimeSaturn p60]
[printTypeAndTimeNormal p59]
```

```
[$saturn p??]
```

— **defun printTypeAndTime** —

```
(defun |printTypeAndTime| (x m)
  (declare (special |$saturn|))
  (if |$saturn|
    (|printTypeAndTimeSaturn| x m)
    (|printTypeAndTimeNormal| x m)))
```

5.3.54 defun printTypeAndTimeNormal

```
[qcar p??]
[paip p??]
[retract p??]
[objNewWrap p??]
[objMode p??]
[sameUnionBranch p61]
[makeLongTimeString p??]
[msgText p62]
[sayKeyedMsg p331]
[justifyMyType p62]
[$outputLines p??]
[$collectOutput p??]
[$printTypeIfTrue p729]
[$printTimeIfTrue p728]
[$outputLines p??]
[$interpreterTimedNames p??]
[$interpreterTimedClasses p??]
```

— **defun printTypeAndTimeNormal** —

```
(defun |printTypeAndTimeNormal| (x m)
  (let (xp mp timeString result)
    (declare (special |$outputLines| |$collectOutput| |$printTypeIfTrue|
                      |$printTimeIfTrue| |$outputLines|
                      |$interpreterTimedNames| |$interpreterTimedClasses|))
    (cond
      ((and (paip m) (eq (qcar m) '|Union|))
       (setq xp (|retract| (|objNewWrap| x m)))
       (setq mp (|objMode| xp))
       (setq m
        (cons '|Union|
              (append
```

```

      (dolist (arg (qcdr m) (nreverse result))
        (when (|sameUnionBranch| arg mp) (push arg result)))
      (list "...")))))))
(when |$printTimeIfTrue|
  (setq timeString
    (|makeLongTimeString|
      |$interpreterTimedNames|
      |$interpreterTimedClasses|)))
(cond
  ((and |$printTimeIfTrue| |$printTypeIfTrue|)
    (if |$collectOutput|
      (push (|msgText| 'S2GL0012 (list m)) |$outputLines|)
      (|sayKeyedMsg| 'S2GL0014 (list m timeString))))
    (|$printTimeIfTrue|
      (unless |$collectOutput| (|sayKeyedMsg| 'S2GL0013 (list timeString))))
    (|$printTypeIfTrue|
      (if |$collectOutput|
        (push (|justifyMyType| (|msgText| 'S2GL0012 (list m))) |$outputLines|)
        (|sayKeyedMsg| 'S2GL0012 (list m)))))))

```

5.3.55 defun printTypeAndTimeSaturn

```

[makeLongTimeString p??]
[form2StringAsTeX p??]
[devaluate p??]
[printAsTeX p61]
[$printTimeIfTrue p728]
[$printTypeIfTrue p729]
[$interpreterTimedClasses p??]
[$interpreterTimedNames p??]

```

— defun printTypeAndTimeSaturn —

```

(defun |printTypeAndTimeSaturn| (x m)
  (declare (ignore x))
  (let (timeString typeString)
    (declare (special |$printTimeIfTrue| |$printTypeIfTrue|
      |$interpreterTimedClasses| |$interpreterTimedNames|))
    (if |$printTimeIfTrue|
      (setq timeString
        (|makeLongTimeString|
          |$interpreterTimedNames|
          |$interpreterTimedClasses|))
      (setq timeString ""))
    (if |$printTypeIfTrue|

```

```

(setq typeString (|form2StringAsTeX| (|devaluate| m)))
(setq typeString "")
(when |$printTypeIfTrue|
  (|printAsTeX| "\\axPrintType{")
  (if (consp typeString)
    (mapc #'|printAsTeX| typeString)
    (|printAsTeX| typeString))
  (|printAsTeX| "}"))
(when |$printTimeIfTrue|
  (|printAsTeX| "\\axPrintTime{")
  (|printAsTeX| timeString)
  (|printAsTeX| "}"))))

```

5.3.56 defun printAsTeX

[*\$texOutputStream* p??]

— defun printAsTeX 0 —

```

(defun |printAsTeX| (x)
  (declare (special |$texOutputStream|))
  (princ x |$texOutputStream|))

```

5.3.57 defun sameUnionBranch

```

sameUnionBranch(uArg, m) ==
  uArg is [":", ., t] => t = m
  uArg = m

```

— defun sameUnionBranch 0 —

```

(defun |sameUnionBranch| (uArg m)
  (let (t1 t2 t3)
    (cond
      ((and (pairp uArg)
        (eq (qcar uArg) '|:|)
        (progn
          (setq t1 (qcdr uArg))
          (and (pairp t1)
            (progn

```

```

      (setq t2 (qcdr t1))
      (and (pairp t2)
            (eq (qcdr t2) nil)
            (progn (setq t3 (qcar t2)) t))))))
    (equal t3 m))
  (t (equal uArg m))))))

```

5.3.58 defun msgText

```

[segmentKeyedMsg p332]
[getKeyedMsg p331]
[substituteSegmentedMsg p??]
[flowSegmentedMsg p??]
[$linelength p751]
[$margin p751]

```

— defun msgText —

```

(defun |msgText| (key args)
  (let (msg)
    (declare (special $linelength $margin))
    (setq msg (|segmentKeyedMsg| (|getKeyedMsg| key)))
    (setq msg (|substituteSegmentedMsg| msg args))
    (setq msg (|flowSegmentedMsg| msg $linelength $margin))
    (apply #'concat (mapcar #'princ-to-string (cdar msg)))))

```

5.3.59 defun Right-justify the Type output

```

[fillerSpaces p20]
[$linelength p751]

```

— defun justifyMyType —

```

(defun |justifyMyType| (arg)
  (let (len)
    (declare (special $linelength))
    (setq len (|#| arg))
    (if (> len $linelength)
        arg
        (concat (|fillerSpaces| (- $linelength len)) arg))))

```

5.3.60 defun Destructively fix quotes in strings

```
[unescapeStringsInForm p63]
[$funnyBacks p??]
[$funnyQuote p??]
```

— **defun unescapeStringsInForm** —

```
(defun |unescapeStringsInForm| (form)
  (let (str)
    (declare (special |$funnyBacks| |$funnyQuote|))
    (cond
      ((stringp form)
       (setq str (nsubstitute #\" |$funnyQuote| form))
       (nsubstitute #\\ |$funnyBacks| str))
      ((consp form)
       (|unescapeStringsInForm| (car form))
       (|unescapeStringsInForm| (cdr form))
       form)
      (t form))))
```

5.3.61 Include a file into the stream

```
[ST p??]
[intloopInclude0 p63]
```

— **defun intloopInclude** —

```
(defun |intloopInclude| (name n)
  "Include a file into the stream"
  (with-open-file (st name) (|intloopInclude0| st name n)))
```

5.3.62 defun intloopInclude0

```
[incStream p74]
[intloopProcess p64]
[next p38]
[intloopEchoParse p70]
```



```
[insertpile p337]
[lineoftoks p113]
[$lines p??]
```

— **defun intloopInclude0** —

```
(defun |intloopInclude0| (|st| |name| |n|)
  (let (|$lines|)
    (declare (special |$lines|))
    (setq |$lines| (|incStream| |st| |name|))
    (|intloopProcess| |n| NIL
      (|next| #'|intloopEchoParse|
        (|next| #'|insertpile|
          (|next| #'|lineoftoks|
            |$lines|))))))
```

5.3.63 defun intloopProcess

```
[StreamNull p335]
[pfAbSynOp? p414]
[setCurrentLine p41]
[tokPart p415]
[intloopProcess p64]
[intloopSpadProcess p65]
[$systemCommandFunction p??]
[$systemCommandFunction p??]
```

— **defun intloopProcess** —

```
(defun |intloopProcess| (n interactive s)
  (let (ptree lines t1)
    (declare (special |$systemCommandFunction|))
    (cond
      ((|StreamNull| s) n)
      (t
        (setq t1 (car s))
        (setq lines (car t1))
        (setq ptree (cadr t1))
        (cond
          ((|pfAbSynOp?| ptree '|command|)
            (when interactive (|setCurrentLine| (|tokPart| ptree)))
            (funcall |$systemCommandFunction| (|tokPart| ptree))
            (|intloopProcess| n interactive (cdr s)))
          (t
```

```
(|intloopProcess|
  (|intloopSpadProcess| n lines ptree interactive)
  interactive (cdr s))))))
```

5.3.64 defun intloopSpadProcess

```
[flung p??]
[SpadCompileItem p??]
[intCoerceFailure p31]
[intSpadReader p32]
[ncPutQ p418]
[CatchAsCan p??]
[Catch p??]
[intloopSpadProcess,interp p66]
[$currentCarrier p??]
[$ncMsgList p27]
[$intCoerceFailure p31]
[$intSpadReader p32]
[$prevCarrier p??]
[$stepNo p??]
[$NeedToSignalSessionManager p43]
[flung p??]
```

— defun intloopSpadProcess —

```
(defun |intloopSpadProcess| (stepNo lines ptree interactive?)
  (let (|$stepNo| result cc)
    (declare (special |$stepNo| |$prevCarrier| |$intSpadReader| |flung|
                      |$intCoerceFailure| |$ncMsgList| |$currentCarrier|
                      |$NeedToSignalSessionManager|))
    (setq |$stepNo| stepNo)
    (setq |$currentCarrier| (setq cc (list '|carrier|)))
    (|ncPutQ| cc '|stepNumber| stepNo)
    (|ncPutQ| cc '|messages| |$ncMsgList|)
    (|ncPutQ| cc '|lines| lines)
    (setq |$ncMsgList| nil)
    (setq result
      (catch '|SpadCompileItem|
        (catch |$intCoerceFailure|
          (catch |$intSpadReader|
            (|intloopSpadProcess,interp| cc ptree interactive?))))))
    (setq |$NeedToSignalSessionManager| t)
    (setq |$prevCarrier| |$currentCarrier|)
    (cond
```

```

((eq result '|ncEnd|) stepNo)
((eq result '|ncError|) stepNo)
((eq result '|ncEndItem|) stepNo)
(t (1+ stepNo))))

```

5.3.65 defun intloopSpadProcess,interp

```

[ncConversationPhase p69]
[ncEltQ p418]
[ncError p70]

```

— defun intloopSpadProcess,interp —

```

(defun |intloopSpadProcess,interp| (cc ptree interactive?)
  (|ncConversationPhase| #'|phParse| (list cc ptree))
  (|ncConversationPhase| #'|phMacro| (list cc))
  (|ncConversationPhase| #'|phIntReportMsgs| (list cc interactive?))
  (|ncConversationPhase| #'|phInterpret| (list cc))
  (unless (eql (length (|ncEltQ| cc '|messages|)) 0) (|ncError|)))

```

5.3.66 defun phParse

TPDHERE: The pform function has a leading percent sign. fix this

```
phParse: carrier[tokens,...] -> carrier[ptree, tokens,...]
```

```

[intSayKeyedMsg p67]
[pform p??]
[ncPutQ p418]

```

— defun phParse —

```

(defun |phParse| (carrier ptree)
  (|ncPutQ| carrier '|ptree| ptree)
  'ok)

```

5.3.67 defun intSayKeyedMsg

[sayKeyedMsg p331]
 [packageTran p67]

— defun intSayKeyedMsg —

```
(defun |intSayKeyedMsg| (key args)
  (|sayKeyedMsg| (|packageTran| key) (|packageTran| args)))
```

—————

5.3.68 defun packageTran

[packageTran p67]

— defun packageTran 0 —

```
(defun |packageTran| (sex)
  (cond
    ((symbolp sex)
     (cond
       ((eq *package* (symbol-package sex)) sex)
       (t (intern (string sex)))))
    ((consp sex)
     (rplaca sex (|packageTran| (car sex)))
     (rplacd sex (|packageTran| (cdr sex)))
     sex)
    (t sex)))
```

—————

5.3.69 defun phIntReportMsgs

carrier[lines,messages,...]-> carrier[lines,messages,...]

[ncEltQ p418]
 [ncPutQ p418]
 [processMsgList p371]
 [intSayKeyedMsg p67]
 [\$erMsgToss p??]

— defun phIntReportMsgs —

```
(defun |phIntReportMsgs| (carrier interactive?)
  (declare (ignore interactive?))
  (let (nerr msgs lines)
    (declare (special |$erMsgToss|))
    (cond
      (|$erMsgToss| 'ok)
      (t
       (setq lines (|ncEltQ| carrier '|lines|))
       (setq msgs (|ncEltQ| carrier '|messages|))
       (setq nerr (length msgs))
       (|ncPutQ| carrier '|ok?' (eql nerr 0))
       (cond
         ((eql nerr 0) 'ok)
         (t
          (|processMsgList| msgs lines)
          (|intSayKeyedMsg| 'S2CTP010 (list nerr)
                           'ok)))))))
```

5.3.70 defun phInterpret

[ncEltQ p418]
 [intInterpretPform p68]
 [ncPutQ p418]

— defun phInterpret —

```
(defun |phInterpret| (carrier)
  (let (val ptree)
    (setq ptree (|ncEltQ| carrier '|ptree|))
    (setq val (|intInterpretPform| ptree))
    (|ncPutQ| carrier '|value| val)))
```

5.3.71 defun intInterpretPform

[processInteractive p49]
 [zeroOneTran p69]
 [packageTran p67]
 [pf2Sex p301]

— defun intInterpretPform —

```
(defun |intInterpretPform| (pf)
  (|processInteractive| (|zeroOneTran| (|packageTran| (|pf2Sex| pf))) pf))
```

5.3.72 defun zeroOneTran

```
[nsubst p??]
```

— defun zeroOneTran 0 —

```
(defun |zeroOneTran| (sex)
  (nsubst '|$EmptyMode| '? sex))
```

5.3.73 defun ncConversationPhase

```
[ncConversationPhase,wrapup p69]
[$ncMsgList p27]
```

— defun ncConversationPhase —

```
(defun |ncConversationPhase| (fn args)
  (let (|$ncMsgList| carrier)
    (declare (special |$ncMsgList|))
    (setq carrier (car args))
    (setq |$ncMsgList| nil)
    (unwind-protect
      (apply fn args)
      (|ncConversationPhase,wrapup| carrier))))
```

5.3.74 defun ncConversationPhase,wrapup

```
[$ncMsgList p27]
```

— defun ncConversationPhase,wrapup —

```
(defun |ncConversationPhase,wrapup| (carrier)
  (declare (special |$ncMsgList|))
  ((lambda (Var5 m)
```

```

(loop
  (cond
    ((or (atom Var5) (progn (setq m (car Var5)) nil))
      (return nil))
    (t
      (|ncPutQ| carrier '|messages| (cons m (|ncEltQ| carrier '|messages|))))
    (setq Var5 (cdr Var5))))
  |$ncMsgList| nil))

```

5.3.75 defun ncError

[SpadCompileItem p??]

— defun ncError 0 —

```

(defun |ncError| ()
  (throw '|SpadCompileItem| '|ncError|))

```

5.3.76 defun intloopEchoParse

[ncloopDQlines p72]
 [setCurrentLine p41]
 [mkLineList p71]
 [ncloopPrintLines p71]
 [npParse p143]
 [dqToList p346]
 [\$EchoLines p??]
 [\$lines p??]

— defun intloopEchoParse —

```

(defun |intloopEchoParse| (s)
  (let (cudr lines stream dq t1)
    (declare (special |$EchoLines| |$lines|))
    (setq t1 (car s))
    (setq dq (car t1))
    (setq stream (cadr t1))
    (setq t1 (|ncloopDQlines| dq |$lines|))
    (setq lines (car t1))
    (setq cudr (cadr t1))

```

```
(|setCurrentLine| (|mkLineList| lines))
(when |$EchoLines| (|ncloopPrintLines| lines))
(setq |$lines| caddr)
(cons (list (list lines (|npParse| (|dqToList| dq)))) (cdr s))))
```

5.3.77 defun ncloopPrintLines

```
;ncloopPrintLines lines ==
;      for line in lines repeat WRITE_-LINE CDR line
;      WRITE_-LINE ' " "
```

— defun ncloopPrintLines 0 —

```
(defun |ncloopPrintLines| (lines)
  ((lambda (Var4 line)
    (loop
      (cond
        ((or (atom Var4) (progn (setq line (car Var4)) nil))
         (return nil))
        (t (write-line (cdr line))))
      (setq Var4 (cdr Var4))))
    lines nil)
  (write-line " " ))
```

5.3.78 defun mkLineList

```
;mkLineList lines ==
;  l := [CDR line for line in lines | nonBlank CDR line]
;  #l = 1 => CAR l
;  l
```

— defun mkLineList —

```
(defun |mkLineList| (lines)
  (let (l)
    (setq l
      ((lambda (Var2 Var1 line)
        (loop
          (cond
```



```

      ((or (atom Var1) (progn (setq line (car Var1)) nil))
       (return (nreverse Var2)))
    (t
     (and (|nonBlank| (cdr line))
          (setq Var2 (cons (cdr line) Var2))))
    (setq Var1 (cdr Var1)))
  nil lines nil))
(cond
 ((eql (length l) 1) (car l))
 (t l)))

```

5.3.79 defun nonBlank

```

;nonBlank str ==
; value := false
; for i in 0..MAXINDEX str repeat
;   str.i ^= char " " =>
;     value := true
;   return value
; value

```

— defun nonBlank 0 —

```

(defun |nonBlank| (str)
  (let (value)
    ((lambda (Var3 i)
      (loop
       (cond
        ((> i Var3) (return nil))
        (t
         (cond
          ((not (equal (elt str i) (|char| ' | |)))
           (identity (progn (setq value t) (return value))))))
       (setq i (+ i 1))))
     (maxindex str) 0)
    value))

```

5.3.80 defun ncloopDQlines

[StreamNull p335]
 [poGlobalLinePosn p73]

[tokPosn p415]
 [streamChop p73]

— **defun ncloopDQlines** —

```
(defun |ncloopDQlines| (dq stream)
  (let (b a)
    (|StreamNull| stream)
    (setq a (|poGlobalLinePosn| (|tokPosn| (cadr dq))))
    (setq b (|poGlobalLinePosn| (caar stream)))
    (|streamChop| (+ (- a b) 1) stream)))
```

—

5.3.81 **defun poGlobalLinePosn**

[lnGlobalNum p348]
 [poGetLineObject p363]
 [ncBug p370]

— **defun poGlobalLinePosn** —

```
(defun |poGlobalLinePosn| (posn)
  (if posn
    (|lnGlobalNum| (|poGetLineObject| posn))
    (|ncBug| "old style pos objects have no global positions" nil)))
```

—

5.3.82 **defun streamChop**

Note that changing the name “lyne” to “line” will break the system. I do not know why. The symptom shows up when there is a file with a large contiguous comment spanning enough lines to overflow the stack. [StreamNull p335]

[streamChop p73]
 [ncloopPrefix? p459]

— **defun streamChop** —

```
(defun |streamChop| (n s)
  (let (d c lyne b a tmp1)
    (cond
      ((|StreamNull| s) (list nil nil))
      ((eq1 n 0) (list nil s))
```

```
(t
  (setq tmp1 (|streamChop| (- n 1) (cdr s)))
  (setq a (car tmp1))
  (setq b (cadr tmp1))
  (setq lyne (car s))
  (setq c (|ncloopPrefix?| ")command" (cdr lyne)))
  (setq d (cons (car lyne) (cond (c c) (t (cdr lyne)))))
  (list (cons d a) b))))
```

5.3.83 defun ncloopInclude0

```
[incStream p74]
[ncloopProcess p??]
[next p38]
[ncloopEchoParse p??]
[insertpile p337]
[lineoftoks p113]
[$lines p??]
```

— defun ncloopInclude0 —

```
(defun |ncloopInclude0| (st name n)
  (let (|$lines|)
    (declare (special |$lines|))
    (setq |$lines| (|incStream| st name))
    (|ncloopProcess| n nil
      (|next| #'|ncloopEchoParse|
        (|next| #'|insertpile|
          (|next| #'|lineoftoks|
            |$lines|))))))
```

5.3.84 defun incStream

```
[incRenumber p75]
[incLude p78]
[incRgen p104]
[Top p78]
```

— defun incStream —

```
(defun |incStream| (st fn)
```

```
(declare (special |Top|))
(|incRenumber| (|incLude| 0 (|incRgen| st) 0 (list fn) (list |Top|))))
```

5.3.85 defun incRenumber

```
[incZip p75]
[incIgen p76]
```

— defun incRenumber —

```
(defun |incRenumber| (ssx)
  (|incZip| #'|incRenumberLine| ssx (|incIgen| 0)))
```

5.3.86 defun incZip

```
[Delay p105]
[incZip1 p75]
```

— defun incZip —

```
(defun |incZip| (g f1 f2)
  (|Delay| #'|incZip1| (list g f1 f2)))
```

5.3.87 defun incZip1

```
[StreamNull p335]
[incZip p75]
```

— defun incZip1 —

```
(defun |incZip1| (&rest z)
  (let (f2 f1 g)
    (setq g (car z))
    (setq f1 (cadr z))
    (setq f2 (caddr z))
    (cond
```

```
((|StreamNull| f1) |StreamNil|)
((|StreamNull| f2) |StreamNil|)
(t
 (cons
  (funcall g (car f1) (car f2))
  (|incZip| g (cdr f1) (cdr f2))))))
```

5.3.88 defun incIgen

[Delay p105]
[incIgen1 p76]

— defun incIgen —

```
(defun |incIgen| (n)
  (|Delay| #'|incIgen1| (list n)))
```

5.3.89 defun incIgen1

[incIgen p76]

— defun incIgen1 —

```
(defun |incIgen1| (&rest z)
  (let (n)
    (setq n (car z))
    (setq n (+ n 1))
    (cons n (|incIgen| n))))
```

5.3.90 defun incRenumberLine

[incRenumberItem p77]
[incHandleMessage p77]

— defun incRenumberLine —

```
(defun |incRenumberLine| (x1 gno)
  (let (l)
    (setq l (|incRenumberItem| (elt x1 0) gno))
    (|incHandleMessage| x1)
    l))
```

5.3.91 defun incRenumberItem

[lnSetGlobalNum p348]

— defun incRenumberItem —

```
(defun |incRenumberItem| (f i)
  (let (l)
    (setq l (caar f))
    (|lnSetGlobalNum| l i) f))
```

5.3.92 defun incHandleMessage

[ncSoftError p353]

[ncBug p370]

— defun incHandleMessage 0 —

```
(defun |incHandleMessage| (x)
  "Message handling for the source includer"
  (let ((msgtype (elt (elt x 1) 1))
        (pos (car (elt x 0)))
        (key (car (elt (elt x 1) 0)))
        (args (cadr (elt (elt x 1) 0))))
    (cond
      ((eq msgtype '|none|) 0)
      ((eq msgtype '|error|) (|ncSoftError| pos key args))
      ((eq msgtype '|warning|) (|ncSoftError| pos key args))
      ((eq msgtype '|say|) (|ncSoftError| pos key args))
      (t (|ncBug| key args)))))
```

5.3.93 defun incLude

[Delay p105]
 [incLude1 p82]

— defun incLude —

```
(defun |incLude| (eb ss ln ufos states)
  (|Delay| #'|incLude1| (list eb ss ln ufos states)))
```

—————

5.3.94 defmacro Rest

— defmacro Rest —

```
(defmacro |Rest| ()
  "used in incLude1 for parsing; s is not used."
  '(|incLude| eb (cdr ss) lno ufos states))
```

—————

5.3.95 defvar \$Top

— initvars —

```
(defvar |Top| 1 "used in incLude1 for parsing")
```

—————

5.3.96 defvar \$IfSkipToEnd

— initvars —

```
(defvar |IfSkipToEnd| 10 "used in incLude1 for parsing")
```

—————

5.3.97 defvar \$IfKeepPart

— initvars —

```
(defvar |IfKeepPart| 11 "used in include1 for parsing")
```

—————

5.3.98 defvar \$IfSkipPart

— initvars —

```
(defvar |IfSkipPart| 12 "used in include1 for parsing")
```

—————

5.3.99 defvar \$ElseifSkipToEnd

— initvars —

```
(defvar |ElseifSkipToEnd| 20 "used in include1 for parsing")
```

—————

5.3.100 defvar \$ElseifKeepPart

— initvars —

```
(defvar |ElseifKeepPart| 21 "used in include1 for parsing")
```

—————

5.3.101 defvar \$ElseifSkipPart

— initvars —


```
(defvar |ElseifSkipPart| 22 "used in include1 for parsing")
```

5.3.102 defvar \$ElseSkipToEnd

— initvars —

```
(defvar |ElseSkipToEnd| 30 "used in include1 for parsing")
```

5.3.103 defvar \$ElseKeepPart

— initvars —

```
(defvar |ElseKeepPart| 31 "used in include1 for parsing")
```

5.3.104 defvar \$Top?

[quotient p??]

— defun Top? 0 —

```
(defun |Top?| (|st|)
  "used in include1 for parsing"
  (eq1 (quotient |st| 10) 0))
```

5.3.105 defvar \$If?

[quotient p??]

— defun If? —

```
(defun |If?| (|st|)
  "used in include1 for parsing"
  (eq1 (quotient |st| 10) 1))
```

5.3.106 defvar \$Elseif?

[QUOTIENT p??]

— defun Elseif? —

```
(defun |Elseif?| (|st|)
  "used in include1 for parsing"
  (eq1 (quotient |st| 10) 2))
```

5.3.107 defvar \$Else?

[QUOTIENT p??]

— defun Else? —

```
(defun |Else?| (|st|)
  "used in include1 for parsing"
  (eq1 (quotient |st| 10) 3))
```

5.3.108 defvar \$SkipEnd?

[remainder p??]

— defun SkipEnd? —

```
(defun |SkipEnd?| (|st|)
  "used in include1 for parsing"
  (eq1 (remainder |st| 10) 0))
```

5.3.109 defvar \$KeepPart?

[remainder p??]

— defun KeepPart? —

```
(defun |KeepPart?| (|st|)
  "used in incLude1 for parsing"
  (eq1 (remainder |st| 10) 1))
```

—————

5.3.110 defvar \$SkipPart?

[remainder p??]

— defun SkipPart? —

```
(defun |SkipPart?| (|st|)
  "used in incLude1 for parsing"
  (eq1 (remainder |st| 10) 2))
```

—————

5.3.111 defvar \$Skipping?

[KeepPart? p82]

— defun Skipping? —

```
(defun |Skipping?| (|st|)
  "used in incLude1 for parsing"
  (null (|KeepPart?| |st|)))
```

—————

5.3.112 defun incLude1

```
[StreamNull p335]
[Top? p80]
[xlPrematureEOF p87]
[Skipping? p82]
```

[xlSkip p90]
 [Rest p78]
 [xlOK p87]
 [xlOK1 p88]
 [concat p1003]
 [incCommandTail p102]
 [xlSay p91]
 [xlNoSuchFile p92]
 [xlCannotRead p93]
 [incActive? p104]
 [xlFileCycle p93]
 [incLude p78]
 [incFileInput p103]
 [incAppend p88]
 [inclFname p103]
 [xlConActive p94]
 [xlConStill p95]
 [incConsoleInput p103]
 [incNConsoles p104]
 [xlConsole p95]
 [xlSkippingFin p96]
 [xlPrematureFin p96]
 [assertCond p97]
 [ifCond p90]
 [If? p80]
 [Elseif? p81]
 [xIfSyntax p97]
 [SkipEnd? p81]
 [KeepPart? p82]
 [SkipPart? p82]
 [xIfBug p98]
 [xlCmdBug p99]
 [expand-tabs p??]
 [incClassify p100]

— **defun incLude1** —

```

(defun |incLude1| (&rest z)
  (let (pred s1 n tail head includee fn1 info str state lno states
        ufos ln ss eb)
    (setq eb (car z))
    (setq ss (cadr . (z)))
    (setq ln (caddr . (z)))
    (setq ufos (caddr . (z)))
    (setq states (car (cddddr . (z))))
    (setq lno (+ ln 1))
    (setq state (elt states 0))

```

```

(cond
  ((|StreamNull| ss)
    (cond
      ((null (|Top?| state))
        (cons (|xlPrematureEOF| eb "--premature end" lno ufos)
          |StreamNil|))
      (t |StreamNil|)))
  (t
    (progn
      (setq str (expand-tabs (car ss)))
      (setq info (|incClassify| str))
      (cond
        ((null (elt info 0))
          (cond
            ((|Skipping?| state)
              (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
            (t
              (cons (|xlOK| eb str lno (elt ufos 0)) (|Rest|))))
          ((equal (elt info 2) "other")
            (cond
              ((|Skipping?| state)
                (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
              (t
                (cons
                  (|xlOK| eb str (concat "command" str) lno (elt ufos 0))
                  (|Rest|))))
          ((equal (elt info 2) "say")
            (cond
              ((|Skipping?| state)
                (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
              (t
                (progn
                  (setq str (|incCommandTail| str info))
                  (cons (|xlSay| eb str lno ufos str)
                    (cons (|xlOK| eb str lno (ELT ufos 0)) (|Rest|))))))
          ((equal (elt info 2) "include")
            (cond
              ((|Skipping?| state)
                (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
              (t
                (progn
                  (setq fn1 (|inclFname| str info))
                  (cond
                    ((null fn1)
                      (cons (|xlNoSuchFile| eb str lno ufos fn1) (|Rest|)))
                    ((null (probe-file fn1))
                      (cons (|xlCannotRead| eb str lno ufos fn1) (|Rest|)))
                    ((|incActive?| fn1 ufos)
                      (cons (|xlFileCycle| eb str lno ufos fn1) (|Rest|)))
                    (t

```

```

(progn
  (setq includee
    (|incLude| (+ eb (elt info 1))
              (|incFileInput| fn1)
              0
              (cons fn1 ufos)
              (cons |Top| states)))
    (cons (|xlOK| eb str lno (elt ufos 0))
          (|incAppend| includee (|Rest|))))))
((equal (elt info 2) "console")
 (cond
  ((|Skipping?| state)
   (cons (|xlSkip| eb str lno (elt ufos 0)) (|Rest|)))
  (t
   (progn
    (setq head
      (|incLude| (+ eb (elt info 1))
                (|incConsoleInput|)
                0
                (cons "console" ufos)
                (cons |Top| states)))
      (setq tail (|Rest|))
      (setq n (|incNConsoles| ufos))
      (cond
       ((< 0 n)
        (setq head
          (cons (|xlConActive| eb str lno ufos n) head))
          (setq tail
            (cons (|xlConStill| eb str lno ufos n) tail))))
        (setq head (cons (|xlConsole| eb str lno ufos) head))
        (cons (|xlOK| eb str lno (elt ufos 0))
              (|incAppend| head tail))))))
    ((equal (elt info 2) "fin")
     (cond
      ((|Skipping?| state)
       (cons (|xlSkippingFin| eb str lno ufos) (|Rest|)))
      ((null (|Top?| state))
       (cons (|xlPrematureFin| eb str lno ufos) |StreamNil|))
      (t
       (cons (|xlOK| eb str lno (elt ufos 0)) |StreamNil|)))
    ((equal (elt info 2) "assert")
     (cond
      ((|Skipping?| state)
       (cons (|xlSkippingFin| eb str lno ufos) (|Rest|)))
      (t
       (progn
        (|assertCond| str info)
        (cons (|xlOK| eb str lno (elt ufos 0))
              (|incAppend| includee (|Rest|))))))
    ((equal (elt info 2) "if")

```

```

(progn
  (setq s1
    (cond
      ((|Skipping?| state) |IfSkipToEnd|)
      (t
        (cond
          ((|ifCond| str info) |IfKeepPart|)
          (t |IfSkipPart|))))))
    (cons (|x1OK| eb str lno (elt ufos 0))
      (|incLude| eb (cdr ss) lno ufos (cons s1 states)))))
  ((equal (elt info 2) "elseif")
    (cond
      ((and (null (|If?| state)) (null (|Elseif?| state)))
        (cons (|xlIfSyntax| eb str lno ufos info states)
          |StreamNil|))
      (t
        (cond
          ((or (|SkipEnd?| state)
            (|KeepPart?| state)
            (|SkipPart?| state))
            (setq s1
              (cond
                ((|SkipPart?| state)
                  (setq pred (|ifCond| str info))
                  (cond
                    (pred |ElseifKeepPart|)
                    (t |ElseifSkipPart|)))
                (t |ElseifSkipToEnd|)))
              (cons (|x1OK| eb str lno (elt ufos 0))
                (|incLude| eb (cdr ss) lno ufos (cons s1 (cdr states)))))
            (t
              (cons (|xlIfBug| eb str lno ufos) |StreamNil|))))))
      ((equal (elt info 2) "else")
        (cond
          ((and (null (|If?| state)) (null (|Elseif?| state)))
            (cons (|xlIfSyntax| eb str lno ufos info states)
              |StreamNil|))
          (t
            (cond
              ((or (|SkipEnd?| state)
                (|KeepPart?| state)
                (|SkipPart?| state))
                (setq s1
                  (cond ((|SkipPart?| state) |ElseKeepPart|) (t |ElseSkipToEnd|)))
                (cons (|x1OK| eb str lno (elt ufos 0))
                  (|incLude| eb (cdr ss) lno ufos (cons s1 (cdr states)))))
              (t
                (cons (|xlIfBug| eb str lno ufos) |StreamNil|))))))
          ((equal (elt info 2) "endif")
            (cond

```

```

(|Top?| state)
  (cons (|xlIfSyntax| eb str lno ufos info states)
        |StreamNil|))
(t
  (cons (|xlOK| eb str lno (elt ufos 0))
        (|incLude| eb (cdr ss) lno ufos (cdr states))))))
(t (cons (|xlCmdBug| eb str lno ufos) |StreamNil|))))))

```

5.3.113 defun xlPrematureEOF

[xlMsg p87]

[inclmsgPrematureEOF p89]

— defun xlPrematureEOF —

```

(defun |xlPrematureEOF| (eb str lno ufos)
  (|xlMsg| eb str lno (elt ufos 0)
    (list (|inclmsgPrematureEOF| (elt ufos 0)) '|error|)))

```

5.3.114 defun xlMsg

[incLine p89]

— defun xlMsg —

```

(defun |xlMsg| (extrablanks string localnum fileobj mess)
  (let ((globalnum -1))
    (list (incLine extrablanks string globalnum localnum fileobj) mess)))

```

5.3.115 defun xlOK

[lxOK1 p??]

— defun xlOK —

```

(defun |xlOK| (extrablanks string localnum fileobj)
  (|xlOK1| extrablanks string string localnum fileobj))

```


5.3.116 defun xLOK1

[incLine1 p89]

— **defun xLOK1** —

```
(defun |xLOK1| (extrablanks string string1 localnum fileobj)
  (let ((globalnum -1))
    (list (incLine1 extrablanks string string1 globalnum localnum fileobj)
          (list nil '|none|))))
```

5.3.117 defun incAppend

[Delay p105]

[incAppend1 p88]

— **defun incAppend** —

```
(defun |incAppend| (x y)
  (|Delay| #'|incAppend1| (list x y)))
```

5.3.118 defun incAppend1

[StreamNull p335]

[incAppend p88]

— **defun incAppend1** —

```
(defun |incAppend1| (&rest z)
  (let (y x)
    (setq x (car z))
    (setq y (cadr z))
    (cond
     ((|StreamNull| x)
      (cond ((|StreamNull| y) |StreamNil|) (t y)))
     (t
      (cons (car x) (|incAppend| (cdr x) y))))))
```

5.3.119 defun incLine

[incLine1 p89]

— defun incLine —

```
(defun incLine (extrablanks string globalnum localnum fileobj)
  (incLine1 extrablanks string string globalnum localnum fileobj))
```

5.3.120 defun incLine1

[lnCreate p347]

— defun incLine1 —

```
(defun incLine1 (extrablanks string string1 globalnum localnum fileobj)
  (cons
    (cons (|lnCreate| extrablanks string globalnum localnum fileobj) 1) string1))
```

5.3.121 defun inclmsgPrematureEOF

[origin p??]

— defun inclmsgPrematureEOF 0 —

```
(defun |inclmsgPrematureEOF| (ufo)
  (list 'S2CI0002 (list (|theorigin| ufo))))
```

5.3.122 defun theorigin

— defun theorigin 0 —

```
(defun |theorigin| (x) (list #'|porigin| x))
```

5.3.123 defun porigin

```
[stringp p??]
```

— defun porigin —

```
(defun |porigin| (x)
  (if (stringp x)
      x
      (|pfname| x)))
```

5.3.124 defun ifCond

```
[MakeSymbol p??]
[incCommandTail p102]
[$inclAssertions p??]
```

— defun ifCond —

```
(defun |ifCond| (s info)
  (let (word)
    (declare (special |$inclAssertions|))
    (setq word
      (|MakeSymbol| (string-trim *whitespace* (|incCommandTail| s info))))
    (member word |$inclAssertions|)))
```

5.3.125 defun xlSkip

```
[incLine p89]
[CONCAT p??]
```

— defun xlSkip —

```
(defun |xlSkip| (extrablanks str localnum fileobj)
```

```
(let ((string (concat "-- Omitting:" str)) (globalnum -1))
(list
 (inclLine extrablanks string globalnum localnum fileobj)
 (list nil '|none|))))
```

5.3.126 defun xlSay

[xlMsg p87]
[inclmsgSay p91]

— defun xlSay —

```
(defun |xlSay| (eb str lno ufos x)
 (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgSay| x) '|say|)))
```

5.3.127 defun inclmsgSay

[id p??]

— defun inclmsgSay —

```
(defun |inclmsgSay| (str)
 (list 'S2CI0001 (list (|theid| str))))
```

5.3.128 defun theid

— defun theid 0 —

```
(defun |theid| (a) (list identity a))
```

5.3.129 defun xlNoSuchFile

[xlMsg p87]

[inclmsgNoSuchFile p92]

— defun xlNoSuchFile —

```
(defun |xlNoSuchFile| (eb str lno ufos fn)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgNoSuchFile| fn) '|error|)))
```

—

5.3.130 defun inclmsgNoSuchFile

[thefname p92]

— defun inclmsgNoSuchFile —

```
(defun |inclmsgNoSuchFile| (fn)
  (list 'S2CI0010 (list (|thefname| fn))))
```

—

5.3.131 defun thefname

[pfname p92]

— defun thefname 0 —

```
(defun |thefname| (x) (list #'|pfname| x))
```

—

5.3.132 defun pfname

[PathnameString p??]

— defun pfname —

```
(defun |pfname| (x) (|PathnameString| x))
```

—

5.3.133 defun xlCannotRead

[xlMsg p87]

[inclmsgCannotRead p93]

— defun xlCannotRead —

```
(defun |xlCannotRead| (eb str lno ufos fn)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgCannotRead| fn) '|error|)))
```

—————

5.3.134 defun inclmsgCannotRead

[thefname p92]

— defun inclmsgCannotRead —

```
(defun |inclmsgCannotRead| (fn)
  (list 'S2CI0011 (list (|thefname| fn))))
```

—————

5.3.135 defun xlFileCycle

[xlMsg p87]

[inclmsgFileCycle p93]

— defun xlFileCycle —

```
(defun |xlFileCycle| (eb str lno ufos fn)
  (|xlMsg| eb str lno (elt ufos 0)
    (list (|inclmsgFileCycle| ufos fn) '|error|)))
```

—————

5.3.136 defun inclmsgFileCycle

```
;inclmsgFileCycle(ufos,fn) ==
;   flist := [porigin n for n in reverse ufos]
;   f1    := porigin fn
;   cycle := [:[:[n,'==>"] for n in flist], f1]
;   ['S2CI0004, [%id cycle, %id f1] ]
```

```
[porigin p90]
[id p??]
```

— defun inclmsgFileCycle —

```
(defun |inclmsgFileCycle| (ufos fn)
  (let (cycle f1 flist)
    (setq flist
      ((lambda (Var8 Var7 n)
        (loop
          (cond
            ((or (atom Var7) (progn (setq n (car Var7)) nil))
              (return (nreverse Var8)))
            (t
              (setq Var8 (cons (|porigin| n) Var8))))
          (setq Var7 (cdr Var7))))
      nil (reverse ufos) nil))
    (setq f1 (|porigin| fn))
    (setq cycle
      (append
        ((lambda (Var10 Var9 n)
          (loop
            (cond
              ((or (atom Var9) (progn (setq n (car Var9)) nil))
                (return (nreverse Var10)))
              (t
                (setq Var10 (append (reverse (list n "==">)) Var10)))
              (setq Var9 (cdr Var9))))
          nil flist nil)
        (cons f1 nil)))
      (list 'S2CI0004 (list (|theid| cycle) (|theid| f1)))))
```

—————

5.3.137 defun xlConActive

```
[xlMsg p87]
[inclmsgConActive p95]
```

— defun xlConActive —

```
(defun |xlConActive| (eb str lno ufos n)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgConActive| n) '|warning|)))
```

—————

5.3.138 defun inclmsgConActive

[id p??]

— defun inclmsgConActive —

```
(defun |inclmsgConActive| (n)
  (list 'S2CI0006 (list (|theid| n))))
```

—————

5.3.139 defun xlConStill

[xlMsg p87]

[inclmsgConStill p95]

— defun xlConStill —

```
(defun |xlConStill| (eb str lno ufos n)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgConStill| n) '|say|)))
```

—————

5.3.140 defun inclmsgConStill

[id p??]

— defun inclmsgConStill —

```
(defun |inclmsgConStill| (n)
  (list 'S2CI0007 (list (|theid| n))))
```

—————

5.3.141 defun xlConsole

[xlMsg p87]

[inclmsgConsole p96]

— defun xlConsole —

```
(defun |xlConsole| (eb str lno ufos)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgConsole|) '|say|)))
```

5.3.142 defun inclmsgConsole

— defun inclmsgConsole 0 —

```
(defun |inclmsgConsole| ()
  (list 'S2CI0005 nil))
```

5.3.143 defun xlSkippingFin

[xlMsg p87]
[inclmsgFinSkipped p96]

— defun xlSkippingFin —

```
(defun |xlSkippingFin| (eb str lno ufos)
  (|xlMsg| eb str lno (elt ufos 0)
    (list (|inclmsgFinSkipped|) '|warning|)))
```

5.3.144 defun inclmsgFinSkipped

— defun inclmsgFinSkipped 0 —

```
(defun |inclmsgFinSkipped| ()
  (list 'S2CI0008 nil))
```

5.3.145 defun xlPrematureFin

[xlMsg p87]
[inclmsgPrematureFin p97]

— defun xlPrematureFin —

```
(defun |xlPrematureFin| (eb str lno ufos)
  (|xlMsg| eb str lno (elt ufos 0)
    (list (|inclmsgPrematureFin| (elt ufos 0)) '|error|)))
```

5.3.146 defun inclmsgPrematureFin

[origin p??]

— defun inclmsgPrematureFin —

```
(defun |inclmsgPrematureFin| (ufo)
  (list 'S2CI0003 (list (|theorigin| ufo))))
```

5.3.147 defun assertCond

[MakeSymbol p??]
 [incCommandTail p102]
 [\$inclAssertions p??]
 [*whitespace* p24]

— defun assertCond —

```
(defun |assertCond| (s info)
  (let (word)
    (declare (special |$inclAssertions| *whitespace*))
    (setq word
      (|MakeSymbol| (string-trim *whitespace* (|incCommandTail| s info))))
    (unless (member word |$inclAssertions|)
      (setq |$inclAssertions| (cons word |$inclAssertions|)))))
```

5.3.148 defun xIfSyntax

[Top? p80]
 [Else? p81]
 [xlMsg p87]
 [inclmsgIfSyntax p98]

— defun xIfSyntax —

```
(defun |xIfSyntax| (eb str lno ufos info sts)
  (let (context found st)
    (setq st (elt sts 0))
    (setq found (elt info 2))
    (setq context
      (cond
        ((|Top?| st) '|not in an )if...endif|)
        ((|Else?| st) '|after an )else|)
        (t '|but can't figure out where|)))
    (|xlMsg| eb str lno (elt ufos 0)
      (list (|inclMsgIfSyntax| (elt ufos 0) found context) '|error|))))
```

—————

5.3.149 defun inclMsgIfSyntax

```
[concat p1003]
[id p??]
[origin p??]
```

— defun inclMsgIfSyntax —

```
(defun |inclMsgIfSyntax| (ufo found context)
  (setq found (concat ")" found))
  (list 'S2CI0009 (list (|theid| found)
                        (|theid| context)
                        (|theorigin| ufo))))
```

—————

5.3.150 defun xIfBug

```
[xlMsg p87]
[inclMsgIfBug p99]
```

— defun xIfBug —

```
(defun |xIfBug| (eb str lno ufos)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclMsgIfBug|) '|bug|)))
```

—————

5.3.151 defun inclmsgIfBug

— defun inclmsgIfBug 0 —

```
(defun |inclmsgIfBug| ()
  (list 'S2CB0002 nil))
```

—————

5.3.152 defun xlCmdBug

```
[xlMsg p87]
[inclmsgCmdBug p99]
```

— defun xlCmdBug —

```
(defun |xlCmdBug| (eb str lno ufos)
  (|xlMsg| eb str lno (elt ufos 0) (list (|inclmsgCmdBug|) '|bug|)))
```

—————

5.3.153 defun inclmsgCmdBug

— defun inclmsgCmdBug 0 —

```
(defun |inclmsgCmdBug| ()
  (list 'S2CB0003 nil))
```

—————

5.3.154 defvar \$incCommands

This is a list of commands that can be in an include file

— postvars —

```
(eval-when (eval load)
  (setq |incCommands|
    (list "say" "include" "console" "fin" "assert" "if" "elseif" "else" "endif")))
```

—————

5.3.155 defvar \$pfMacros

The \$pfMacros variable is an alist [[id, state, body-pform], ...] where state is one of: mbody, mparam, mlambda

User-defined macros are maintained in a stack of definitions. This is the stack sequence resulting from the command lines:

```
a ==> 3
a ==> 4
b ==> 7
(
  (|b| |mbody| ((|integer| (|posn| (0 "b ==> 7" 1 1 "strings") . 6)) . "7"))
  (|a| |mbody| ((|integer| (|posn| (0 "a ==> 4" 1 1 "strings") . 6)) . "4"))
  (|a| |mbody| ((|integer| (|posn| (0 "a ==> 3" 1 1 "strings") . 6)) . "3"))
)
```

— initvars —

```
(defvar |$pfMacros| nil)
```

5.3.156 defun incClassify

```
;incClassify(s) ==
;      not incCommand? s => [false,0, '"]
;      i := 1; n := #s
;      while i < n and s.i = char " " repeat i := i + 1
;      i >= n => [true,0,'other"]
;      eb := (i = 1 => 0; i)
;      bad:=true
;      for p in incCommands while bad repeat
;          incPrefix?(p, i, s) =>
;              bad:=false
;              p1 :=p
;          if bad then [true,0,'other"] else [true,eb,p1]
```

```
[incCommand? p101]
```

```
[incCommands p99]
```

— defun incClassify —

```
(defun |incClassify| (s)
  (let (p1 bad eb n i)
    (declare (special |incCommands|)))
```

```

(if (null (|incCommand?| s))
  (list nil 0 ""))
(progn
  (setq i 1)
  (setq n (length s))
  ((lambda ()
    (loop
      (cond
        ((not (and (< i n) (char= (elt s i) #\space)))
         (return nil))
        (t (setq i (1+ i)))))))
  (cond
    ((not (< i n)) (list t 0 "other"))
    (t
     (if (= i 1)
         (setq eb 0)
         (setq eb i))
     (setq bad t)
     ((lambda (tmp1 p)
        (loop
          (cond
            ((or (atom tmp1)
                 (progn (setq p (car tmp1)) nil)
                 (not bad))
             (return nil))
            (t
             (cond
               ((|incPrefix?| p i s)
                (identity
                 (progn
                  (setq bad nil)
                  (setq p1 p))))))
            (setq tmp1 (cdr tmp1))))
         |incCommands| nil)
     (if bad
         (list t 0 "other")
         (list t eb p1))))))

```

5.3.157 defun incCommand?

[char p??]

— defun incCommand? 0 —

```

(defun |incCommand?| (s)
  "does this start with a close paren?"

```

```
(and (< 0 (length s)) (equal (elt s 0) (|char| '|)|))))
```

5.3.158 defun incPrefix?

```
;incPrefix?(prefix, start, whole) ==
;      #prefix > #whole-start => false
;      good:=true
;      for i in 0..#prefix-1 for j in start.. while good repeat
;          good:= prefix.i = whole.j
;      good
```

— defun incPrefix? 0 —

```
(defun |incPrefix?| (prefix start whole)
  (let (good)
    (cond
      ((< (- (length whole) start) (length prefix)) nil)
      (t
       (setq good t)
       ((lambda (Var i j)
          (loop
            (cond
              ((or (> i Var) (not good)) (return nil))
              (t (setq good (equal (elt prefix i) (elt whole j))))
            (setq i (+ i 1))
            (setq j (+ j 1))))
          (- (length prefix) 1) 0 start)
       good))))
```

5.3.159 defun incCommandTail

[incDrop p103]

— defun incCommandTail —

```
(defun |incCommandTail| (s info)
  (let ((start (elt info 1)))
    (when (= start 0) (setq start 1))
    (|incDrop| (+ start (length (elt info 2)) 1) s)))
```

5.3.160 defun incDrop

[substring p??]

— defun incDrop 0 —

```
(defun |incDrop| (n b)
  (if (>= n (length b))
      '||
      (substring b n nil)))
```

5.3.161 defun inclFname

[incFileName p600]

[incCommandTail p102]

— defun inclFname —

```
(defun |inclFname| (s info)
  (|incFileName| (|incCommandTail| s info)))
```

5.3.162 defun incFileInput

[incRgen p104]

[make-instream p937]

— defun incFileInput —

```
(defun |incFileInput| (fn)
  (|incRgen| (make-instream fn)))
```

5.3.163 defun incConsoleInput

[incRgen p104]

[make-instream p937]

— defun incConsoleInput —


```
(defun |incConsoleInput| ()
  (|incRgen| (make-instream 0)))
```

5.3.164 defun incNConsoles

[incNConsoles p104]

— defun incNConsoles —

```
(defun |incNConsoles| (ufos)
  (let ((a (member "console" ufos)))
    (if a
      (+ 1 (|incNConsoles| (cdr a)))
      0)))
```

5.3.165 defun incActive?

— defun incActive? 0 —

```
(defun |incActive?| (fn ufos)
  (member fn ufos))
```

5.3.166 defun incRgen

Note that incRgen1 recursively calls this function. [Delay p105]
[incRgen1 p105]

— defun incRgen —

```
(defun |incRgen| (s)
  (|Delay| #'|incRgen1| (list s)))
```

5.3.167 defun Delay

— defun Delay 0 —

```
(defun |Delay| (f x)
  (cons '|nonnullstream| (cons f x)))
```

—————

5.3.168 defvar \$StreamNil

— initvars —

```
(defvar |StreamNil| (list '|nullstream|))
```

—————

5.3.169 defvar \$StreamNil

— postvars —

```
(eval-when (eval load)
  (setq |StreamNil| (list '|nullstream|)))
```

—————

5.3.170 defun incRgen1

This function reads a line from the stream and then conses it up with a recursive call to incRgen. Note that incRgen recursively wraps this function in a delay list. [incRgen p104] [StreamNil p105]

— defun incRgen1 —

```
(defun |incRgen1| (&rest z)
  (let (a s)
    (declare (special |StreamNil|))
    (setq s (car z))
    (setq a (read-line s nil nil)))
```

```
(if (null a)
  (progn
    (close s)
    |StreamNil|)
  (cons a (|incRgen| s))))
```

Chapter 6

The Token Scanner

6.0.171 defvar \$space

— postvars —

```
(eval-when (eval load)
  (defvar space (qenum " " 0)))
```

6.0.172 defvar \$escape

— postvars —

```
(eval-when (eval load)
  (defvar escape (qenum "_" 0)))
```

6.0.173 defvar \$stringchar

— postvars —

```
(eval-when (eval load)
  (defvar stringchar (qenum "\" 0)))
```

6.0.174 defvar \$pluscomment

— postvars —

```
(eval-when (eval load)
(defvar pluscomment (qenum "+ " 0)))
```

6.0.175 defvar \$minuscomment

— postvars —

```
(eval-when (eval load)
(defvar minuscomment (qenum "- " 0)))
```

6.0.176 defvar \$radixchar

— postvars —

```
(eval-when (eval load)
(defvar radixchar (qenum "r " 0)))
```

6.0.177 defvar \$dot

— postvars —

```
(eval-when (eval load)
(defvar dot (qenum "." 0)))
```

6.0.178 defvar \$exponent1

— postvars —

```
(eval-when (eval load)
  (defvar exponent1 (qenum "E" 0)))
```

—————

6.0.179 defvar \$exponent2

— postvars —

```
(eval-when (eval load)
  (defvar exponent2 (qenum "e" 0)))
```

—————

6.0.180 defvar \$closeparen

— postvars —

```
(eval-when (eval load)
  (defvar closeparen (qenum ")" 0)))
```

—————

6.0.181 defvar \$closeangle

— postvars —

```
(eval-when (eval load)
  (defvar closeangle (qenum ">" 0)))
```

—————

6.0.182 defvar \$question

— postvars —

```
(eval-when (eval load)
(defvar question (qenum "? " 0)))
```

—————

6.0.183 defvar \$scanKeyWords

— postvars —

```
(eval-when (eval load)
(defvar |scanKeyWords|
(list
(list "add" 'add)
(list "and" 'and)
(list "break" 'break)
(list "by" 'by)
(list "case" 'case)
(list "default" 'default)
(list "define" 'defn)
(list "do" 'do)
(list "else" 'else)
(list "exit" 'exit)
(list "export" 'export)
(list "for" 'for)
(list "free" 'free)
(list "from" 'from)
(list "has" 'has)
(list "if" 'if)
(list "import" 'import)
(list "in" 'in)
(list "inline" 'inline)
(list "is" 'is)
(list "isnt" 'isnt)
(list "iterate" 'iterate)
(list "local" '|local|)
(list "macro" 'macro)
(list "mod" 'mod)
(list "or" 'or)
(list "pretend" 'pretend)
(list "quo" 'quo)
(list "rem" 'rem)
```

```

(list "repeat" 'repeat)
(list "return" 'return)
(list "rule" 'rule)
(list "then" 'then)
(list "where" 'where)
(list "while" 'while)
(list "with" 'with)
(list "|" 'bar)
(list "." 'dot)
(list "::" 'coerce)
(list ":" 'colon)
(list ":-" 'colondash)
(list "@" 'at)
(list "@@" 'atat)
(list "," 'comma)
(list ";" 'semicolon)
(list "**" 'power)
(list "*" 'times)
(list "+" 'plus)
(list "-" 'minus)
(list "<" 'lt)
(list ">" 'gt)
(list "<=" 'le)
(list ">=" 'ge)
(list "=" 'equal)
(list "~=" 'notequal)
(list "~" '~)
(list "^" 'carat)
(list ".." 'seg)
(list "#" '|#|)
(list "&" 'ampersand)
(list "$" '$)
(list "/" 'slash)
(list "\" 'backslash)
(list "//" 'slashslash)
(list "\\\" 'backslashbackslash)
(list "/\" 'slashbackslash)
(list "\\/" 'backslashslash)
(list "=>" 'exit)
(list ":=" 'becomes)
(list "==" 'def)
(list "==" 'mdef)
(list "->" 'arrow)
(list "<-" 'larrow)
(list "+->" 'gives)
(list "(" '|(|)
(list ")" '|)|)
(list "|" '|(|)
(list "|)" '|)|)
(list "[" '[]

```



```

(list "]" '])
(list "[" '[])
(list "{" '{})
(list "}" '})
(list "{_}" '{})
(list "[" '["\|])
(list "]" ']\|])
(list "[_]" '[_\|])
(list "{" '{\|])
(list "}" '{\|])
(list "{_}" '{\|])
(list "<<" 'oangle)
(list ">>" 'cangle)
(list "\"" '")
(list "`" 'backquote)))

```

—

6.0.184 defvar \$infgeneric

— postvars —

```

(eval-when (eval load)
(prog ()
  (return
    ((lambda (var value)
      (loop
        (cond
          ((or (atom var) (progn (setq value (car var)) nil))
           (return nil))
          (t
           (setf (get (car value) 'infgeneric) (cadr value))))
        (setq var (cdr var))))
    (list
      (list 'equal '=)
      (list 'times '*)
      (list 'has '|has|)
      (list 'case '|case|)
      (list 'rem '|rem|)
      (list 'mod '|mod|)
      (list 'quo '|quo|)
      (list 'slash '/')
      (list 'backslash '\\)
      (list 'slashslash '//)
      (list 'backslashbackslash '\\\\)
      (list 'slashbackslash '|/\\|)
      (list 'backslashslash '\\/|)

```

```

(list 'power '**)
(list 'carat '^)
(list 'plus '+)
(list 'minus '-')
(list 'lt '<)
(list 'gt '>)
(list 'oangle '<<)
(list 'cangle '>>)
(list 'le '<=)
(list 'ge '>=)
(list 'notequal '~=)
(list 'by '|by|)
(list 'arrow '->)
(list 'larrow '<-)
(list 'bar '|\\|)
(list 'seg '|..|)
nil)))

```

6.0.185 defun lineoftoks

lineoftoks bites off a token-dq from a line-stream returning the token-dq and the rest of the line-stream

```

;lineoftoks(s)==
;  $f: local:=nil
;  $r:local :=nil
;  $ln:local :=nil
;  $linepos:local:=nil
;  $n:local:=nil
;  $sz:local := nil
;  $floatok:local:=true
;  if not nextline s
;  then CONS(nil,nil)
;  else
;    if null scanIgnoreLine($ln,$n) -- line of spaces or starts ) or >
;    then cons(nil,$r)
;    else
;      toks:=[]
;      a:= incPrefix?('command",1,$ln)
;      a =>
;          $ln:=SUBSTRING($ln,8,nil)
;          b:= dqUnit constoken($ln,$linepos,["command",$ln],0)
;          cons([ [b,s] ],$r)
;
;      while $n<$sz repeat toks:=dqAppend(toks,scanToken())
;      if null toks

```

```
;      then cons([], $r)
;      else cons([ [toks,s] ], $r)
```

```
[nextline p115]
[scanIgnoreLine p115]
[incPrefix? p102]
[substring p??]
[dqUnit p345]
[constoken p116]
[$floatok p??]
[$f p??]
[$sz p??]
[$linepos p??]
[$r p??]
[$n p??]
[$ln p??]
```

— defun lineoftoks —

```
(defun |lineoftoks| (s)
  (let (|$floatok| |$sz| |$n| |$linepos| |$ln| |$r| |$f| |b| |a| |toks|)
    (declare (special |$floatok| |$f| |$sz| |$linepos| |$r| |$n| |$ln|))
    (setq |$f| nil)
    (setq |$r| nil)
    (setq |$ln| nil)
    (setq |$linepos| nil)
    (setq |$n| nil)
    (setq |$sz| nil)
    (setq |$floatok| t)
    (cond
      ((null (|nextline| s)) (cons nil nil))
      ((null (|scanIgnoreLine| |$ln| |$n|)) (cons nil |$r|))
      (t
       (setq |toks| nil)
       (setq |a| (|incPrefix?| "command" 1 |$ln|))
       (cond
         (|a|
          (setq |$ln| (substring |$ln| 8 nil))
          (setq |b|
            (|dqUnit| (|constoken| |$ln| |$linepos| (list '|command| |$ln|) 0)))
          (cons (list (list |b| s)) |$r|))
         (t
          ((lambda ()
              (loop
                (cond
                  ((not (< |$n| |$sz|)) (return nil))
                  (t (setq |toks| (|dqAppend| |toks| (|scanToken|))))))))
          (cond
            ((null |toks|) (cons nil |$r|))
```

```
(t (cons (list (list |toks| s)) |$r|)))))))))
```

6.0.186 defun nextline

[npNull p335]
 [strpos1 p1002]
 [\$sz p??]
 [\$n p??]
 [\$linepos p??]
 [\$ln p??]
 [\$r p??]
 [\$f p??]

— defun nextline —

```
(defun |nextline| (s)
  (declare (special |$sz| |$n| |$linepos| |$ln| |$r| |$f|))
  (cond
    ((|npNull| s) nil)
    (t
     (setq |$f| (car s))
     (setq |$r| (cdr s))
     (setq |$ln| (cdr |$f|))
     (setq |$linepos| (caar |$f|))
     (setq |$n| (strpos1 " " |$ln| 0 t)) ; spaces at beginning
     (setq |$sz| (length |$ln|))
     t)))
```

6.0.187 defun scanIgnoreLine

[qenum p1002]
 [incPrefix? p102]

— defun scanIgnoreLine —

```
(defun |scanIgnoreLine| (ln n)
  (let (fst)
    (cond
      ((null n) n)
      (t
```

```

(setq fst (qenum ln 0))
(cond
  ((eq fst closeparen)
    (cond
      ((|incPrefix?| "command" 1 ln) t)
      (t nil)))
  (t n))))))

```

6.0.188 defun constoken

[ncPutQ p418]

— **defun constoken** —

```

(defun |constoken| (ln lp b n)
  (declare (ignore ln))
  (let (a)
    (setq a (cons (elt b 0) (elt b 1)))
    (|ncPutQ| a '|posn| (cons lp n))
    a))

```

6.0.189 defun scanToken

[qenum p1002]
 [startsComment? p118]
 [scanComment p118]
 [startsNegComment? p119]
 [scanNegComment p119]
 [lfd p117]
 [punctuation? p120]
 [scanPunct p120]
 [startsId? p1000]
 [scanWord p128]
 [scanSpace p131]
 [scanString p132]
 [digit? p124]
 [scanNumber p134]
 [scanEscape p137]
 [scanError p137]
 [dqUnit p345]

```
[constoken p116]
[lnExtraBlanks p347]
[$linepos p??]
[$n p??]
[$ln p??]
```

— **defun scanToken** —

```
(defun |scanToken| ()
  (let (b ch n linepos c ln)
    (declare (special |$linepos| |$n| |$ln|))
    (setq ln |$ln|)
    (setq c (qenum |$ln| |$n|))
    (setq linepos |$linepos|)
    (setq n |$n|)
    (setq ch (elt |$ln| |$n|))
    (setq b
      (cond
        ((|startsComment?|) (|scanComment|) nil)
        ((|startsNegComment?|) (|scanNegComment|) nil)
        ((equal c question)
         (setq |$n| (+ |$n| 1))
         (|lfid| "?"))
        ((|punctuation?| c) (|scanPunct|))
        ((|startsId?| ch) (|scanWord| nil))
        ((equal c space) (|scanSpace|) nil)
        ((equal c stringchar) (|scanString|))
        ((|digit?| ch) (|scanNumber|))
        ((equal c escape) (|scanEscape|))
        (t (|scanError|))))
    (cond
      ((null b) nil)
      (t
       (|dqUnit|
        (|constoken| ln linepos b (+ n (|lnExtraBlanks| linepos))))))))
```

—

6.0.190 defun lfid

To pair badge and badgee

— **defun lfid 0** —

```
(defun |lfid| (x)
  (list '|id| (intern x "BOOT")))
```

—

6.0.191 defun startsComment?

```
[qenum p1002]
[$ln p??]
[$sz p??]
[$n p??]
[pluscomment p108]
```

— **defun startsComment?** —

```
(defun |startsComment?| ()
  (let (www)
    (declare (special |$ln| |$sz| |$n| pluscomment))
    (cond
      ((< |$n| |$sz|)
        (cond
          ((equal (qenum |$ln| |$n|) pluscomment)
            (setq www (+ |$n| 1))
            (cond
              ((not (< www |$sz|)) nil)
              (t (equal (qenum |$ln| www) pluscomment))))
          (t nil)))
      (t nil))))
```

6.0.192 defun scanComment

```
[lfcomment p119]
[substring p??]
[$ln p??]
[$sz p??]
[$n p??]
```

— **defun scanComment** —

```
(defun |scanComment| ()
  (let (n)
    (declare (special |$ln| |$sz| |$n|))
    (setq n |$n|)
    (setq |$n| |$sz|)
    (|lfcomment| (substring |$ln| n nil))))
```

6.0.193 defun lfcomment

— defun lfcomment 0 —

```
(defun |lfcomment| (x)
  (list '|comment| x))
```

—————

6.0.194 defun startsNegComment?

```
[qenum p1002]
[$ln p??]
[$sz p??]
[$n p??]
```

— defun startsNegComment? —

```
(defun |startsNegComment?| ()
  (let (www)
    (declare (special |$ln| |$sz| |$n|))
    (cond
      ((< |$n| |$sz|)
        (cond
          ((equal (qenum |$ln| |$n|) minuscomment)
            (setq www (+ |$n| 1))
            (cond
              ((not (< www |$sz|)) nil)
              (t (equal (qenum |$ln| www) minuscomment))))
          (t nil)))
      (t nil))))
```

—————

6.0.195 defun scanNegComment

```
[lfnegcomment p120]
[substring p??]
[$ln p??]
[$sz p??]
[$n p??]
```

— defun scanNegComment —


```
(defun |scanNegComment| ()
  (let (n)
    (declare (special |$ln| |$sz| |$n|))
    (setq n |$n|)
    (setq |$n| |$sz|)
    (|lfnegcomment| (substring |$ln| n nil))))
```

6.0.196 defun lfnegcomment

— defun lfnegcomment 0 —

```
(defun |lfnegcomment| (x)
  (list '|negcomment| x))
```

6.0.197 defun punctuation?

— defun punctuation? —

```
(defun |punctuation?| (c)
  (eq1 (elt |scanPun| c) 1))
```

6.0.198 defun scanPunct

```
[subMatch p121]
[scanError p137]
[scanKeyTr p122]
[$n p??]
[$ln p??]
```

— defun scanPunct —

```
(defun |scanPunct| ()
  (let (a sss)
    (declare (special |$n| |$ln|))
```

```

(setq sss (|subMatch| |$ln| |$n|))
(setq a (length sss))
(cond
  ((eql a 0) (|scanError|))
  (t (setq |$n| (+ |$n| a)) (|scanKeyTr| sss))))

```

6.0.199 defun subMatch

[substringMatch p121]

— defun subMatch —

```

(defun |subMatch| (a b)
  (|substringMatch| a |scanDict| b))

```

6.0.200 defun substringMatch

```

;substringMatch (l,d,i)==
;      h:= QENUM(l, i)
;      u:=ELT(d,h)
;      ll:=SIZE l
;      done:=false
;      s1:=""
;      for j in 0.. SIZE u - 1 while not done repeat
;        s:=ELT(u,j)
;        ls:=SIZE s
;        done:=if ls+i > ll
;              then false
;              else
;                eql:= true
;                for k in 1..ls-1 while eql repeat
;                  eql:= EQL(QENUM(s,k),QENUM(l,k+i))
;                if eql
;                then
;                  s1:=s
;                  true
;                else false
;      s1

```

[qenum p1002]

[size p1001]

— defun substringMatch —

```
(defun |substringMatch| (l dict i)
  (let (equal ls s s1 done ll u h)
    (setq h (qenum l i))
    (setq u (elt dict h))
    (setq ll (size l))
    (setq s1 "")
    ((lambda (Var4 j)
      (loop
        (cond
          ((or (> j Var4) done) (return nil))
          (t
            (setq s (elt u j))
            (setq ls (size s))
            (setq done
              (cond
                ((< ll (+ ls i)) nil)
                (t
                  (setq equal t)
                  ((lambda (Var5 k)
                    (loop
                      (cond
                        ((or (> k Var5) (not equal)) (return nil))
                        (t
                          (setq equal (eq1 (qenum s k) (qenum l (+ k i))))
                          (setq k (+ k 1))))
                        (- ls 1) 1)
                    (cond (equal (setq s1 s) t) (t nil)))))))
                  (setq j (+ j 1))))
            (- (size u) 1) 0)
          s1))
```

—————

6.0.201 defun scanKeyTr

```
[keyword p123]
[scanPossFloat p123]
[lfkey p124]
[scanCloser? p128]
[$floatok p??]
```

— defun scanKeyTr —

```
(defun |scanKeyTr| (w)
```

```
(declare (special |$floatok|))
(cond
  ((eq (|keyword| w) 'dot)
    (cond
      (|$floatok| (|scanPossFloat| w))
      (t (|lfkey| w))))
  (t (setq |$floatok| (null (|scanCloser?| w))) (|lfkey| w))))
```

6.0.202 defun keyword

[hget p1000]

— defun keyword 0 —

```
(defun |keyword| (st)
  (hget |scanKeyTable| st))
```

6.0.203 defun keyword?

[hget p1000]

— defun keyword? 0 —

```
(defun |keyword?| (st)
  (null (null (hget |scanKeyTable| st))))
```

6.0.204 defun scanPossFloat

```
[digit? p124]
[lfkey p124]
[spleI p124]
[scanExponent p128]
[$ln p??]
[$sz p??]
[$n p??]
```

— defun scanPossFloat —

```
(defun |scanPossFloat| (w)
  (declare (special |$ln| |$sz| |$n|))
  (cond
    ((or (not (< |$n| |$sz|)) (null (|digit?| (elt |$ln| |$n|))))
      (|lfkey| w))
    (t
      (setq w (|spleI| #'|digit?|)) (|scanExponent| "0" w))))
```

6.0.205 defun digit?

[digitp p1001]

— defun digit? —

```
(defun |digit?| (x)
  (digitp x))
```

6.0.206 defun lfkey

[keyword p123]

— defun lfkey —

```
(defun |lfkey| (x)
  (list 'key (|keyword| x)))
```

6.0.207 defun spleI

[spleI1 p125]

— defun spleI —

```
(defun |spleI| (dig)
  (|spleI1| dig nil))
```

6.0.208 defun spleI1

```
[qenum p1002]
[substring p??]
[scanEsc p125]
[spleI1 p125]
[concat p1003]
[$ln p??]
[$sz p??]
[$n p??]
```

— defun spleI1 —

```
(defun |spleI1| (dig zro)
  (let (bb a str l n)
    (declare (special |$ln| |$sz| |$n|))
    (setq n |$n|)
    (setq l |$sz|)
    ; while $n<l and FUNCALL(dig,($ln.$n)) repeat $n:=$n+1
    ((lambda ()
      (loop
        (cond
          ((not (and (< |$n| l) (funcall dig (elt |$ln| |$n|))))
            (return nil))
          (t
            (setq |$n| (+ |$n| 1)))))))
    (cond
      ((or (equal |$n| l) (not (equal (qenum |$ln| |$n|) escape)))
        (cond
          ((and (equal n |$n|) zro) "0")
          (t (substring |$ln| n (- |$n| n)))))
      (t
        ; escaped
        (setq str (substring |$ln| n (- |$n| n)))
        (setq |$n| (+ |$n| 1))
        (setq a (|scanEsc|))
        (setq bb (|spleI1| dig zro)) ; escape, any number of spaces are ignored
        (concat str bb)))))
```

—————

6.0.209 defun scanEsc

```
;scanEsc()==
;   if $n>=$sz
;   then if nextline($r)
;       then
;           while null $n repeat nextline($r)
```

```

;      scanEsc()
;      false
;      else false
;      else
;      n1:=STRPOS(' " ', $ln, $n, true)
;      if null n1
;      then if nextline($r)
;      then
;      while null $n repeat nextline($r)
;      scanEsc()
;      false
;      else false
;      else
;      if $n=n1
;      then true
;      else if QENUM($ln, n1)=ESCAPE
;      then
;      $n:=n1+1
;      scanEsc()
;      false
;      else
;      $n:=n1
;      startsNegComment?() or startsComment?() =>
;      nextline($r)
;      scanEsc()
;      false
;      false

```

```

[nextline p115]
[scanEsc p125]
[strposl p1002]
[qenum p1002]
[startsNegComment? p119]
[startsComment? p118]
[$ln p??]
[$r p??]
[$sz p??]
[$n p??]

```

— defun scanEsc —

```

(defun |scanEsc| ()
  (let (n1)
    (declare (special |$ln| |$r| |$sz| |$n|))
    (cond
      ((not (< |$n| |$sz|))
        (cond
          ((|nextline| |$r|)
            ((lambda ()

```

```

(loop
  (cond
    (|$n| (return nil))
    (t (|nextline| |$r|))))))
(|scanEsc|)
nil)
(t nil)))
(t
  (setq n1 (strpos1 " " |$ln| |$n| t))
  (cond
    ((null n1)
     (cond
       ((|nextline| |$r|)
        (lambda ()
          (loop
            (cond
              (|$n| (return nil))
              (t (|nextline| |$r|))))))
        (|scanEsc|)
        nil)
        (t nil)))
    ((equal |$n| n1) t)
    ((equal (qenum |$ln| n1) escape)
     (setq |$n| (+ n1 1))
     (|scanEsc|)
     nil)
    (t (setq |$n| n1)
        (cond
          ((or (|startsNegComment?|) (|startsComment?|))
           (progn
            (|nextline| |$r|)
            (|scanEsc|)
            nil))
          (t nil)))))))))

```

6.0.210 defvar \$scanCloser

— postvars —

```

(eval-when (eval load)
  (defvar |scanCloser| (list '|'| '}' ']' '|\\'| '|\\}| '|\\||'))

```

6.0.211 defun scanCloser?

```
[keyword p123]
[scanCloser p127]

— defun scanCloser? 0 —

(defun |scanCloser?| (w)
  (declare (special |scanCloser|))
  (member (|keyword| w) |scanCloser|))
```

6.0.212 defun scanWord

```
[scanW p130]
[lfid p117]
[keyword? p123]
[lfkey p124]
[$floatok p??]

— defun scanWord —

(defun |scanWord| (esp)
  (let (w aaa)
    (declare (special |$floatok|))
    (setq aaa (|scanW| nil))
    (setq w (elt aaa 1))
    (setq |$floatok| nil)
    (cond
      ((or esp (elt aaa 0))
       (|lfid| w))
      ((|keyword?| w)
       (setq |$floatok| t)
       (|lfkey| w))
      (t
       (|lfid| w)))))
```

6.0.213 defun scanExponent

```
[lffloat p130]
[qenum p1002]
```

```
[digit? p124]
[spleI p124]
[concat p1003]
[$ln p??]
[$sz p??]
[$n p??]
```

— **defun scanExponent** —

```
(defun |scanExponent| (a w)
  (let (c1 e c n)
    (declare (special |$ln| |$sz| |$n|))
    (cond
      ((not (< |$n| |$sz|)) (|lffloat| a w "0"))
      (t
       (setq n |$n|)
       (setq c (qenum |$ln| |$n|))
       (cond
         ((or (equal c exponent1) (equal c exponent2))
          (setq |$n| (+ |$n| 1))
          (cond
            ((not (< |$n| |$sz|))
             (setq |$n| n)
             (|lffloat| a w "0"))
            ((|digit?| (elt |$ln| |$n|))
             (setq e (|spleI| #'|digit?|))
             (|lffloat| a w e))
            (t
             (setq c1 (qenum |$ln| |$n|))
             (cond
               ((or (equal c1 pluscomment) (equal c1 minuscomment))
                (setq |$n| (+ |$n| 1))
                (cond
                  ((not (< |$n| |$sz|))
                   (setq |$n| n)
                   (|lffloat| a w "0"))
                  ((|digit?| (elt |$ln| |$n|))
                   (setq e (|spleI| #'|digit?|))
                   (|lffloat| a w
                     (cond
                       ((equal c1 minuscomment)
                        (concat "-" e))
                       (t e))))
                  (t
                   (setq |$n| n)
                   (|lffloat| a w "0"))))))))
             (t (|lffloat| a w "0"))))))))
```

6.0.214 defun lffloat

[concat p1003]

— defun lffloat 0 —

```
(defun |lffloat| (a w e)
  (list '|float| (concat a "." w "e" e)))
```

6.0.215 defmacro idChar?

— defmacro idChar? 0 —

```
(defmacro |idChar?| (x)
  '(or (alphanumericp ,x) (member ,x '(#\? #\% #\' #\!) :test #'char=)))
```

6.0.216 defun scanW

```
[posend p131]
[qenum p1002]
[substring p??]
[scanEsc p125]
[scanW p130]
[idChar? p130]
[concat p1003]
[$ln p??]
[$sz p??]
[$n p??]
```

— defun scanW —

```
(defun |scanW| (b)
  (let (bb a str endid l n1)
    (declare (special |$ln| |$sz| |$n|))
    (setq n1 |$n|)
    (setq |$n| (+ |$n| 1))
```

```

(setq l |$sz|)
(setq endid (|posend| |$ln| |$n|))
(cond
  ((or (equal endid l) (not (equal (qenum |$ln| endid) escape)))
    (setq |$n| endid)
    (list b (substring |$ln| n1 (- endid n1))))
  (t
    (setq str (substring |$ln| n1 (- endid n1)))
    (setq |$n| (+ endid 1))
    (setq a (|scanEsc|))
    (setq bb
      (cond
        (a (|scanW| t))
        ((not (< |$n| |$sz|)) (list b ""))
        ((|idChar?| (elt |$ln| |$n|)) (|scanW| b))
        (t (list b ""))))
    (list (or (elt bb 0) b) (concat str (elt bb 1))))))

```

6.0.217 defun posend

```

;posend(line,n)==
;   while n<#line and idChar? line.n repeat n:=n+1
;   n

```

NOTE: do not replace “lyne” with “line”

— **defun posend** —

```

(defun |posend| (lyne n)
  ((lambda ()
    (loop
      (cond
        ((not (and (< n (length lyne)) (|idChar?| (elt lyne n))))
          (return nil))
        (t (setq n (+ n 1))))))
    n)

```

6.0.218 defun scanSpace

```

[strposl p1002]
[lfspace p132]
[$floatok p??]

```

```
[$ln p??]
[$n p??]
```

— defun scanSpace —

```
(defun |scanSpace| ()
  (let (n)
    (declare (special |$floatok| |$ln| |$n|))
    (setq n |$n|)
    (setq |$n| (strpos1 " " |$ln| |$n| t))
    (when (null |$n|) (setq |$n| (length |$ln|)))
    (setq |$floatok| t)
    (|lfspaces| (- |$n| n))))
```

—————

6.0.219 defun lfspaces

— defun lfspaces 0 —

```
(defun |lfspaces| (x)
  (list '|spaces| x))
```

—————

6.0.220 defun scanString

```
[lfstring p133]
[scanS p133]
[$floatok p??]
[$n p??]
```

— defun scanString —

```
(defun |scanString| ()
  (declare (special |$floatok| |$n|))
  (setq |$n| (+ |$n| 1))
  (setq |$floatok| nil)
  (|lfstring| (|scanS|)))
```

—————

6.0.221 defun lfstring

— defun lfstring 0 —

```
(defun |lfstring| (x)
  (if (eql (length x) 1)
      (list '|char| x)
      (list '|string| x)))
```

—

6.0.222 defun scanS

```
[ncSoftError p353]
[lnExtraBlanks p347]
[strpos p1002]
[substring p??]
[scanEsc p125]
[concat p1003]
[scanTransform p134]
[scanS p133]
[$ln p??]
[$linepos p??]
[$sz p??]
[$n p??]
```

— defun scanS —

```
(defun |scanS| ()
  (let (b a str mn escsym strsym n)
    (declare (special |$ln| |$linepos| |$sz| |$n|))
    (cond
      ((not (< |$n| |$sz|))
       (|ncSoftError|
        (cons |$linepos| (+ (|lnExtraBlanks| |$linepos|) |$n|)) 'S2CN0001 nil) ""))
      (t
       (setq n |$n|)
       (setq strsym (or (strpos "\" |$ln| |$n| nil) |$sz|))
       (setq escsym (or (strpos "_" |$ln| |$n| nil) |$sz|))
       (setq mn (min strsym escsym))
       (cond
         ((equal mn |$sz|)
          (setq |$n| |$sz|)
          (|ncSoftError|
           (cons |$linepos| (+ (|lnExtraBlanks| |$linepos|) |$n|)) 'S2CN0001 nil))
```

```

      (substring |$ln| n nil))
    ((equal mn strsym)
     (setq |$n| (+ mn 1))
     (substring |$ln| n (- mn n)))
    (t
     (setq str (substring |$ln| n (- mn n)))
     (setq |$n| (+ mn 1))
     (setq a (|scanEsc|))
     (setq b
      (cond
       (a
        (setq str (concat str (|scanTransform| (elt |$ln| |$n|))))
        (setq |$n| (+ |$n| 1)) (|scanS|))
       (t (|scanS|))))
     (concat str b))))))

```

6.0.223 defun scanTransform

— defun scanTransform —

```
(defun |scanTransform| (x) x)
```

6.0.224 defun scanNumber

```

[spleI p124]
[linteger p136]
[qenum p1002]
[spleI1 p125]
[scanExponent p128]
[scanCheckRadix p136]
[lfrinteger p136]
[concat p1003]
[$floatok p??]
[$ln p??]
[$sz p??]
[$n p??]

```

— defun scanNumber —

```

(defun |scanNumber| ()
  (let (v w n a)
    (declare (special |$floatok| |$ln| |$sz| |$n|))
    (setq a (|spleI| #'|digit?|))
    (cond
      ((not (< |$n| |$sz|))
       (|lfinteger| a))
      ((not (equal (qenum |$ln| |$n|) radixchar))
       (cond
         ((and |$floatok| (equal (qenum |$ln| |$n|) dot))
          (setq n |$n|)
          (setq |$n| (+ |$n| 1))
          (cond
            ((and (< |$n| |$sz|) (equal (qenum |$ln| |$n|) dot))
             (setq |$n| n)
             (|lfinteger| a))
            (t
             (setq w (|spleI1| #'|digit?| t))
             (|scanExponent| a w))))
          (t (|lfinteger| a))))
      (t
       (setq |$n| (+ |$n| 1))
       (setq w (|spleI1| #'|rdigit?| t))
       (|scanCheckRadix| (parse-integer a) w)
       (cond
         ((not (< |$n| |$sz|))
          (|lfrinteger| a w))
         ((equal (qenum |$ln| |$n|) dot)
          (setq n |$n|)
          (setq |$n| (+ |$n| 1))
          (cond
            ((and (< |$n| |$sz|) (equal (qenum |$ln| |$n|) dot))
             (setq |$n| n)
             (|lfrinteger| a w))
            (t
             (setq v (|spleI1| #'|rdigit?| t))
             (|scanCheckRadix| (parse-integer a) v)
             (|scanExponent| (concat a "r" w) v))))
          (t (|lfrinteger| a w)))))))

```

6.0.225 defun rdigit?

[strpos p1002]

— defun rdigit? 0 —


```
(defun |rdigit?| (x)
  (strpos x "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" 0 nil))
```

6.0.226 defun lfinteger

— defun lfinteger 0 —

```
(defun |lfinteger| (x)
  (list '|integer| x))
```

6.0.227 defun lfrinteger

[concat p1003]

— defun lfrinteger 0 —

```
(defun |lfrinteger| (r x)
  (list '|integer| (concat r (concat "r" x))))
```

6.0.228 defun scanCheckRadix

```
;scanCheckRadix(r,w)==
;      ns:=#w
;      done:=false
;      for i in 0..ns-1 repeat
;        a:=rdigit? w.i
;        if null a or a>=r
;          then ncSoftError(cons($linepos,lnExtraBlanks $linepos+$n-ns+i),
;                                "S2CN0002", [w.i])
```

[\$n p??]
[\$linepos p??]

— defun scanCheckRadix —

```

(defun |scanCheckRadix| (r w)
  (let (a ns)
    (declare (special |$n| |$linepos|))
    (setq ns (length w))
    ((lambda (Var1 i)
      (loop
        (cond
          ((> i Var1) (return nil))
          (t
           (setq a (|rdigit?| (elt w i)))
           (cond
             ((or (null a) (not (< a r)))
              (|ncSoftError|
               (cons |$linepos| (+ (- (+ (|lnExtraBlanks| |$linepos|) |$n|) ns) i))
               'S2CN0002 (list (elt w i)))))))
      (setq i (+ i 1))))
    (- ns 1) 0)))

```

6.0.229 defun scanEscape

```

[scanEsc p125]
[scanWord p128]
[$n p??]

```

— defun scanEscape —

```

(defun |scanEscape| ()
  (declare (special |$n|))
  (setq |$n| (+ |$n| 1))
  (when (|scanEsc|) (|scanWord| t)))

```

6.0.230 defun scanError

```

[ncSoftError p353]
[lnExtraBlanks p347]
[lferror p138]
[$ln p??]
[$linepos p??]
[$n p??]

```

— defun scanError —

```
(defun |scanError| ()
  (let (n)
    (declare (special |$ln| |$linepos| |$n|))
    (setq n |$n|)
    (setq |$n| (+ |$n| 1))
    (|ncSoftError|
     (cons |$linepos| (+ (|lnExtraBlanks| |$linepos|) |$n|))
     'S2CN0003 (list (elt |$ln| n)))
    (|lferror| (elt |$ln| n))))
```

6.0.231 defun lferror

— defun lferror 0 —

```
(defun |lferror| (x)
  (list 'error| x))
```

6.0.232 defvar \$scanKeyTable

— postvars —

```
(eval-when (eval load)
  (defvar |scanKeyTable| (|scanKeyTableCons|)))
```

6.0.233 defun scanKeyTableCons

This function is used to build the scanKeyTable

```
;scanKeyTableCons()==
;  KeyTable:=MAKE_-HASHTABLE("CVEC",true)
;  for st in scanKeyWords repeat
;    HPUT(KeyTable,CAR st,CADR st)
;  KeyTable
```

— defun scanKeyTableCons —

```
(defun |scanKeyTableCons| ()
  (let (KeyTable)
    (setq KeyTable (make-hash-table :test #'equal))
    ((lambda (Var6 st)
      (loop
        (cond
          ((or (atom Var6) (progn (setq st (car Var6)) nil))
            (return nil))
          (t
            (hput KeyTable (car st) (cadr st))))
        (setq Var6 (cdr Var6))))
      |scanKeyWords| nil)
    KeyTable))
```

6.0.234 defvar \$scanDict

— postvars —

```
(eval-when (eval load)
  (defvar |scanDict| (|scanDictCons|)))
```

6.0.235 defun scanDictCons

```
;scanDictCons()==
;      l:= HKEYS scanKeyTable
;      d :=
;          a:=MAKE_-VEC(256)
;          b:=MAKE_-VEC(1)
;          VEC_-SETELT(b,0,MAKE_-CVEC 0)
;          for i in 0..255 repeat VEC_-SETELT(a,i,b)
;          a
;      for s in l repeat scanInsert(s,d)
;      d
```

[hkeys p1000]

— defun scanDictCons —

```
(defun |scanDictCons| ()
  (let (d b a l)
```

```

(setq l (hkeys |scanKeyTable|))
(setq d
  (progn
    (setq a (make-array 256))
    (setq b (make-array 1))
    (setf (svref b 0)
      (make-array 0 :fill-pointer 0 :element-type 'string-char))
    ((lambda (i)
      (loop
        (cond
          ((> i 255) (return nil))
          (t (setf (svref a i) b)))
        (setq i (+ i 1))))
      0)
    a))
((lambda (Var7 s)
  (loop
    (cond
      ((or (atom Var7) (progn (setq s (car Var7)) nil))
        (return nil))
      (t (|scanInsert| s d)))
    (setq Var7 (cdr Var7))))
  l nil)
d))

```

6.0.236 defun scanInsert

```

;scanInsert(s,d) ==
;   l := #s
;   h := QENUM(s,0)
;   u := ELT(d,h)
;   n := #u
;   k:=0
;   while l <= #(ELT(u,k)) repeat
;     k:=k+1
;   v := MAKE_-VEC(n+1)
;   for i in 0..k-1 repeat VEC_-SETELT(v,i,ELT(u,i))
;   VEC_-SETELT(v,k,s)
;   for i in k..n-1 repeat VEC_-SETELT(v,i+1,ELT(u,i))
;   VEC_-SETELT(d,h,v)
;   s

```

[qenum p1002]

— defun scanInsert —

```

(defun |scanInsert| (s d)
  (let (v k n u h l)
    (setq l (length s))
    (setq h (qenum s 0))
    (setq u (elt d h))
    (setq n (length u))
    (setq k 0)
    ((lambda ()
      (loop
        (cond
          ((< (length (elt u k)) l) (return nil))
          (t (setq k (+ k 1))))))
      (setq v (make-array (+ n 1)))
      ((lambda (Var2 i)
        (loop
          (cond
            ((> i Var2) (return nil))
            (t (setf (svref v i) (elt u i))))
          (setq i (+ i 1))))
        (- k 1) 0)
      (setf (svref v k) s)
      ((lambda (Var3 i)
        (loop
          (cond
            ((> i Var3) (return nil))
            (t (setf (svref v (+ i 1)) (elt u i))))
          (setq i (+ i 1))))
        (- n 1) k)
      (setf (svref d h) v)
      s))

```

6.0.237 defvar \$scanPun

— postvars —

```

(eval-when (eval load)
  (defvar |scanPun| (|scanPunCons|)))

```

6.0.238 defun scanPunCons

```

;scanPunCons() ==

```

```

;   listing := HKEYS scanKeyTable
;   a:=MAKE_-BVEC 256
;   for i in 0..255 repeat BVEC_-SETELT(a,i,0)
;   for k in listing repeat
;       if not startsId? k.0
;       then BVEC_-SETELT(a,QENUM(k,0),1)
;   a

```

[hkeys p1000]

— defun scanPunCons —

```

(defun |scanPunCons| ()
  (let (a listing)
    (setq listing (hkeys |scanKeyTable|))
    (setq a (make-array (list 256) :element-type 'bit :initial-element 0))
    ((lambda (i)
      (loop
        (cond
          (> i 255) (return nil))
          (t (setf (sbit a i) 0)))
        (setq i (+ i 1))))
      0)
    ((lambda (Var8 k)
      (loop
        (cond
          ((or (atom Var8) (progn (setq k (car Var8)) nil))
            (return nil))
          (t
            (cond
              ((null (|startsId?| (elt k 0)))
                (setf (sbit a (qenum k 0)) 1))))
            (setq Var8 (cdr Var8))))
        listing nil)
      a))

```

— —

Chapter 7

Input Stream Parser

7.0.239 defun Input Stream Parser

```
[trappoint p??]  
[npFirstTok p145]  
[npItem p144]  
[ncSoftError p353]  
[tokPosn p415]  
[pfWrong p298]  
[pfDocument p248]  
[pfListOf p247]  
[$ttok p??]  
[$stok p??]  
[$stack p??]  
[$inputStream p??]
```

— defun npParse —

```
(defun |npParse| (stream)  
  (let (|$ttok| |$stok| |$stack| |$inputStream| found)  
    (declare (special |$ttok| |$stack| |$inputStream| |$stok|))  
    (setq |$inputStream| stream)  
    (setq |$stack| nil)  
    (setq |$stok| nil)  
    (setq |$ttok| nil)  
    (|npFirstTok|)  
    (setq found (catch 'trappoint (|npItem|)))  
    (cond  
      ((eq found 'trapped)  
       (|ncSoftError| (|tokPosn| |$stok|) 's2cy0006 nil)  
       (|pfWrong| (|pfDocument| "top level syntax error") (|pfListOf| nil)))  
      ((null (null |$inputStream|))
```



```

(ncSoftError| (|tokPosn| |$stok|) 's2cy0002 nil)
(pfWrong|
 (|pfDocument| (list "input stream not exhausted"))
 (|pfListOf| nil)))
(null |$stack|)
(ncSoftError| (|tokPosn| |$stok|) 's2cy0009 nil)
(pfWrong| (|pfDocument| (list "stack empty")) (|pfListOf| nil)))
(t (car |$stack|))))

```

7.0.240 defun npItem

```

[npQualDef p147]
[npEqKey p147]
[npItem1 p144]
[npPop1 p146]
[pfEnSequence p265]
[npPush p145]
[pfNovalue p281]

```

— defun npItem —

```

(defun |npItem| ()
  (let (c b a tmp1)
    (when (|npQualDef|)
      (if (|npEqKey| 'semicolon)
        (progn
          (setq tmp1 (|npItem1| (|npPop1|)))
          (setq a (car tmp1))
          (setq b (cadr tmp1))
          (setq c (|pfEnSequence| b))
          (if a
            (|npPush| c)
            (|npPush| (|pfNovalue| c))))
        (|npPush| (|pfEnSequence| (|npPop1|)))))))

```

7.0.241 defun npItem1

```

[npQualDef p147]
[npEqKey p147]
[npItem1 p144]
[npPop1 p146]

```

— defun npItem1 —

```
(defun |npItem1| (c)
  (let (b a tmp1)
    (if (|npQualDef|)
      (if (|npEqKey| 'semicolon)
        (progn
          (setq tmp1 (|npItem1| (|npPop1|)))
          (setq a (car tmp1))
          (setq b (cadr tmp1))
          (list a (append c b)))
        (list t (append c (|npPop1|))))
      (list nil c))))
```

—————

7.0.242 defun npFirstTok

Sets the current leaf (\$tok) to the next leaf in the input stream. Sets the current token (\$ttok) cdr of the leaf. A leaf token looks like [head, token, position] where head is either an id or (id . alist) [tokConstruct p413]

[tokPosn p415]

[tokPart p415]

[\$ttok p??]

[\$tok p??]

[\$inputStream p??]

— defun npFirstTok —

```
(defun |npFirstTok| ()
  (declare (special |$ttok| |$tok| |$inputStream|))
  (if (null |$inputStream|)
    (setq |$tok| (|tokConstruct| 'error 'nomore (|tokPosn| |$tok|)))
    (setq |$tok| (car |$inputStream|)))
  (setq |$ttok| (|tokPart| |$tok|)))
```

—————

7.0.243 defun Push one item onto \$stack

[\$stack p??]

— defun npPush 0 —

```
(defun |npPush| (x)
  (declare (special |$stack|))
  (push x |$stack|))
```

7.0.244 defun Pop one item off \$stack

[\$stack p??]

— defun npPop1 0 —

```
(defun |npPop1| ()
  (declare (special |$stack|))
  (pop |$stack|))
```

7.0.245 defun Pop the second item off \$stack

[\$stack p??]

— defun npPop2 0 —

```
(defun |npPop2| ()
  (let (a)
    (declare (special |$stack|))
    (setq a (cadr |$stack|))
    (rplacd |$stack| (cddr |$stack|))
    a))
```

7.0.246 defun Pop the third item off \$stack

[\$stack p??]

— defun npPop3 0 —

```
(defun |npPop3| ()
  (let (a)
    (declare (special |$stack|))
    (setq a (caddr |$stack|))
```

```
(rplacd (cdr |$stack|) (cdddr |$stack|)) a))
```

7.0.247 defun npQualDef

```
[npComma p148]
[npPush p145]
[npPop1 p146]
```

— defun npQualDef —

```
(defun |npQualDef| ()
  (and (|npComma|) (|npPush| (list (|npPop1|))))))
```

7.0.248 defun Advance over a keyword

Test for the keyword, if found advance the token stream [npNext p147]

```
[$ttok p??]
[$stok p??]
```

— defun npEqKey —

```
(defun |npEqKey| (keyword)
  (declare (special |$ttok| |$stok|))
  (and
    (eq (caar |$stok|) '|key|)
    (eq keyword |$ttok|)
    (|npNext|)))
```

7.0.249 defun Advance the input stream

This advances the input stream. The call to npFirstTok picks off the next token in the input stream and updates the current leaf (\$stok) and the current token (\$ttok) [npFirstTok p145]

```
[$inputStream p??]
```

— defun npNext —

```
(defun |npNext| ()
  (declare (special |$inputStream|))
  (setq |$inputStream| (cdr |$inputStream|))
  (|npFirstTok|))
```

7.0.250 defun npComma

[npTuple p148]
[npQualifiedDefinition p149]

— defun npComma —

```
(defun |npComma| ()
  (|npTuple| #'|npQualifiedDefinition|))
```

7.0.251 defun npTuple

[npListofFun p223]
[npCommaBackSet p148]
[pfTupleListOf p294]

— defun npTuple —

```
(defun |npTuple| (|p|)
  (|npListofFun| |p| #'|npCommaBackSet| #'|pfTupleListOf|))
```

7.0.252 defun npCommaBackSet

[npEqKey p147]

— defun npCommaBackSet —

```
(defun |npCommaBackSet| ()
  (and
    (|npEqKey| 'comma)
    (or (|npEqKey| 'backset) t)))
```

7.0.253 defun npQualifiedDefinition

[npQualified p149]
[npDefinitionOrStatement p149]

— defun npQualifiedDefinition —

```
(defun |npQualifiedDefinition| ()
  (|npQualified| #'|npDefinitionOrStatement|))
```

7.0.254 defun npQualified

[npEqKey p147]
[npDefinition p169]
[npTrap p214]
[npPush p145]
[pfWhere p296]
[npPop1 p146]
[npLetQualified p168]

— defun npQualified —

```
(defun |npQualified| (f)
  (if (funcall f)
      (progn
        (do () ; while ... do
          ((not (and (|npEqKey| 'where) (or (|npDefinition|) (|npTrap|)))))
          (|npPush| (|pfWhere| (|npPop1|) (|npPop1|))))
        t)
      (|npLetQualified| f)))
```

7.0.255 defun npDefinitionOrStatement

[npBackTrack p150]
[npGives p150]
[npDef p189]

— defun npDefinitionOrStatement —

```
(defun |npDefinitionOrStatement| ()
  (|npBackTrack| #'|npGives| 'def #'|npDef|))
```

7.0.256 defun npBackTrack

```
[npState p214]
[npEqPeek p154]
[npRestore p154]
[npTrap p214]
```

— defun npBackTrack —

```
(defun |npBackTrack| (p1 p2 p3)
  (let (a)
    (setq a (|npState|))
    (when (apply p1 nil)
      (cond
        ((|npEqPeek| p2)
         (|npRestore| a)
         (or (apply p3 nil) (|npTrap|)))
        (t t)))))
```

7.0.257 defun npGives

```
[npBackTrack p150]
[npExit p217]
[npLambda p150]
```

— defun npGives —

```
(defun |npGives| ()
  (|npBackTrack| #'|npExit| 'gives #'|npLambda|))
```

7.0.258 defun npLambda

```
[npVariable p215]
[npLambda p150]
```

[npTrap p214]
 [npPush p145]
 [pfLam p274]
 [npPop2 p146]
 [npPop1 p146]
 [npEqKey p147]
 [npDefinitionOrStatement p149]
 [npType p151]
 [pfReturnTyped p287]

— **defun npLambda** —

```
(defun |npLambda| ()
  (or
    (and
      (|npVariable|)
      (or (|npLambda|) (|npTrap|))
      (|npPush| (|pfLam| (|npPop2|) (|npPop1|))))
    (and
      (|npEqKey| 'gives)
      (or (|npDefinitionOrStatement|) (|npTrap|)))
    (and
      (|npEqKey| 'colon)
      (or (|npType|) (|npTrap|))
      (|npEqKey| 'gives)
      (or (|npDefinitionOrStatement|) (|npTrap|))
      (|npPush| (|pfReturnTyped| (|npPop2|) (|npPop1|))))))
```

—————

7.0.259 **defun npType**

[npMatch p152]
 [npPop1 p146]
 [npWith p152]
 [npPush p145]

— **defun npType** —

```
(defun |npType| ()
  (and
    (|npMatch|)
    (let ((a (|npPop1|)))
      (or
        (|npWith| a)
        (|npPush| a))))))
```


7.0.260 defun npMatch

[npLeftAssoc p208]
[npSuch p152]

— **defun npMatch** —

```
(defun |npMatch| ()
  (|npLeftAssoc| '(is isnt) #'|npSuch|))
```

7.0.261 defun npSuch

[npLeftAssoc p208]
[npLogical p199]

— **defun npSuch** —

```
(defun |npSuch| ()
  (|npLeftAssoc| '(bar) #'|npLogical|))
```

7.0.262 defun npWith

[npEqKey p147]
[npState p214]
[npCategoryL p154]
[npTrap p214]
[npEqPeek p154]
[npRestore p154]
[npVariable p215]
[npCompMissing p153]
[npPush p145]
[pfWith p298]
[npPop2 p146]
[npPop1 p146]
[pfNothing p247]

— **defun npWith** —

```
(defun |npWith| (extra)
  (let (a)
    (and
      (|npEqKey| 'with)
      (progn
        (setq a (|npState|))
        (or (|npCategoryL|) (|npTrap|))
        (if (|npEqPeek| 'in)
            (progn
              (|npRestore| a)
              (and
                (or (|npVariable|) (|npTrap|))
                (|npCompMissing| 'in)
                (or (|npCategoryL|) (|npTrap|))
                (|npPush| (|pfWith| (|npPop2|) (|npPop1|) extra))))
              (|npPush| (|pfWith| (|pfNothing|) (|npPop1|) extra)))))))
```

7.0.263 defun npCompMissing

[npEqKey p147]
[npMissing p153]

— defun npCompMissing —

```
(defun |npCompMissing| (s)
  (or (|npEqKey| s) (|npMissing| s)))
```

7.0.264 defun npMissing

[trappoint p??]
[ncSoftError p353]
[tokPosn p415]
[pname p1001]
[\$stok p??]

— defun npMissing —

```
(defun |npMissing| (s)
  (declare (special |$stok|))
  (|ncSoftError| (|tokPosn| |$stok|) 'S2CY0007 (list (pname s)))
  (throw 'trappoint 'trapped))))
```

7.0.265 defun npRestore

```
[npFirstTok p145]
[$stack p??]
[$inputStream p??]
```

— **defun npRestore** —

```
(defun |npRestore| (x)
  (declare (special |$stack| |$inputStream|))
  (setq |$inputStream| (car x))
  (|npFirstTok|)
  (setq |$stack| (cdr x))
  t)
```

7.0.266 defun Peek for keyword s, no advance of token stream

```
[$ttok p??]
[$stok p??]
```

— **defun npEqPeek 0** —

```
(defun |npEqPeek| (s)
  (declare (special |$ttok| |$stok|))
  (and (eq (caar |$stok|) '|key|) (eq s |$ttok|)))
```

7.0.267 defun npCategoryL

```
[npCategory p155]
[npPush p145]
[pfUnSequence p295]
[npPop1 p146]
```

— **defun npCategoryL** —

```
(defun |npCategoryL| ()
  (and
    (|npCategory|)
    (|npPush| (|pfUnSequence| (|npPop1|))))))
```

7.0.268 defun npCategory

[npPP p211]
[npSCategory p155]

— defun npCategory —

```
(defun |npCategory| ()
  (|npPP| #'|npSCategory|))
```

7.0.269 defun npSCategory

[npWConditional p197]
[npCategoryL p154]
[npPush p145]
[npPop1 p146]
[npDefaultValue p196]
[npState p214]
[npPrimary p159]
[npEqPeek p154]
[npRestore p154]
[npSignature p156]
[npApplication p164]
[pfAttribute p255]
[npTrap p214]

— defun npSCategory —

```
(defun |npSCategory| ()
  (let (a)
    (cond
      ((|npWConditional| #'|npCategoryL|) (|npPush| (list (|npPop1|))))
      ((|npDefaultValue|) t)
      (t
       (setq a (|npState|))
```

```

(cond
  ((|npPrimary|)
   (cond
    ((|npEqPeek| 'colon) (|npRestore| a) (|npSignature|))
    (t
     (|npRestore| a)
     (or
      (and (|npApplication|) (|npPush| (list (|pfAttribute| (|npPop1|)))))
      (|npTrap|))))))
  (t nil))))))

```

7.0.270 defun npSignature

```

[|npSigItemList| p156]
[|npPush| p145]
[|pfWDec| p295]
[|pfNothing| p247]
[|npPop1| p146]

```

— defun npSignature —

```

(defun |npSignature| ()
  (and (|npSigItemList|) (|npPush| (|pfWDec| (|pfNothing|) (|npPop1|)))))

```

7.0.271 defun npSigItemList

```

[|npListing| p157]
[|npSigItem| p158]
[|npPush| p145]
[|pfListOf| p247]
[|pfAppend| p257]
[|pfParts| p251]
[|npPop1| p146]

```

— defun npSigItemList —

```

(defun |npSigItemList| ()
  (and
   (|npListing| #'|npSigItem|)
   (|npPush| (|pfListOf| (|pfAppend| (|pfParts| (|npPop1|)))))
  ))

```

7.0.272 defun npListing

[npList p157]
[pfListOf p247]

— defun npListing —

```
(defun |npListing| (p)
  (|npList| p 'comma #'|pfListOf|))
```

7.0.273 defun Always produces a list, fn is applied to it

[npEqKey p147]
[npTrap p214]
[npPush p145]
[npPop3 p146]
[npPop2 p146]
[npPop1 p146]
[\$stack p??]

— defun npList —

```
(defun |npList| (f str1 fn)
  (let (a)
    (declare (special |$stack|))
    (cond
      ((apply f nil)
       (cond
         ((and (|npEqKey| str1)
              (or (|npEqKey| 'backset) t)
              (or (apply f nil) (|npTrap|))))
          (setq a |$stack|)
          (setq |$stack| nil)
          (do () ; while .. do nothing
              ((not
                (and (|npEqKey| str1)
                     (or (|npEqKey| 'backset) t)
                     (or (apply f nil) (|npTrap|))))
               nil))
          (setq |$stack| (cons (nreverse |$stack|) a))
          (|npPush| (funcall fn (cons (|npPop3|) (cons (|npPop2|) (|npPop1|))))))
      (t (|npPush| (funcall fn (list (|npPop1|)))))))
```

```
(t (|npPush| (funccall fn nil))))))
```

7.0.274 defun npSigItem

```
[npTypeVariable p158]
[npSigDecl p159]
[npTrap p214]
```

— defun npSigItem —

```
(defun |npSigItem| ()
  (and (|npTypeVariable|) (or (|npSigDecl|) (|npTrap|)))))
```

7.0.275 defun npTypeVariable

```
[npParenthesized p216]
[npTypeVariablelist p159]
[npSignatureDefinee p158]
[npPush p145]
[pfListOf p247]
[npPop1 p146]
```

— defun npTypeVariable —

```
(defun |npTypeVariable| ()
  (or
    (|npParenthesized| #'|npTypeVariablelist|)
    (and (|npSignatureDefinee|) (|npPush| (|pfListOf| (list (|npPop1|)))))))
```

7.0.276 defun npSignatureDefinee

```
[npName p206]
[npInfixOperator p162]
[npPrefixColon p163]
```

— defun npSignatureDefinee —

```
(defun |npSignatureDefinee| ()
  (or (|npName|) (|npInfixOperator|) (|npPrefixColon|)))
```

7.0.277 defun npTypeVariablelist

[npListing p157]
[npSignatureDefinee p158]

— defun npTypeVariablelist —

```
(defun |npTypeVariablelist| ()
  (|npListing| #'|npSignatureDefinee|))
```

7.0.278 defun npSigDecl

[npEqKey p147]
[npType p151]
[npTrap p214]
[npPush p145]
[pfSpread p241]
[pfParts p251]
[npPop2 p146]
[npPop1 p146]

— defun npSigDecl —

```
(defun |npSigDecl| ()
  (and
    (|npEqKey| 'colon)
    (or (|npType|) (|npTrap|))
    (|npPush| (|pfSpread| (|pfParts| (|npPop2|)) (|npPop1|))))))
```

7.0.279 defun npPrimary

[npPrimary1 p166]
[npPrimary2 p160]

— defun npPrimary —

```
(defun |npPrimary| ()
  (or (|npPrimary1|) (|npPrimary2|)))
```

—————

7.0.280 defun npPrimary2

```
[npEncAp p184]
[npAtom2 p161]
[npAdd p161]
[pfNothing p247]
[npWith p152]
```

— defun npPrimary2 —

```
(defun |npPrimary2| ()
  (or
    (|npEncAp| #'|npAtom2|)
    (|npAdd| (|pfNothing|))
    (|npWith| (|pfNothing|))))
```

—————

7.0.281 defun npADD

TPDHERE: Note that there is also an npAdd function [npType p151]

```
[npPop1 p146]
[npAdd p161]
[npPush p145]
```

— defun npADD —

```
(defun |npADD| ()
  (let (a)
    (and
      (|npType|)
      (progn
        (setq a (|npPop1|))
        (or
          (|npAdd| a)
          (|npPush| a))))))
```

7.0.282 defun npAdd

TPDHERE: Note that there is also an npADD function [npEqKey p147]

[npState p214]
 [npDefinitionOrStatement p149]
 [npTrap p214]
 [npEqPeek p154]
 [npRestore p154]
 [npVariable p215]
 [npCompMissing p153]
 [npDefinitionOrStatement p149]
 [npPush p145]
 [pfAdd p254]
 [npPop2 p146]
 [npPop1 p146]
 [pfNothing p247]

— defun npAdd —

```
(defun |npAdd| (extra)
  (let (a)
    (and
      (|npEqKey| 'add)
      (progn
        (setq a (|npState|))
        (or (|npDefinitionOrStatement|) (|npTrap|))
        (cond
          ((|npEqPeek| 'in)
            (progn
              (|npRestore| a)
              (and
                (or (|npVariable|) (|npTrap|))
                (|npCompMissing| 'in)
                (or (|npDefinitionOrStatement|) (|npTrap|))
                (|npPush| (|pfAdd| (|npPop2|) (|npPop1|) extra))))))
          (t
            (|npPush| (|pfAdd| (|pfNothing|) (|npPop1|) extra))))))))
```

7.0.283 defun npAtom2

[npInfixOperator p162]
 [npAmpersand p206]

[npPrefixColon p163]
 [npFromdom p204]

— defun npAtom2 —

```
(defun |npAtom2| ()
  (and
    (or (|npInfixOperator|) (|npAmpersand|) (|npPrefixColon|))
    (|npFromdom|)))
```

—————

7.0.284 defun npInfixOperator

[npInfixOp p163]
 [npState p214]
 [npEqKey p147]
 [npInfixOp p163]
 [npPush p145]
 [pfSymb p253]
 [npPop1 p146]
 [tokPosn p415]
 [npRestore p154]
 [tokConstruct p413]
 [tokPart p415]
 [\$stok p??]

— defun npInfixOperator —

```
(defun |npInfixOperator| ()
  (let (b a)
    (declare (special |$stok|))
    (or (|npInfixOp|)
      (progn
        (setq a (|npState|))
        (setq b |$stok|)
        (cond
          ((and (|npEqKey| ' '|) (|npInfixOp|))
           (|npPush| (|pfSymb| (|npPop1|) (|tokPosn| b))))
          (t
           (|npRestore| a)
           (cond
             ((and (|npEqKey| 'backquote) (|npInfixOp|))
              (setq a (|npPop1|))
              (|npPush| (|tokConstruct| 'lidsy| (|tokPart| a) (|tokPosn| a))))
             (t
```

```
(|npRestore| a)
nil))))))
```

7.0.285 defun npInfixOp

```
[npPushId p211]
[$ttok p??]
[$stok p??]
```

— defun npInfixOp —

```
(defun |npInfixOp| ()
  (declare (special |$ttok| |$stok|))
  (and
    (eq (caar |$stok|) '|key|)
    (get |$ttok| 'infgeneric)
    (|npPushId|)))
```

7.0.286 defun npPrefixColon

```
[npEqPeek p154]
[npPush p145]
[tokConstruct p413]
[tokPosn p415]
[npNext p147]
[$stok p??]
```

— defun npPrefixColon —

```
(defun |npPrefixColon| ()
  (declare (special |$stok|))
  (and
    (|npEqPeek| 'colon)
    (progn
      (|npPush| (|tokConstruct| '|id| '|:| (|tokPosn| |$stok|)))
      (|npNext|))))
```

7.0.287 defun npApplication

```
[npDotted p164]
 [npPrimary p159]
 [npApplication2 p165]
 [npPush p145]
 [pfApplication p255]
 [npPop2 p146]
 [npPop1 p146]
```

— **defun npApplication** —

```
(defun |npApplication| ()
  (and
    (|npDotted| #'|npPrimary|)
    (or
      (and
        (|npApplication2|)
        (|npPush| (|pfApplication| (|npPop2|) (|npPop1|))))
      t)))
```

—————

7.0.288 defun npDotted

```
[ p??]
```

— **defun npDotted** —

```
(defun |npDotted| (f)
  (and (apply f nil) (|npAnyNo| #'|npSelector|)))
```

—————

7.0.289 defun npAnyNo

fn must transform the head of the stack

— **defun npAnyNo 0** —

```
(defun |npAnyNo| (fn)
  (do () ((not (apply fn nil)))) ; while apply do...
  t)
```

—————

7.0.290 defun npSelector

[npEqKey p147]
 [npPrimary p159]
 [npTrap p214]
 [npPush p145]
 [pfApplication p255]
 [npPop2 p146]
 [npPop1 p146]

— **defun npSelector** —

```
(defun |npSelector| ()
  (and
    (|npEqKey| 'dot)
    (or (|npPrimary|) (|npTrap|))
    (|npPush| (|pfApplication| (|npPop2|) (|npPop1|))))))
```

—————

7.0.291 defun npApplication2

[npDotted p164]
 [npPrimary1 p166]
 [npApplication2 p165]
 [npPush p145]
 [pfApplication p255]
 [npPop2 p146]
 [npPop1 p146]

— **defun npApplication2** —

```
(defun |npApplication2| ()
  (and
    (|npDotted| #'|npPrimary1|)
    (or
      (and
        (|npApplication2|)
        (|npPush| (|pfApplication| (|npPop2|) (|npPop1|))))
      t)))
```

—————

7.0.292 defun npPrimary1

[npEncAp p184]
 [npAtom1 p185]
 [npLet p168]
 [npFix p168]
 [npMacro p166]
 [npBPPileDefinition p190]
 [npDefn p189]
 [npRule p195]

— **defun npPrimary1** —

```
(defun |npPrimary1| ()
  (or
    (|npEncAp| #'|npAtom1|)
    (|npLet|)
    (|npFix|)
    (|npMacro|)
    (|npBPPileDefinition|)
    (|npDefn|)
    (|npRule|)))
```

—————

7.0.293 defun npMacro

[npPP p211]
 [npMdef p166]

— **defun npMacro** —

```
(defun |npMacro| ()
  (and
    (|npEqKey| 'macro)
    (|npPP| #'|npMdef|)))
```

—————

7.0.294 defun npMdef

TPDHERE: Beware that this function occurs with uppercase also [npQuiver p200]

[pfCheckMacroOut p242]
 [npPop1 p146]

[npDefTail p196]
 [npTrap p214]
 [npPop1 p146]
 [npPush p145]
 [pfMacro p279]
 [pfPushMacroBody p245]

— **defun npMdef** —

```
(defun |npMdef| ()
  (let (body arg op tmp)
    (when (|npQuiver|) ;[op,arg]:= pfCheckMacroOut(npPop1())
      (setq tmp (|pfCheckMacroOut| (|npPop1|)))
      (setq op (car tmp))
      (setq arg (cadr tmp))
      (or (|npDefTail|) (|npTrap|))
      (setq body (|npPop1|))
      (if (null arg)
          (|npPush| (|pfMacro| op body))
          (|npPush| (|pfMacro| op (|pfPushMacroBody| arg body)))))))
```

—————

7.0.295 **defun npMDEF**

TPDHERE: Beware that this function occurs with lowercase also [npBackTrack p150]

[npStatement p172]
 [npMDEFinition p167]

— **defun npMDEF** —

```
(defun |npMDEF| ()
  (|npBackTrack| #'|npStatement| 'mdef #'|npMDEFinition|))
```

—————

7.0.296 **defun npMDEFinition**

[npPP p211]
 [npMdef p166]

— **defun npMDEFinition** —


```
(defun |npMDEFinition| ()
  (|npPP| #'|npMdef|))
```

7.0.297 defun npFix

```
[npEqKey p147]
[npDef p189]
[npPush p145]
[pfFix p267]
[npPop1 p146]
```

— defun npFix —

```
(defun |npFix| ()
  (and
    (|npEqKey| 'fix)
    (|npPP| #'|npDef|)
    (|npPush| (|pfFix| (|npPop1|))))))
```

7.0.298 defun npLet

```
[npLetQualified p168]
[npDefinitionOrStatement p149]
```

— defun npLet —

```
(defun |npLet| ()
  (|npLetQualified| #'|npDefinitionOrStatement|))
```

7.0.299 defun npLetQualified

```
[npEqKey p147]
[npDefinition p169]
[npTrap p214]
[npCompMissing p153]
[npPush p145]
```

[pfWhere p296]
 [npPop2 p146]
 [npPop1 p146]

— **defun npLetQualified** —

```
(defun |npLetQualified| (f)
  (and
    (|npEqKey| 'let)
    (or (|npDefinition|) (|npTrap|))
    (|npCompMissing| 'in)
    (or #'f (|npTrap|))
    (|npPush| (|pfWhere| (|npPop2|) (|npPop1|))))))
```

—————

7.0.300 **defun npDefinition**

[npPP p211]
 [npDefinitionItem p169]
 [npPush p145]
 [pfSequenceToList p240]
 [npPop1 p146]

— **defun npDefinition** —

```
(defun |npDefinition| ()
  (and
    (|npPP| #'|npDefinitionItem|)
    (|npPush| (|pfSequenceToList| (|npPop1|))))))
```

—————

7.0.301 **defun npDefinitionItem**

[npTyping p170]
 [npImport p182]
 [npState p214]
 [npStatement p172]
 [npEqPeek p154]
 [npRestore p154]
 [npDef p189]
 [npMacro p166]
 [npDefn p189]

[npTrap p214]

— **defun npDefinitionItem** —

```
(defun |npDefinitionItem| ()
  (let (a)
    (or (|npTyping|)
        (|npImport|)
        (progn
          (setq a (|npState|))
          (cond
            ((|npStatement|)
             (cond
              ((|npEqPeek| 'def)
               (|npRestore| a)
               (|npDef|))
              (t
               (|npRestore| a)
               (or (|npMacro|) (|npDefn|))))))
          (t (|npTrap|))))))
```

7.0.302 **defun npTyping**

[npEqKey p147]
 [npDefaultItemList p170]
 [npTrap p214]
 [npPush p145]
 [pfTyping p293]
 [npPop1 p146]

— **defun npTyping** —

```
(defun |npTyping| ()
  (and
    (|npEqKey| 'default)
    (or (|npDefaultItemList|) (|npTrap|))
    (|npPush| (|pfTyping| (|npPop1|)))))
```

7.0.303 **defun npDefaultItemList**

[npPC p??]
 [npSDefaultItem p171]

[npPush p145]
 [pfUnSequence p295]
 [npPop1 p146]

— defun npDefaultItemList —

```
(defun |npDefaultItemList| ()
  (and
    (|npPC| #'|npSDefaultItem|)
    (|npPush| (|pfUnSequence| (|npPop1|)))))
```

—————

7.0.304 defun npSDefaultItem

[npListing p157]
 [npDefaultItem p171]
 [npPush p145]
 [pfAppend p257]
 [pfParts p251]
 [npPop1 p146]

— defun npSDefaultItem —

```
(defun |npSDefaultItem| ()
  (and
    (|npListing| #'|npDefaultItem|)
    (|npPush| (|pfAppend| (|pfParts| (|npPop1|)))))
```

—————

7.0.305 defun npDefaultItem

[npTypeVariable p158]
 [npDefaultDecl p172]
 [npTrap p214]

— defun npDefaultItem —

```
(defun |npDefaultItem| ()
  (and
    (|npTypeVariable|)
    (or (|npDefaultDecl|) (|npTrap|))))
```

—————

7.0.306 defun npDefaultDecl

[npEqKey p147]
 [npType p151]
 [npTrap p214]
 [npPush p145]
 [pfSpread p241]
 [pfParts p251]
 [npPop2 p146]
 [npPop1 p146]

— **defun npDefaultDecl** —

```
(defun |npDefaultDecl| ()
  (and
    (|npEqKey| 'colon)
    (or (|npType|) (|npTrap|))
    (|npPush| (|pfSpread| (|pfParts| (|npPop2|)) (|npPop1|)))))
```

—————

7.0.307 defun npStatement

[npExpress p181]
 [npLoop p177]
 [npIterate p176]
 [npReturn p180]
 [npBreak p176]
 [npFree p175]
 [npImport p182]
 [npInline p176]
 [npLocal p175]
 [npExport p173]
 [npTyping p170]
 [npVoid p181]

— **defun npStatement** —

```
(defun |npStatement| ()
  (or
    (|npExpress|)
    (|npLoop|)
    (|npIterate|)
    (|npReturn|)
    (|npBreak|)
    (|npFree|))
```

```
(|npImport|)
(|npInline|)
(|npLocal|)
(|npExport|)
(|npTyping|)
(|npVoid|)))
```

7.0.308 defun npExport

```
[npEqKey p147]
[npLocalItemlist p173]
[npTrap p214]
[npPush p145]
[pfExport p266]
[npPop1 p146]
```

— defun npExport —

```
(defun |npExport| ()
  (and
    (|npEqKey| 'export)
    (or (|npLocalItemlist|) (|npTrap|))
    (|npPush| (|pfExport| (|npPop1|)))))
```

7.0.309 defun npLocalItemlist

```
[npPC p??]
[npSLocalItem p174]
[npPush p145]
[pfUnSequence p295]
[npPop1 p146]
```

— defun npLocalItemlist —

```
(defun |npLocalItemlist| ()
  (and
    (|npPC| #'|npSLocalItem|)
    (|npPush| (|pfUnSequence| (|npPop1|)))))
```

7.0.310 defun npSLocalItem

[npListing p157]

[npLocalItem p174]

[npPush p145]

[pfAppend p257]

[pfParts p251]

[npPop1 p146]

— **defun npSLocalItem** —

(defun |npSLocalItem| ()

(and

(|npListing| #'|npLocalItem|)

(|npPush| (|pfAppend| (|pfParts| (|npPop1|))))))

—————

7.0.311 defun npLocalItem

[npTypeVariable p158]

[npLocalDecl p174]

— **defun npLocalItem** —

(defun |npLocalItem| ()

(and

(|npTypeVariable|)

(|npLocalDecl|))))

—————

7.0.312 defun npLocalDecl

[npEqKey p147]

[npType p151]

[npTrap p214]

[npPush p145]

[pfSpread p241]

[pfParts p251]

[npPop2 p146]

[npPop1 p146]

[pfNothing p247]

— defun npLocalDecl —

```
(defun |npLocalDecl| ()
  (or
    (and
      (|npEqKey| 'colon)
      (or (|npType|) (|npTrap|))
      (|npPush| (|pfSpread| (|pfParts| (|npPop2|)) (|npPop1|))))
      (|npPush| (|pfSpread| (|pfParts| (|npPop1|)) (|pfNothing|)))))
```

—————

7.0.313 defun npLocal

```
[npEqKey p147]
[npLocalItemlist p173]
[npTrap p214]
[npPush p145]
[pfLocal p276]
[npPop1 p146]
```

— defun npLocal —

```
(defun |npLocal| ()
  (and
    (|npEqKey| '|local|)
    (or (|npLocalItemlist|) (|npTrap|))
    (|npPush| (|pfLocal| (|npPop1|)))))
```

—————

7.0.314 defun npFree

```
[npEqKey p147]
[npLocalItemlist p173]
[npTrap p214]
[npPush p145]
[pfFree p267]
[npPop1 p146]
```

— defun npFree —

```
(defun |npFree| ()
```



```
(and
  (|npEqKey| 'free)
  (or (|npLocalItemlist|) (|npTrap|))
  (|npPush| (|pfFree| (|npPop1|))))))
```

7.0.315 defun npInline

```
[npAndOr p183]
[npQualTypelist p182]
[pfInline p274]
```

— defun npInline —

```
(defun |npInline| ()
  (|npAndOr| 'inline #'|npQualTypelist| #'|pfInline|))
```

7.0.316 defun npIterate

```
[npEqKey p147]
[npPush p145]
[pfIterate p273]
[pfNothing p247]
```

— defun npIterate —

```
(defun |npIterate| ()
  (and (|npEqKey| 'iterate) (|npPush| (|pfIterate| (|pfNothing|)))))
```

7.0.317 defun npBreak

```
[npEqKey p147]
[npPush p145]
[pfBreak p260]
[pfNothing p247]
```

— defun npBreak —

```
(defun |npBreak| ()
  (and (|npEqKey| 'break) (|npPush| (|pfBreak| (|pfNothing|)))))
```

7.0.318 defun npLoop

```
[npIterators p177]
[npCompMissing p153]
[npAssign p218]
[npTrap p214]
[npPush p145]
[pfLp p278]
[npPop2 p146]
[npPop1 p146]
[npEqKey p147]
[pfLoop1 p277]
```

— defun npLoop —

```
(defun |npLoop| ()
  (or
    (and
      (|npIterators|)
      (|npCompMissing| 'repeat)
      (or (|npAssign|) (|npTrap|))
      (|npPush| (|pfLp| (|npPop2|) (|npPop1|))))
    (and
      (|npEqKey| 'repeat)
      (or (|npAssign|) (|npTrap|))
      (|npPush| (|pfLoop1| (|npPop1|)))))
```

7.0.319 defun npIterators

```
[npForIn p179]
[npZeroOrMore p179]
[npIterator p178]
[npPush p145]
[npPop2 p146]
[npPop1 p146]
[npWhile p179]
[npIterators p177]
```

— **defun npIterators** —

```
(defun |npIterators| ()
  (or
    (and
      (|npForIn|)
      (|npZeroOrMore| #'|npIterator|)
      (|npPush| (cons (|npPop2|) (|npPop1|))))
    (and
      (|npWhile|)
      (or
        (and (|npIterators|) (|npPush| (cons (|npPop2|) (|npPop1|))))
        (|npPush| (list (|npPop1|)))))))
```

—————

7.0.320 **defun npIterator**

```
[npForIn p179]
[npSuchThat p178]
[npWhile p179]
```

— **defun npIterator** —

```
(defun |npIterator| ()
  (or
    (|npForIn|)
    (|npSuchThat|)
    (|npWhile|))))
```

—————

7.0.321 **defun npSuchThat**

```
[npAndOr p183]
[npLogical p199]
[pfSuchthat p290]
```

— **defun npSuchThat** —

```
(defun |npSuchThat| ()
  (|npAndOr| 'bar #'|npLogical| #'|pfSuchthat|))
```

—————

7.0.322 defun Apply argument 0 or more times

[npPush p145]
 [npPop2 p146]
 [npPop1 p146]
 [\$stack p??]

— defun npZeroOrMore —

```
(defun |npZeroOrMore| (f)
  (let (a)
    (declare (special |$stack|))
    (cond
      ((apply f nil)
       (setq a |$stack|)
       (setq |$stack| nil)
       (do () ((not (apply f nil)))) ; while .. do
       (setq |$stack| (cons (nreverse |$stack|) a))
       (|npPush| (cons (|npPop2|) (|npPop1|))))
      (t (progn (|npPush| nil) t)))))
```

—————

7.0.323 defun npWhile

[npAndOr p183]
 [npLogical p199]
 [pfWhile p297]

— defun npWhile —

```
(defun |npWhile| ()
  (|npAndOr| 'while #'|npLogical| #'|pfWhile|))
```

—————

7.0.324 defun npForIn

[npEqKey p147]
 [npVariable p215]
 [npTrap p214]
 [npCompMissing p153]
 [npBy p201]
 [npPush p145]

```
[pfForin p268]
[npPop2 p146]
[npPop1 p146]
```

— **defun npForIn** —

```
(defun |npForIn| ()
  (and
    (|npEqKey| 'for)
    (or (|npVariable|) (|npTrap|))
    (|npCompMissing| 'in)
    (or (|npBy|) (|npTrap|))
    (|npPush| (|pfForin| (|npPop2|) (|npPop1|))))))
```

—————

7.0.325 **defun npReturn**

```
[npEqKey p147]
[npExpress p181]
[npPush p145]
[pfNothing p247]
[npEqKey p147]
[npName p206]
[npTrap p214]
[pfReturn p286]
[npPop2 p146]
[npPop1 p146]
[pfReturnNoName p287]
```

— **defun npReturn** —

```
(defun |npReturn| ()
  (and
    (|npEqKey| 'return)
    (or
      (|npExpress|)
      (|npPush| (|pfNothing|)))
    (or
      (and
        (|npEqKey| 'from)
        (or (|npName|) (|npTrap|))
        (|npPush| (|pfReturn| (|npPop2|) (|npPop1|))))
      (|npPush| (|pfReturnNoName| (|npPop1|))))))
```

—————

7.0.326 defun npVoid

[npAndOr p183]
 [npStatement p172]
 [pfNoValue p281]

— **defun npVoid** —

```
(defun |npVoid| ()
  (|npAndOr| 'do #'|npStatement| #'|pfNoValue|))
```

—————

7.0.327 defun npExpress

[npExpress1 p181]
 [npIterators p177]
 [npPush p145]
 [pfCollect p262]
 [npPop2 p146]
 [pfListOf p247]
 [npPop1 p146]

— **defun npExpress** —

```
(defun |npExpress| ()
  (and
    (|npExpress1|)
    (or
      (and
        (|npIterators|)
        (|npPush| (|pfCollect| (|npPop2|) (|pfListOf| (|npPop1|))))))
      t)))
```

—————

7.0.328 defun npExpress1

[npConditionalStatement p182]
 [npADD p160]

— **defun npExpress1** —

```
(defun |npExpress1| ()
```

```
(or (|npConditionalStatement|) (|npADD|)))
```

7.0.329 defun npConditionalStatement

[npConditional p197]
[npQualifiedDefinition p149]

— defun npConditionalStatement —

```
(defun |npConditionalStatement| ()
  (|npConditional| #'|npQualifiedDefinition|))
```

7.0.330 defun npImport

[npAndOr p183]
[npQualTypelist p182]
[pfImport p273]

— defun npImport —

```
(defun |npImport| ()
  (|npAndOr| 'import #'|npQualTypelist| #'|pfImport|))
```

7.0.331 defun npQualTypelist

[npPC p??]
[npSQualTypelist p183]
[npPush p145]
[pfUnSequence p295]
[npPop1 p146]

— defun npQualTypelist —

```
(defun |npQualTypelist| ()
  (and
    (|npPC| #'|npSQualTypelist|)
```

```
(|npPush| (|pfUnSequence| (|npPop1|))))))
```

7.0.332 defun npSQualTypelist

```
[npListing p157]
[npQualType p183]
[npPush p145]
[pfParts p251]
[npPop1 p146]
```

— defun npSQualTypelist —

```
(defun |npSQualTypelist| ()
  (and
    (|npListing| #'|npQualType|)
    (|npPush| (|pfParts| (|npPop1|))))))
```

7.0.333 defun npQualType

```
[npType p151]
[npPush p145]
[pfQualType p284]
[npPop1 p146]
[pfNothing p247]
```

— defun npQualType —

```
(defun |npQualType| ()
  (and
    (|npType|)
    (|npPush| (|pfQualType| (|npPop1|) (|pfNothing|)))))
```

7.0.334 defun npAndOr

```
[npEqKey p147]
[npTrap p214]
```


[npPush p145]
[npPop1 p146]

— **defun npAndOr** —

```
(defun |npAndOr| (keyword p f)
  (and
    (|npEqKey| keyword)
    (or (apply p nil) (|npTrap|))
    (|npPush| (funcall f (|npPop1|)))))
```

—————

7.0.335 **defun npEncAp**

[npAnyNo p164]
[npEncl p184]
[npFromdom p204]

— **defun npEncAp** —

```
(defun |npEncAp| (f)
  (and (apply f nil) (|npAnyNo| #'|npEncl|) (|npFromdom|)))
```

—————

7.0.336 **defun npEncl**

[npBDefinition p187]
[npPush p145]
[pfApplication p255]
[npPop2 p146]
[npPop1 p146]

— **defun npEncl** —

```
(defun |npEncl| ()
  (and
    (|npBDefinition|)
    (|npPush| (|pfApplication| (|npPop2|) (|npPop1|)))))
```

—————

7.0.337 defun npAtom1

[npPDefinition p185]
 [npName p206]
 [npConstTok p186]
 [npDollar p185]
 [npBDefinition p187]
 [npFromdom p204]

— **defun npAtom1** —

```
(defun |npAtom1| ()
  (or
    (|npPDefinition|)
    (and
      (or (|npName|) (|npConstTok|) (|npDollar|) (|npBDefinition|))
      (|npFromdom|))))
```

7.0.338 defun npPDefinition

[npParenthesized p216]
 [npDefinitionlist p195]
 [npPush p145]
 [pfEnSequence p265]
 [npPop1 p146]

— **defun npPDefinition** —

```
(defun |npPDefinition| ()
  (and
    (|npParenthesized| #'|npDefinitionlist|)
    (|npPush| (|pfEnSequence| (|npPop1|)))))
```

7.0.339 defun npDollar

[npEqPeek p154]
 [npPush p145]
 [tokConstruct p413]
 [tokPosn p415]
 [npNext p147]

```
[$tok p??]
```

— defun npDollar —

```
(defun |npDollar| ()
  (declare (special |$tok|))
  (and (|npEqPeek| '$)
    (progn
      (|npPush| (|tokConstruct| '|id| '$ (|tokPosn| |$tok|)))
      (|npNext|))))
```

—————

7.0.340 defun npConstTok

```
[tokType p415]
[npPush p145]
[npNext p147]
[npEqPeek p154]
[npState p214]
[npPrimary1 p166]
[pfSymb p253]
[npPop1 p146]
[tokPosn p415]
[npRestore p154]
[$tok p??]
```

— defun npConstTok —

```
(defun |npConstTok| ()
  (let (b a)
    (declare (special |$tok|))
    (cond
      ((member (|tokType| |$tok|) '(|integer| |string| |char| |float| |command|))
        (|npPush| |$tok|)
        (|npNext|))
      ((|npEqPeek| '| '|)
        (setq a |$tok|)
        (setq b (|npState|))
        (|npNext|)
        (cond
          ((and (|npPrimary1|)
            (|npPush| (|pfSymb| (|npPop1|) (|tokPosn| a))))
            t)
          (t (|npRestore| b) nil)))
      (t nil))))
```

7.0.341 defun npBDefinition

[npPDefinition p185]
 [npBracketed p187]
 [npDefinitionlist p195]

— defun npBDefinition —

```
(defun |npBDefinition| ()
  (or
    (|npPDefinition|)
    (|npBracketed| #'|npDefinitionlist|)))
```

7.0.342 defun npBracketed

[npParened p187]
 [npBracked p188]
 [npBraced p188]
 [npAngleBared p188]

— defun npBracketed —

```
(defun |npBracketed| (f)
  (or
    (|npParened| f)
    (|npBracked| f)
    (|npBraced| f)
    (|npAngleBared| f)))
```

7.0.343 defun npParened

[npEnclosed p213]
 [pfParen p283]

— defun npParened —

```
(defun |npParened| (f)
```

```
(or (|npEnclosed| '(| ')| #'|pfParen| f)
    (|npEnclosed| '(\|| '|\)| #'|pfParen| f)))
```

7.0.344 defun npBracked

```
[npEnclosed p213]
[pfBracket p259]
[pfBracketBar p259]
```

— defun npBracked —

```
(defun |npBracked| (f)
  (or (|npEnclosed| '[' ']' #'|pfBracket| f)
      (|npEnclosed| '([\|| '|\]| #'|pfBracketBar| f)))
```

7.0.345 defun npBraced

```
[npEnclosed p213]
[pfBrace p259]
[pfBraceBar p259]
```

— defun npBraced —

```
(defun |npBraced| (f)
  (or (|npEnclosed| '{ '}' #'|pfBrace| f)
      (|npEnclosed| '({\|| '|\}| #'|pfBraceBar| f)))
```

7.0.346 defun npAngleBared

```
[npEnclosed p213]
[pfHide p271]
```

— defun npAngleBared —

```
(defun |npAngleBared| (f)
  (|npEnclosed| '<|>' #'|pfHide| f))
```

7.0.347 defun npDefn

[npEqKey p147]
 [npPP p211]
 [npDef p189]

— defun npDefn —

```
(defun |npDefn| ()
  (and
    (|npEqKey| 'defn)
    (|npPP| #'|npDef|)))
```

7.0.348 defun npDef

[npMatch p152]
 [pfCheckItOut p241]
 [npPop1 p146]
 [npDefTail p196]
 [npTrap p214]
 [npPop1 p146]
 [npPush p145]
 [pfDefinition p263]
 [pfPushBody p251]

— defun npDef —

```
(defun |npDef| ()
  (let (body rt arg op tmp1)
    (when (|npMatch|)
      ; [op,arg,rt]:= pfCheckItOut(npPop1())
      (setq tmp1 (|pfCheckItOut| (|npPop1|)))
      (setq op (car tmp1))
      (setq arg (cadr tmp1))
      (setq rt (caddr tmp1))
      (or (|npDefTail|) (|npTrap|))
      (setq body (|npPop1|))
      (if (null arg)
        (|npPush| (|pfDefinition| op body))
        (|npPush| (|pfDefinition| op (|pfPushBody| rt arg body)))))))
```

7.0.349 defun npBPileDefinition

[npPileBracketed p190]
 [npPileDefinitionlist p191]
 [npPush p145]
 [pfSequence p289]
 [pfListOf p247]
 [npPop1 p146]

— defun npBPileDefinition —

```
(defun |npBPileDefinition| ()
  (and
    (|npPileBracketed| #'|npPileDefinitionlist|)
    (|npPush| (|pfSequence| (|pfListOf| (|npPop1|))))))
```

7.0.350 defun npPileBracketed

[npEqKey p147]
 [npPush p145]
 [pfNothing p247]
 [npMissing p153]
 [pfPile p251]
 [npPop1 p146]

— defun npPileBracketed —

```
(defun |npPileBracketed| (f)
  (cond
    ((|npEqKey| 'settab)
     (cond
       ((|npEqKey| 'backtab) (|npPush| (|pfNothing|))) ; never happens
       ((and (apply f nil)
              (or (|npEqKey| 'backtab) (|npMissing| 'backtab))))
         (|npPush| (|pfPile| (|npPop1|))))
       (t nil)))
    (t nil)))
```

7.0.351 defun npPileDefinitionlist

[npListAndRecover p191]
 [npDefinitionlist p195]
 [npPush p145]
 [pfAppend p257]
 [npPop1 p146]

— defun npPileDefinitionlist —

```
(defun |npPileDefinitionlist| ()
  (and
    (|npListAndRecover| #'|npDefinitionlist|)
    (|npPush| (|pfAppend| (|npPop1|)))))
```

—————

7.0.352 defun npListAndRecover

[trappoint p??]
 [npRecoverTrap p192]
 [syGeneralErrorHere p194]
 [npEqKey p147]
 [npEqPeek p154]
 [npNext p147]
 [npPop1 p146]
 [npPush p145]
 [\$inputStream p??]
 [\$stack p??]

— defun npListAndRecover —

```
(defun |npListAndRecover| (f)
  (let (found c done b savestack)
    (declare (special |$inputStream| |$stack|))
    (setq savestack |$stack|)
    (setq |$stack| nil)
    (setq c |$inputStream|)
    (do ()
      (done)
      (setq found (catch 'trappoint (apply f nil)))
      (cond
        ((eq found 'trapped)
         (setq |$inputStream| c)
         (|npRecoverTrap|))
        ((null found)
```



```

      (setq |$inputStream| c)
      (|syGeneralErrorHere|) (|npRecoverTrap|)))
(cond
  ((|npEqKey| 'backset) (setq c |$inputStream|))
  ((|npEqPeek| 'backtab) (setq done t))
  (t
    (setq |$inputStream| c)
    (|syGeneralErrorHere|)
    (|npRecoverTrap|)
    (cond
      ((|npEqPeek| 'backtab) (setq done t))
      (t
        (|npNext|)
        (setq c |$inputStream|))))))
(setq b (cons (|npPop1|) b)))
(setq |$stack| savestack)
(|npPush| (nreverse b)))

```

7.0.353 defun npRecoverTrap

```

[|npFirstTok| p145]
[|tokPosn| p415]
[|npMoveTo| p193]
[|syIgnoredFromTo| p193]
[|npPush| p145]
[|pfWrong| p298]
[|pfDocument| p248]
[|pfListOf| p247]
[$stok p??]

```

— defun npRecoverTrap —

```

(defun |npRecoverTrap| ()
  (let (pos2 pos1)
    (declare (special |$stok|))
    (|npFirstTok|)
    (setq pos1 (|tokPosn| |$stok|))
    (|npMoveTo| 0)
    (setq pos2 (|tokPosn| |$stok|))
    (|syIgnoredFromTo| pos1 pos2)
    (|npPush|
      (list (|pfWrong| (|pfDocument| (list "pile syntax error")))
        (|pfListOf| nil)))))

```

7.0.354 defun npMoveTo

[npEqPeek p154]
 [npNext p147]
 [npMoveTo p193]
 [npEqKey p147]
 [\$inputStream p??]

— defun npMoveTo —

```
(defun |npMoveTo| (|n|)
  (declare (special |$inputStream|))
  (cond
    ((null |$inputStream|) t)
    ((|npEqPeek| 'backtab)
     (cond
       ((eq1 |n| 0) t)
       (t (|npNext|) (|npMoveTo| (1- |n|)))))
    ((|npEqPeek| 'backset)
     (cond
       ((eq1 |n| 0) t)
       (t (|npNext|) (|npMoveTo| |n|))))
    ((|npEqKey| 'settab) (|npMoveTo| (+ |n| 1)))
    (t (|npNext|) (|npMoveTo| |n|))))
```

7.0.355 defun syIgnoredFromTo

[pfGlobalLinePosn p237]
 [ncSoftError p353]
 [FromTo p382]
 [From p382]
 [To p382]

— defun syIgnoredFromTo —

```
(defun |syIgnoredFromTo| (pos1 pos2)
  (cond
    ((equal (|pfGlobalLinePosn| pos1) (|pfGlobalLinePosn| pos2))
     (|ncSoftError| (|FromTo| pos1 pos2) 'S2CY0005 nil))
    (t
     (|ncSoftError| (|From| pos1) 'S2CY0003 nil)
     (|ncSoftError| (|To| pos2) 'S2CY0004 nil))))
```

7.0.356 defun syGeneralErrorHere

[sySpecificErrorHere p194]

— defun syGeneralErrorHere —

```
(defun |syGeneralErrorHere| ()
  (|sySpecificErrorHere| 'S2CY0002 nil))
```

7.0.357 defun sySpecificErrorHere

[sySpecificErrorAtToken p194]
[\$stok p??]

— defun sySpecificErrorHere —

```
(defun |sySpecificErrorHere| (key args)
  (declare (special |$stok|))
  (|sySpecificErrorAtToken| |$stok| key args))
```

7.0.358 defun sySpecificErrorAtToken

[ncSoftError p353]
[tokPosn p415]

— defun sySpecificErrorAtToken —

```
(defun |sySpecificErrorAtToken| (tok key args)
  (|ncSoftError| (|tokPosn| tok) key args))
```

7.0.359 defun npDefinitionlist

[npSemiListing p195]
 [npQualDef p147]

— **defun npDefinitionlist** —

```
(defun |npDefinitionlist| ()
  (|npSemiListing| #'|npQualDef|))
```

—————

7.0.360 defun npSemiListing

[npListofFun p223]
 [npSemiBackSet p195]
 [pfAppend p257]

— **defun npSemiListing** —

```
(defun |npSemiListing| (p)
  (|npListofFun| p #'|npSemiBackSet| #'|pfAppend|))
```

—————

7.0.361 defun npSemiBackSet

[npEqKey p147]

— **defun npSemiBackSet** —

```
(defun |npSemiBackSet| ()
  (and (|npEqKey| 'semicolon) (or (|npEqKey| 'backset) t)))
```

—————

7.0.362 defun npRule

[npEqKey p147]
 [npPP p211]
 [npSingleRule p196]

— **defun npRule** —

```
(defun |npRule| ()
  (and
    (|npEqKey| 'rule)
    (|npPP| #'|npSingleRule|)))
```

7.0.363 defun npSingleRule

```
[npQuiver p200]
[npDefTail p196]
[npTrap p214]
[npPush p145]
[pfRule p287]
[npPop2 p146]
[npPop1 p146]
```

— defun npSingleRule —

```
(defun |npSingleRule| ()
  (when (|npQuiver|)
    (or (|npDefTail|) (|npTrap|))
    (|npPush| (|pfRule| (|npPop2|) (|npPop1|))))))
```

7.0.364 defun npDefTail

```
[npEqKey p147]
[npDefinitionOrStatement p149]
```

— defun npDefTail —

```
(defun |npDefTail| ()
  (and
    (or (|npEqKey| 'def) (|npEqKey| 'mdef))
    (|npDefinitionOrStatement|)))
```

7.0.365 defun npDefaultValue

```
[npEqKey p147]
[npDefinitionOrStatement p149]
```

[npTrap p214]
 [npPush p145]
 [pfAdd p254]
 [pfNothing p247]
 [npPop1 p146]

— **defun npDefaultValue** —

```
(defun |npDefaultValue| ()
  (and
    (|npEqKey| 'default)
    (or (|npDefinitionOrStatement|) (|npTrap|))
    (|npPush| (list (|pfAdd| (|pfNothing|) (|npPop1|) (|pfNothing|))))))
```

7.0.366 **defun npWConditional**

[npConditional p197]
 [npPush p145]
 [pfTweakIf p292]
 [npPop1 p146]

— **defun npWConditional** —

```
(defun |npWConditional| (f)
  (when (|npConditional| f) (|npPush| (|pfTweakIf| (|npPop1|))))))
```

7.0.367 **defun npConditional**

[npEqKey p147]
 [npLogical p199]
 [npTrap p214]
 [npMissing p153]
 [npElse p198]

— **defun npConditional** —

```
(defun |npConditional| (f)
  (cond
    ((and (|npEqKey| 'IF)
          (or (|npLogical|) (|npTrap|))
```

```

      (or (|npEqKey| 'backset) t))
(cond
  ((|npEqKey| 'settab)
   (cond
    ((|npEqKey| 'then)
     (and (or (apply f nil) (|npTrap|))
          (|npElse| f)
          (|npEqKey| 'backtab)))
    (t (|npMissing| 'then))))
  ((|npEqKey| 'then)
   (and (or (apply f nil) (|npTrap|)) (|npElse| f)))
  (t (|npMissing| 'then))))
(t nil))

```

7.0.368 defun npElse

```

[npState p214]
[npBacksetElse p199]
[npTrap p214]
[npPush p145]
[pfIf p271]
[npPop3 p146]
[npPop2 p146]
[npPop1 p146]
[npRestore p154]
[pfIfThenOnly p272]

```

— defun npElse —

```

(defun |npElse| (f)
  (let (a)
    (setq a (|npState|))
    (cond
      ((|npBacksetElse|)
       (and
        (or (apply f nil) (|npTrap|))
        (|npPush| (|pfIf| (|npPop3|) (|npPop2|) (|npPop1|)))))
      (t
       (|npRestore| a)
       (|npPush| (|pfIfThenOnly| (|npPop2|) (|npPop1|)))))))

```

7.0.369 defun npBacksetElse

TPDHERE: Well this makes no sense. [npEqKey p147]

— defun npBacksetElse —

```
(defun |npBacksetElse| ()
  (if (|npEqKey| 'backset)
      (|npEqKey| 'else)
      (|npEqKey| 'else)))
```

—————

7.0.370 defun npLogical

[npLeftAssoc p208]

[npDisjand p199]

— defun npLogical —

```
(defun |npLogical| ()
  (|npLeftAssoc| '(or) #'|npDisjand|))
```

—————

7.0.371 defun npDisjand

[npLeftAssoc p208]

[npDiscrim p199]

— defun npDisjand —

```
(defun |npDisjand| ()
  (|npLeftAssoc| '(and) #'|npDiscrim|))
```

—————

7.0.372 defun npDiscrim

[npLeftAssoc p208]

[npQuiver p200]

— defun npDiscrim —


```
(defun |npDiscrim| ()
  (|npLeftAssoc| '(case has) #'|npQuiver|))
```

7.0.373 defun npQuiver

```
[npRightAssoc p208]
[npRelation p200]
```

— defun npQuiver —

```
(defun |npQuiver| ()
  (|npRightAssoc| '(arrow larrow) #'|npRelation|))
```

7.0.374 defun npRelation

```
[npLeftAssoc p208]
[npSynthetic p200]
```

— defun npRelation —

```
(defun |npRelation| ()
  (|npLeftAssoc| '(equal notequal lt le gt ge oangle cangle) #'|npSynthetic|))
```

7.0.375 defun npSynthetic

```
[npBy p201]
[npAmpersandFrom p204]
[npPush p145]
[pfApplication p255]
[npPop2 p146]
[npPop1 p146]
[pfInfApplication p273]
```

— defun npSynthetic —

```
(defun |npSynthetic| ()
```

```

(cond
  ((|npBy|)
   (lambda ()
    (loop
     (cond
      ((not (and (|npAmpersandFrom|)
                 (or (|npBy|)
                     (progn
                      (|npPush| (|pfApplication| (|npPop2|) (|npPop1|)))
                      nil))))
       (return nil))
      (t
       (|npPush| (|pfInfApplication| (|npPop2|) (|npPop2|) (|npPop1|)))))))
    t)
  (t nil)))

```

7.0.376 defun npBy

[npLeftAssoc p208]
 [npInterval p201]

— defun npBy —

```

(defun |npBy| ()
  (|npLeftAssoc| '(by) #'|npInterval|))

```

7.0.377 defun

[npArith p202]
 [npSegment p202]
 [npEqPeek p154]
 [npPush p145]
 [pfApplication p255]
 [npPop1 p146]
 [pfInfApplication p273]
 [npPop2 p146]

— defun npInterval —

```

(defun |npInterval| ()

```

```

(and
  (|npArith|)
  (or
    (and
      (|npSegment|)
      (or
        (and
          (|npEqPeek| 'bar)
          (|npPush| (|pfApplication| (|npPop1|) (|npPop1|))))
        (and
          (|npArith|)
          (|npPush| (|pfInfApplication| (|npPop2|) (|npPop2|) (|npPop1|))))
          (|npPush| (|pfApplication| (|npPop1|) (|npPop1|))))
      t)))

```

7.0.378 defun npSegment

```

[|npEqPeek| p154]
[|npPushId| p211]
[|npFromdom| p204]

```

— defun npSegment —

```

(defun |npSegment| ()
  (and (|npEqPeek| 'seg) (|npPushId|) (|npFromdom|)))

```

7.0.379 defun npArith

```

[|npLeftAssoc| p208]
[|npSum| p203]

```

— defun npArith —

```

(defun |npArith| ()
  (|npLeftAssoc| '(mod) #'|npSum|))

```

7.0.380 defun npSum

[npLeftAssoc p208]
[npTerm p203]

— **defun npSum** —

```
(defun |npSum| ()
  (|npLeftAssoc| '(plus minus) #'|npTerm|))
```

—————

7.0.381 defun npTerm

[npInfGeneric p209]
[npRemainder p203]
[npPush p145]
[pfApplication p255]
[npPop2 p146]
[npPop1 p146]

— **defun npTerm** —

```
(defun |npTerm| ()
  (or
    (and
      (|npInfGeneric| '(minus plus))
      (or
        (and (|npRemainder|) (|npPush| (|pfApplication| (|npPop2|) (|npPop1|))))
        t))
    (|npRemainder|)))
```

—————

7.0.382 defun npRemainder

[npLeftAssoc p208]
[npProduct p204]

— **defun npRemainder** —

```
(defun |npRemainder| ()
  (|npLeftAssoc| '(rem quo) #'|npProduct|))
```

—————

7.0.383 defun npProduct

[npLeftAssoc p208]
 [npPower p204]

— **defun npProduct** —

```
(defun |npProduct| ()
  (|npLeftAssoc|
   '(times slash backslash slashslash backslashbackslash
        slashbackslash backslashslash)
   #'|npPower|))
```

—————

7.0.384 defun npPower

[npRightAssoc p208]
 [npColon p219]

— **defun npPower** —

```
(defun |npPower| ()
  (|npRightAssoc| '(power carat) #'|npColon|))
```

—————

7.0.385 defun npAmpersandFrom

[npAmpersand p206]
 [npFromdom p204]

— **defun npAmpersandFrom** —

```
(defun |npAmpersandFrom| ()
  (and (|npAmpersand|) (|npFromdom|)))
```

—————

7.0.386 defun npFromdom

[npEqKey p147]
 [npApplication p164]

```
[npTrap p214]
[npFromdom1 p205]
[npPop1 p146]
[npPush p145]
[pfFromDom p269]
```

— **defun npFromdom** —

```
(defun |npFromdom| ()
  (or
    (and
      (|npEqKey| '$)
      (or (|npApplication|) (|npTrap|))
      (|npFromdom1| (|npPop1|))
      (|npPush| (|pfFromDom| (|npPop1|) (|npPop1|))))
    t))
```

—————

7.0.387 **defun npFromdom1**

```
[npEqKey p147]
[npApplication p164]
[npTrap p214]
[npFromdom1 p205]
[npPop1 p146]
[npPush p145]
[pfFromDom p269]
```

— **defun npFromdom1** —

```
(defun |npFromdom1| (c)
  (or
    (and
      (|npEqKey| '$)
      (or (|npApplication|) (|npTrap|))
      (|npFromdom1| (|npPop1|))
      (|npPush| (|pfFromDom| (|npPop1|) c)))
    (|npPush| c))))
```

—————

7.0.388 defun npAmpersand

```
[npEqKey p147]
[npName p206]
[npTrap p214]
```

— defun npAmpersand —

```
(defun |npAmpersand| ()
  (and
    (|npEqKey| 'ampersand)
    (or (|npName|) (|npTrap|))))
```

—————

7.0.389 defun npName

```
[npId p206]
[npSymbolVariable p207]
```

— defun npName —

```
(defun |npName| ()
  (or (|npId|) (|npSymbolVariable|)))
```

—————

7.0.390 defvar \$npPParg

— initvars —

```
(defvar |$npTokToNames| (list '~ '|#| '[] '{} '|[\|\\|]| '|{\|\\|}|))
```

—————

7.0.391 defun npId

```
[npPush p145]
[npNext p147]
[tokConstruct p413]
[tokPosn p415]
```

```

[npTokToNames p??]
[$ttok p??]
[$stok p??]

```

— **defun npId** —

```

(defun |npId| ()
  (declare (special |$npTokToNames| |$ttok| |$stok|))
  (cond
    ((eq (caar |$stok|) '|id|)
      (|npPush| |$stok|)
      (|npNext|))
    ((and (eq (caar |$stok|) '|key|) (member |$ttok| |$npTokToNames|))
      (|npPush| (|tokConstruct| '|id| |$ttok| (|tokPosn| |$stok|)))
      (|npNext|))
    (t nil)))

```

7.0.392 **defun npSymbolVariable**

```

[npState p214]
[npEqKey p147]
[npId p206]
[npPop1 p146]
[npPush p145]
[tokConstruct p413]
[tokPart p415]
[tokPosn p415]
[npRestore p154]

```

— **defun npSymbolVariable** —

```

(defun |npSymbolVariable| ()
  (let (a)
    (setq a (|npState|))
    (cond
      ((and (|npEqKey| 'backquote) (|npId|))
        (setq a (|npPop1|))
        (|npPush| (|tokConstruct| '|idsy| (|tokPart| a) (|tokPosn| a))))
      (t (|npRestore| a) nil))))

```

7.0.393 defun npRightAssoc

```
[npState p214]
[npInfGeneric p209]
[npRightAssoc p208]
[npPush p145]
[pfApplication p255]
[npPop2 p146]
[npPop1 p146]
[pfInfApplication p273]
[npRestore p154]
```

— **defun npRightAssoc** —

```
(defun |npRightAssoc| (o p)
  (let (a)
    (setq a (|npState|))
    (cond
      ((apply p nil)
        ((lambda ()
          (loop
            (cond
              ((not
                (and
                  (|npInfGeneric| o)
                  (or
                    (|npRightAssoc| o p)
                    (progn (|npPush| (|pfApplication| (|npPop2|) (|npPop1|))) nil))))
              (return nil))
            (t
              (|npPush| (|pfInfApplication| (|npPop2|) (|npPop2|) (|npPop1|))))))))
        t)
      (t
        (|npRestore| a)
        nil))))
```

—————

7.0.394 defun p o p o p o p = (((p o p) o p) o p)

```
p o p o p o p = (((p o p) o p) o p)
p o p o = (p o p) o
;npLeftAssoc(operations,parser)==
;  if APPLY(parser,nil)
;  then
;    while npInfGeneric(operations)
;      and (APPLY(parser,nil) or
```

```

;      (npPush pfApplication(npPop2(),npPop1());false))
;      repeat
;      npPush pfInfApplication(npPop2(),npPop2(),npPop1())
;      true
;      else false

```

```

[npInfGeneric p209]
[npPush p145]
[pfApplication p255]
[npPop2 p146]
[npPop1 p146]
[pfInfApplication p273]

```

— defun npLeftAssoc —

```

(defun |npLeftAssoc| (operations parser)
  (when (apply parser nil)
    ((lambda nil
      (loop
        (cond
          ((not
            (and
              (|npInfGeneric| operations)
              (or
                (apply parser nil)
                (progn (|npPush| (|pfApplication| (|npPop2|) (|npPop1|))) nil))))
            (return nil))
          (t
            (|npPush| (|pfInfApplication| (|npPop2|) (|npPop2|) (|npPop1|))))))))
    t))

```

7.0.395 defun npInfGeneric

```

[npDDInfKey p210]
[npEqKey p147]

```

— defun npInfGeneric —

```

(defun |npInfGeneric| (s)
  (and
    (|npDDInfKey| s)
    (or (|npEqKey| 'backset) t)))

```

7.0.396 defun npDDInfKey

```

[npInfKey p210]
[npState p214]
[npEqKey p147]
[npPush p145]
[pfSymb p253]
[npPop1 p146]
[tokPosn p415]
[npRestore p154]
[tokConstruct p413]
[tokPart p415]
[$stok p??]

```

— defun npDDInfKey —

```

(defun |npDDInfKey| (s)
  (let (b a)
    (declare (special |$stok|))
    (or
      (|npInfKey| s)
      (progn
        (setq a (|npState|))
        (setq b |$stok|)
        (cond
          ((and (|npEqKey| '||) (|npInfKey| s))
            (|npPush| (|pfSymb| (|npPop1|) (|tokPosn| b))))
          (t
            (|npRestore| a)
            (cond
              ((and (|npEqKey| 'backquote) (|npInfKey| s))
                (setq a (|npPop1|))
                (|npPush| (|tokConstruct| '|idsy| (|tokPart| a) (|tokPosn| a))))
              (t
                (|npRestore| a)
                nil))))))))))

```

—

7.0.397 defun npInfKey

```

[npPushId p211]
[$stok p??]
[$ttok p??]

```

— defun npInfKey —

```
(defun |npInfKey| (s)
  (declare (special |$ttok| |$stok|))
  (and (eq (caar |$stok|) '|key|) (member |$ttok| s) (|npPushId|)))
```

7.0.398 defun npPushId

```
[tokConstruct p413]
[tokPosn p415]
[npNext p147]
[$stack p??]
[$stok p??]
[$ttok p??]
```

— defun npPushId —

```
(defun |npPushId| ()
  (let (a)
    (declare (special |$stack| |$stok| |$ttok|))
    (setq a (get |$ttok| 'infgeneric))
    (when a (setq |$ttok| a))
    (setq |$stack|
      (cons (|tokConstruct| '|id| |$ttok| (|tokPosn| |$stok|)) |$stack|)
      (|npNext|)))
```

7.0.399 defvar \$npPParg

— initvars —

```
(defvar *npPParg* nil "rewrite npPP without flets, using global scoping")
```

7.0.400 defun npPP

This was rewritten by NAG to remove flet. [npParened p187]
 [npPPf p213]
 [npPileBracketed p190]

```
[npPPg p212]
[npPush p145]
[pfEnSequence p265]
[npPop1 p146]
[npPParg p211]
```

— **defun npPP** —

```
(defun |npPP| (f)
  (declare (special *npPParg*))
  (setq *npPParg* f)
  (or
    (|npParened| #'npPPf)
    (and (|npPileBracketed| #'npPPg) (|npPush| (|pfEnSequence| (|npPop1|))))
    (funcall f)))
```

—————

7.0.401 **defun npPPff**

```
[npPop1 p146]
[npPush p145]
[$npPParg p211]
```

— **defun npPPff** —

```
(defun npPPff ()
  (and (funcall *npPParg*) (|npPush| (list (|npPop1|)))))
```

—————

7.0.402 **defun npPPg**

```
[npListAndRecover p191]
[npPPf p213]
[npPush p145]
[pfAppend p257]
[npPop1 p146]
```

— **defun npPPg** —

```
(defun npPPg ()
  (and (|npListAndRecover| #'npPPf)
    (|npPush| (|pfAppend| (|npPop1|)))))
```

7.0.403 defun npPPf

[npSemiListing p195]
[npPPff p212]

— defun npPPf —

```
(defun npPPf ()
  (|npSemiListing| #'npPPff))
```

7.0.404 defun npEnclosed

[npEqKey p147]
[npPush p145]
[pfTuple p294]
[pfListOf p247]
[npMissingMate p217]
[pfEnSequence p265]
[npPop1 p146]
[\$stok p??]

— defun npEnclosed —

```
(defun |npEnclosed| (open close fn f)
  (let (a)
    (declare (special |$stok|))
    (setq a |$stok|)
    (when (|npEqKey| open)
      (cond
        ((|npEqKey| close)
         (|npPush| (funcall fn a (|pfTuple| (|pfListOf| NIL))))))
        ((and (apply f nil)
              (or (|npEqKey| close)
                  (|npMissingMate| close a)))
         (|npPush| (funcall fn a (|pfEnSequence| (|npPop1|))))))
        ('t nil))))
```

7.0.405 defun npState

```
[$stack p??]
[$inputStream p??]
```

— defun npState —

```
(defun |npState| ()
  (declare (special |$stack| |$inputStream|))
  (cons |$inputStream| |$stack|))
```

—————

7.0.406 defun npTrap

```
[trappoint p??]
[tokPosn p415]
[ncSoftError p353]
[$stok p??]
```

— defun npTrap —

```
(defun |npTrap| ()
  (declare (special |$stok|))
  (|ncSoftError| (|tokPosn| |$stok|) 'S2CY0002 nil)
  (throw 'trappoint 'trapped))
```

—————

7.0.407 defun npTrapForm

```
[trappoint p??]
[pfSourceStok p245]
[syGeneralErrorHere p194]
[ncSoftError p353]
[tokPosn p415]
```

— defun npTrapForm —

```
(defun |npTrapForm| (x)
  (let (a)
    (setq a (|pfSourceStok| x))
    (cond
      ((eq a '|NoToken|)
```

```

(|syGeneralErrorHere|)
(throw 'trappoint 'trapped))
(t
(|ncSoftError| (|tokPosn| a) 'S2CY0002 nil)
(throw 'trappoint 'trapped))))

```

7.0.408 defun npVariable

[npParenthesized p216]
 [npVariablelist p215]
 [npVariableName p215]
 [npPush p145]
 [pfListOf p247]
 [npPop1 p146]

— defun npVariable —

```

(defun |npVariable| ()
  (or
    (|npParenthesized| #'|npVariablelist|)
    (and (|npVariableName|) (|npPush| (|pfListOf| (list (|npPop1|)))))))

```

7.0.409 defun npVariablelist

[npListing p157]
 [npVariableName p215]

— defun npVariablelist —

```

(defun |npVariablelist| ()
  (|npListing| #'|npVariableName|))

```

7.0.410 defun npVariableName

[npName p206]
 [npDecl p216]

[npPush p145]
 [pfTyped p292]
 [npPop1 p146]
 [pfNothing p247]

— defun npVariableName —

```
(defun |npVariableName| ()
  (and
    (|npName|)
    (or (|npDecl|) (|npPush| (|pfTyped| (|npPop1|) (|pfNothing|))))))
```

—————

7.0.411 defun npDecl

[npEqKey p147]
 [npType p151]
 [npTrap p214]
 [npPush p145]
 [pfTyped p292]
 [npPop2 p146]
 [npPop1 p146]

— defun npDecl —

```
(defun |npDecl| ()
  (and
    (|npEqKey| 'colon)
    (or (|npType|) (|npTrap|))
    (|npPush| (|pfTyped| (|npPop2|) (|npPop1|)))))
```

—————

7.0.412 defun npParenthesized

[npParenthesize p217]

— defun npParenthesized —

```
(defun |npParenthesized| (f)
  (or (|npParenthesize| '(| ')| f) (|npParenthesize| '(\| '|\)| f)))
```

—————

7.0.413 defun npParenthesize

[npEqKey p147]
 [npMissingMate p217]
 [npPush p145]
 [\$stok p??]

— defun npParenthesize —

```
(defun |npParenthesize| (open close f)
  (let (a)
    (declare (special |$stok|))
    (setq a |$stok|)
    (cond
      ((|npEqKey| open)
       (cond
         ((and (apply f nil)
              (or (|npEqKey| close)
                  (|npMissingMate| close a))))
         t)
       ((|npEqKey| close) (|npPush| nil))
       (t (|npMissingMate| close a))))
    (t nil))))
```

7.0.414 defun npMissingMate

[ncSoftError p353]
 [tokPosn p415]
 [npMissing p153]

— defun npMissingMate —

```
(defun |npMissingMate| (close open)
  (|ncSoftError| (|tokPosn| open) 'S2CY0008 nil)
  (|npMissing| close))
```

7.0.415 defun npExit

[npBackTrack p150]
 [npAssign p218]

[npPileExit p218]

— **defun npExit** —

```
(defun |npExit| ()
  (|npBackTrack| #'|npAssign| 'exit #'|npPileExit|))
```

—————

7.0.416 **defun npPileExit**

[npAssign p218]
 [npEqKey p147]
 [npStatement p172]
 [npPush p145]
 [pfExit p265]
 [npPop2 p146]
 [npPop1 p146]

— **defun npPileExit** —

```
(defun |npPileExit| ()
  (and
    (|npAssign|)
    (or (|npEqKey| 'exit) (|npTrap|))
    (or (|npStatement|) (|npTrap|))
    (|npPush| (|pfExit| (|npPop2|) (|npPop1|))))))
```

—————

7.0.417 **defun npAssign**

[npBackTrack p150]
 [npMDEF p167]
 [npAssignment p219]

— **defun npAssign** —

```
(defun |npAssign| ()
  (|npBackTrack| #'|npMDEF| 'becomes #'|npAssignment|))
```

—————

7.0.418 defun npAssignment

[npAssignVariable p219]
 [npEqKey p147]
 [npTrap p214]
 [npGives p150]
 [npPush p145]
 [pfAssign p257]
 [npPop2 p146]
 [npPop1 p146]

— **defun npAssignment** —

```
(defun |npAssignment| ()
  (and
    (|npAssignVariable|)
    (or (|npEqKey| 'becomes) (|npTrap|))
    (or (|npGives|) (|npTrap|))
    (|npPush| (|pfAssign| (|npPop2|) (|npPop1|)))))
```

—————

7.0.419 defun npAssignVariable

[npColon p219]
 [npPush p145]
 [pfListOf p247]
 [npPop1 p146]

— **defun npAssignVariable** —

```
(defun |npAssignVariable| ()
  (and (|npColon|) (|npPush| (|pfListOf| (list (|npPop1|))))))
```

—————

7.0.420 defun npColon

[npTypified p220]
 [npAnyNo p164]
 [npTagged p220]

— **defun npColon** —

```
(defun |npColon| ()
  (and (|npTypified|) (|npAnyNo| #'|npTagged|)))
```

7.0.421 defun npTagged

```
[npTypedForm1 p220]
[pfTagged p290]
```

— defun npTagged —

```
(defun |npTagged| ()
  (|npTypedForm1| 'colon #'|pfTagged|))
```

7.0.422 defun npTypedForm1

```
[npEqKey p147]
[npType p151]
[npTrap p214]
[npPush p145]
[npPop2 p146]
[npPop1 p146]
```

— defun npTypedForm1 —

```
(defun |npTypedForm1| (sy fn)
  (and
    (|npEqKey| sy)
    (or (|npType|) (|npTrap|))
    (|npPush| (funcall fn (|npPop2|) (|npPop1|)))))
```

7.0.423 defun npTypified

```
[npApplication p164]
[npAnyNo p164]
[npTypeStyle p221]
```

— defun npTypified —

```
(defun |npTypified| ()
  (and (|npApplication|) (|npAnyNo| #'|npTypeStyle|)))
```

7.0.424 defun npTypeStyle

```
[npCoerceTo p222]
[npRestrict p222]
[npPretend p221]
[npColonQuery p221]
```

— defun npTypeStyle —

```
(defun |npTypeStyle| ()
  (or (|npCoerceTo|) (|npRestrict|) (|npPretend|) (|npColonQuery|)))
```

7.0.425 defun npPretend

```
[npTypedForm p222]
[pfPretend p283]
```

— defun npPretend —

```
(defun |npPretend| ()
  (|npTypedForm| 'pretend #'|pfPretend|))
```

7.0.426 defun npColonQuery

```
[npTypedForm p222]
[pfRetractTo p286]
```

— defun npColonQuery —

```
(defun |npColonQuery| ()
  (|npTypedForm| 'atat #'|pfRetractTo|))
```

7.0.427 defun npCoerceTo

[npTypedForm p222]
 [pfCoerceto p261]

— **defun npCoerceTo** —

```
(defun |npCoerceTo| ()
  (|npTypedForm| 'coerce #'|pfCoerceto|))
```

—————

7.0.428 defun npTypedForm

[npEqKey p147]
 [npApplication p164]
 [npTrap p214]
 [npPush p145]
 [npPop2 p146]
 [npPop1 p146]

— **defun npTypedForm** —

```
(defun |npTypedForm| (sy fn)
  (and
    (|npEqKey| sy)
    (or (|npApplication|) (|npTrap|))
    (|npPush| (funcall fn (|npPop2|) (|npPop1|))))))
```

—————

7.0.429 defun npRestrict

[npTypedForm p222]
 [pfRestrict p285]

— **defun npRestrict** —

```
(defun |npRestrict| ()
  (|npTypedForm| 'at #'|pfRestrict|))
```

—————

7.0.430 defun npListofFun

```
[npTrap p214]
[npPush p145]
[npPop3 p146]
[npPop2 p146]
[npPop1 p146]
[$stack p??]
```

— **defun npListofFun** —

```
(defun |npListofFun| (f h g)
  (let (a)
    (declare (special |$stack|))
    (cond
      ((apply f nil)
       (cond
         ((and (apply h nil) (or (apply f nil) (|npTrap|))))
         (setq a |$stack|)
         (setq |$stack| nil)
         (do ()
              ((not (and (apply h nil)
                        (or (apply f nil) (|npTrap|)))))
              (setq |$stack| (cons (nreverse |$stack|) a))
              (|npPush| (funcall g (cons (|npPop3|) (cons (|npPop2|) (|npPop1|))))))
            (t t)))
      (t nil))))
```

—————

7.1 Macro handling**7.1.1 defun phMacro**

TPDHERE: The pform function has a leading percent sign. fix this

```
carrier[ptree,...] -> carrier[ptree, ptreePremacro,...]
```

```
[ncEltQ p418]
[ncPutQ p418]
[macroExpanded p224]
[intSayKeyedMsg p67]
[pform p??]
```

— **defun phMacro** —


```
(defun |phMacro| (carrier)
  (let (ptree)
    (setq ptree (|ncEltQ| carrier '|ptree|))
    (|ncPutQ| carrier '|ptreePremacro| ptree)
    (setq ptree (|macroExpanded| ptree))
    (|ncPutQ| carrier '|ptree| ptree)
    'ok))
```

7.1.2 defun macroExpanded

\$macActive is a list of the bodies being expanded. \$posActive is a list of the parse forms where the bodies came from. [macExpand p224]

```
[$posActive p??]
[$macActive p??]
```

— defun macroExpanded —

```
(defun |macroExpanded| (pf)
  (let (|$posActive| |$macActive|)
    (declare (special |$posActive| |$macActive|))
    (setq |$macActive| nil)
    (setq |$posActive| nil)
    (|macExpand| pf)))
```

7.1.3 defun macExpand

```
[pfWhere? p296]
[macWhere p230]
[pfLambda? p275]
[macLambda p230]
[pfMacro? p279]
[macMacro p231]
[pfId? p248]
[macId p229]
[pfApplication? p257]
[macApplication p225]
[pfMapParts p238]
[macExpand p224]
```

— defun macExpand —

```
(defun |macExpand| (pf)
  (cond
    ((|pfWhere?| pf)      (|macWhere| pf))
    ((|pfLambda?| pf)     (|macLambda| pf))
    ((|pfMacro?| pf)      (|macMacro| pf))
    ((|pfId?| pf)         (|macId| pf))
    ((|pfApplication?| pf) (|macApplication| pf))
    (t                    (|pfMapParts| #'|macExpand| pf))))
```

7.1.4 defun macApplication

[pfMapParts p238]
 [macExpand p224]
 [pfApplicationOp p256]
 [pfMLambda? p280]
 [pf0ApplicationArgs p239]
 [mac0MLambdaApply p225]
 [\$pfMacros p100]

— defun macApplication —

```
(defun |macApplication| (pf)
  (let (args op)
    (declare (special |$pfMacros|))
    (setq pf (|pfMapParts| #'|macExpand| pf))
    (setq op (|pfApplicationOp| pf))
    (cond
      ((null (|pfMLambda?| op)) pf)
      (t
       (setq args (|pf0ApplicationArgs| pf))
       (|mac0MLambdaApply| op args pf |$pfMacros|))))
```

7.1.5 defun mac0MLambdaApply

TPDHERE: The pform function has a leading percent sign. fix this [pf0MLambdaArgs p280]

[pfMLambdaBody p281]
 [pfSourcePosition p240]
 [ncHardError p354]
 [pfId? p248]

```
[pform p??]
[mac0Define p232]
[mac0ExpandBody p226]
[$pfMacros p100]
[$posActive p??]
[$macActive p??]
```

— **defun mac0MLambdaApply** —

```
(defun |mac0MLambdaApply| (mlambda args opf |$pfMacros|)
  (declare (special |$pfMacros|))
  (let (pos body params)
    (declare (special |$posActive| |$macActive|))
    (setq params (|pf0MLambdaArgs| mlambda))
    (setq body (|pfMLambdaBody| mlambda))
    (cond
      ((not (eql (length args) (length params)))
        (setq pos (|pfSourcePosition| opf))
        (|incHardError| pos 'S2CM0003 (list (length params) (length args))))
      (t
        ((lambda (parms p arrgs a) ; for p in params for a in args repeat
          (loop
            (cond
              ((or (atom parms)
                (progn (setq p (car parms)) nil)
                (atom arrgs)
                (progn (setq a (CAR arrgs)) nil))
              (return nil))
            (t
              (cond
                ((null (|pfId?| p))
                 (setq pos (|pfSourcePosition| opf))
                 (|incHardError| pos 'S2CM0004 (list (|%pform| p))))
                (t
                 (|mac0Define| (|pfIdSymbol| p) '|mparam| a))))))
          (setq parms (cdr parms))
          (setq arrgs (cdr arrgs))))
        params nil args nil)
      (|mac0ExpandBody| body opf |$macActive| |$posActive|))))
```

—————

7.1.6 defun mac0ExpandBody

```
[pfSourcePosition p240]
[mac0InfiniteExpansion p227]
[macExpand p224]
```

[\$posActive p??]
 [\$macActive p??]

— defun mac0ExpandBody —

```
(defun |mac0ExpandBody| (body opf |$macActive| |$posActive|)
  (declare (special |$macActive| |$posActive|))
  (let (posn pf)
    (cond
      ((member body |$macActive|)
       (setq pf (cadr |$posActive|))
       (setq posn (|pfSourcePosition| pf))
       (|mac0InfiniteExpansion| posn body |$macActive|))
      (t
       (setq |$macActive| (cons body |$macActive|))
       (setq |$posActive| (cons opf |$posActive|))
       (|macExpand| body))))))
```

7.1.7 defun mac0InfiniteExpansion

TPDHERE: The pform function has a leading percent sign. fix this [mac0InfiniteExpansion,name p228]
 [ncSoftError p353]
 [pform p??]

— defun mac0InfiniteExpansion —

```
(defun |mac0InfiniteExpansion| (posn body active)
  (let (rnames fname tmp1 blist result)
    (setq blist (cons body active))
    (setq tmp1 (mapcar #'|mac0InfiniteExpansion,name| blist))
    (setq fname (car tmp1)) ;[fname, :rnames] := [name b for b in blist]
    (setq rnames (cdr tmp1))
    (|ncSoftError| posn 'S2CM0005
      (list
        (dolist (n (reverse rnames) (nreverse result))
          (setq result (append (reverse (list n "==">)) result)))
        fname (|%pform| body)))
    body))
```

7.1.8 defun mac0InfiniteExpansion,name

[mac0GetName p228]
[pname p1001]

— defun mac0InfiniteExpansion,name 0 —

```
(defun |mac0InfiniteExpansion,name| (b)
  (let (st sy got)
    (setq got (|mac0GetName| b))
    (cond
      ((null got) "???" )
      (t
       (setq sy (car got))
       (setq st (cadr got))
       (if (eq st '|mlambda|)
           (concat (pname sy) "(...)")
           (pname sy))))))
```

—————

7.1.9 defun mac0GetName

Returns [state, body] or NIL. Returns [sy, state] or NIL. [pfMLambdaBody p281]
[\$pfMacros p100]

— defun mac0GetName —

```
(defun |mac0GetName| (body)
  (let (bd tmp1 st tmp2 sy name)
    (declare (special |$pfMacros|))
    ; for [sy,st,bd] in $pfMacros while not name repeat
    ((lambda (macros tmp1list)
      (loop
        (cond
          ((or (atom macros)
               (progn (setq tmp1list (car macros)) nil)
                      name)
           (return nil))
          (t
           (and (consp tmp1list)
                (progn
                  (setq sy (car tmp1list))
                  (setq tmp2 (cdr tmp1list))
                  (and (consp tmp2)
                       (progn
                        (setq st (car tmp2))

```

```

      (setq tmp1 (cdr tmp2))
      (and (consp tmp1)
           (eq (cdr tmp1) nil)
           (progn
            (setq bd (car tmp1))
            t))))
    (progn
     (when (eq st '|mlambda|) (setq bd (|pfMLambdaBody| bd)))
     (when (eq bd body) (setq name (list sy st))))))
    (setq macros (cdr macros)))
  |$pfMacros| nil)
  name))

```

7.1.10 defun macId

```

[pfIdSymbol p249]
[mac0Get p230]
[pfCopyWithPos p238]
[pfSourcePosition p240]
[mac0ExpandBody p226]
[$posActive p??]
[$macActive p??]

```

— defun macId —

```

(defun |macId| (pf)
  (let (body state got sy)
    (declare (special |$posActive| |$macActive|))
    (setq sy (|pfIdSymbol| pf))
    (cond
     ((null (setq got (|mac0Get| sy))) pf)
     (t
      (setq state (car got))
      (setq body (cadr got))
      (cond
       ((eq state '|mparam|) body)
       ((eq state '|mlambda|) (|pfCopyWithPos| body (|pfSourcePosition| pf)))
       (t
        (|pfCopyWithPos|
         (|mac0ExpandBody| body pf |$macActive| |$posActive|)
         (|pfSourcePosition| pf)))))))

```

7.1.11 defun mac0Get

```
[ifcdr p??]
[$pfMacros p100]

— defun mac0Get —

(defun |mac0Get| (sy)
  (declare (special |$pfMacros|))
  (ifcdr (assoc sy |$pfMacros|)))
```

7.1.12 defun macWhere

```
[macWhere,mac p230]
[$pfMacros p100]

— defun macWhere —

(defun |macWhere| (pf)
  (declare (special |$pfMacros|))
  (|macWhere,mac| pf |$pfMacros|))
```

7.1.13 defun macWhere,mac

```
[pfMapParts p238]
[macExpand p224]
[$pfMacros p100]

— defun macWhere,mac —

(defun |macWhere,mac| (pf |$pfMacros|)
  (declare (special |$pfMacros|))
  (|pfMapParts| #'|macExpand| pf))
```

7.1.14 defun macLambda

```
[macLambda,mac p231]
[$pfMacros p100]
```

— **defun macLambda** —

```
(defun |macLambda| (pf)
  (declare (special |$pfMacros|))
  (|macLambda,mac| pf |$pfMacros|))
```

—————

7.1.15 **defun macLambda,mac**

[pfMapParts p238]
 [macExpand p224]
 [\$pfMacros p100]

— **defun macLambda,mac** —

```
(defun |macLambda,mac| (pf |$pfMacros|)
  (declare (special |$pfMacros|))
  (|pfMapParts| #'|macExpand| pf))
```

—————

7.1.16 **defun Add appropriate definition the a Macro pform**

This function adds the appropriate definition and returns the original Macro pform. **TPDHERE:**
The pform function has a leading percent sign. fix this [pfMacroLhs p279]

[pfMacroRhs p279]
 [pfId? p248]
 [ncSoftError p353]
 [pfSourcePosition p240]
 [pfIdSymbol p249]
 [mac0Define p232]
 [pform p??]
 [pfMLambda? p280]
 [macSubstituteOuter p232]
 [pfNothing? p247]
 [pfMacro p279]
 [pfNothing p247]

— **defun macMacro** —

```
(defun |macMacro| (pf)
  (let (sy rhs lhs)
```



```

(setq lhs (|pfMacroLhs| pf))
(setq rhs (|pfMacroRhs| pf))
(cond
  ((null (|pfId?| lhs))
   (|ncSoftError| (|pfSourcePosition| lhs) 'S2CM0001 (list (|%pform| lhs)))
   pf)
  (t
   (setq sy (|pfIdSymbol| lhs))
   (|mac0Define| sy
    (cond
      ((|pfMLambda?| rhs) '|mlambda|)
      (t '|mbody|))
    (|macSubstituteOuter| rhs))
   (cond
     ((|pfNothing?| rhs) pf)
     (t (|pfMacro| lhs (|pfNothing|))))))))

```

7.1.17 defun Add a macro to the global pfMacros list

[*\$pfMacros* p100]

— defun *mac0Define* 0 —

```

(defun |mac0Define| (sy state body)
  (declare (special |$pfMacros|))
  (setq |$pfMacros| (cons (list sy state body) |$pfMacros|)))

```

7.1.18 defun *macSubstituteOuter*

[*mac0SubstituteOuter* p233]
 [*macLambdaParameterHandling* p233]

— defun *macSubstituteOuter* —

```

(defun |macSubstituteOuter| (pform)
  (|mac0SubstituteOuter| (|macLambdaParameterHandling| nil pform) pform))

```

7.1.19 defun mac0SubstituteOuter

[pfId? p248]
 [macSubstituteId p234]
 [pfLeaf? p249]
 [pfLambda? p275]
 [macLambdaParameterHandling p233]
 [mac0SubstituteOuter p233]
 [pfParts p251]

— **defun mac0SubstituteOuter** —

```
(defun |mac0SubstituteOuter| (replist pform)
  (let (tmplist)
    (cond
      ((|pfId?| pform) (|macSubstituteId| replist pform))
      ((|pfLeaf?| pform) pform)
      ((|pfLambda?| pform)
       (setq tmplist (|macLambdaParameterHandling| replist pform))
       (dolist (p (|pfParts| pform)) (|mac0SubstituteOuter| tmplist p))
       pform)
      (t
       (dolist (p (|pfParts| pform)) (|mac0SubstituteOuter| replist p))
       pform))))
```

7.1.20 defun macLambdaParameterHandling

[pfLeaf? p249]
 [pfLambda? p275]
 [pfTypeId p293]
 [pf0LambdaArgs p276]
 [pfIdSymbol p249]
 [pfMLambda? p280]
 [pf0MLambdaArgs p280]
 [pfLeaf p249]
 [pfAbSynOp p414]
 [pfLeafPosition p250]
 [pfParts p251]
 [macLambdaParameterHandling p233]

— **defun macLambdaParameterHandling** —

```
(defun |macLambdaParameterHandling| (replist pform)
  (let (parlist symlist result)
```

```

(cond
  ((|pfLeaf?| pform) nil)
  ((|pfLambda?| pform) ; remove ( identifier . replacement ) from assoclist
   (setq parlist (mapcar #'|pfTypedId| (|pf0LambdaArgs| pform)))
   (setq symlist (mapcar #'|pfIdSymbol| parlist))
   (dolist (par symlist)
     (setq replist
       (let ((pr (assoc par replist :test #'equal)))
         (if pr (remove par replist :test #'equal) 1))))
     replist)
  ((|pfMLambda?| pform) ;construct assoclist ( identifier . replacement )
   (setq parlist (|pf0MLambdaArgs| pform)) ; extract parameter list
   (dolist (par parlist (nreverse result))
     (push
       (cons (|pfIdSymbol| par)
              (|pfLeaf| (|pfAbSynOp| par) (gensym) (|pfLeafPosition| par)))
       result)))
  (t
   (dolist (p (|pfParts| pform))
     (|macLambdaParameterHandling| replist p))))))

```

7.1.21 defun macSubstituteId

[pfIdSymbol p249]

— defun macSubstituteId —

```

(defun |macSubstituteId| (repllist pform)
  (let (ex)
    (setq ex (assoc (|pfIdSymbol| pform) replist :test #'eq))
    (cond
      (ex
       (rplpair pform (cdr ex))
       pform)
      (t pform))))

```

Chapter 8

Pftrees

8.1 Abstract Syntax Trees Overview

Th functions create and examine abstract syntax trees. These are called pforms, for short.

The pform data structure

- Leaves: [hd, tok, pos] where pos is optional
- Trees: [hd, tree, tree, ...]
- hd is either an id or (id . alist)

The leaves are:

char	:=	('char expr position)
Document	:=	('Document expr position)
error	:=	('error expr position)
expression	:=	('expression expr position)
float	:=	('float expr position)
id	:=	('id expr position)
idsy	:=	('idsy expr position)
integer	:=	('integer expr position)
string	:=	('string expr position)
symbol	:=	('symbol expr position)

The special nodes:

ListOf	:=	('listOf items)
Nothing	:=	('nothing)
SemiColon	:=	('SemiColon (Body: Expr))

The expression nodes:

Add	:= ('Add (Base: [Typed], Addin: Expr))
And	:= ('And left right)
Application	:= ('Application (Op: Expr, Arg: Expr))
Assign	:= ('Assign (LhsItems: [AssLhs], Rhs: Expr))
Attribute	:= ('Attribute (Expr: Primary))
Break	:= ('Break (From: ? Id))
Coerceto	:= ('Coerceto (Expr: Expr, Type: Type))
Collect	:= ('Collect (Body: Expr, Iterators: [Iterator]))
ComDefinition	:= ('ComDefinition (Doc: Document, Def: Definition))
DeclPart	
Definition	:= ('Definition (LhsItems: [Typed], Rhs: Expr))
DefinitionSequence	:= (Args: [DeclPart])
Do	:= ('Do (Body: Expr))
Document	:= ('Document strings)
DWhere	:= ('DWhere (Context: [DeclPart], Expr: [DeclPart]))
EnSequence	:=
Exit	:= ('Exit (Cond: ? Expr, Expr: ? Expr))
Export	:= ('Export (Items: [Typed]))
Forin	:= ('Forin (Lhs: [AssLhs], Whole: Expr))
Free	:= ('Free (Items: [Typed]))
Fromdom	:= ('Fromdom (What: Id, Domain: Type))
Hide	:= ('hide, arg)
If	:= ('If (Cond: Expr, Then: Expr, Else: ? Expr))
Import	:= ('Import (Items: [QualType]))
Inline	:= ('Inline (Items: [QualType]))
Iterate	:= ('Iterate (From: ? Id))
Lambda	:= ('Lambda (Args: [Typed], Rets: ReturnedTyped, Body: Expr))
Literal	
Local	:= ('Local (Items: [Typed]))
Loop	:= ('Loop (Iterators: [Iterator]))
Macro	:= ('Macro (Lhs: Id, Rhs: ExprorNot))
MLambda	:= ('MLambda (Args: [Id], Body: Expr))
Not	:= ('Not arg)
Novalue	:= ('Novalue (Expr: Expr))
Or	:= ('Or left right)
Pretend	:= ('Pretend (Expr: Expr, Type: Type))
QualType	:= ('QualType (Type: Type, Qual: ? Type))
Restrict	:= ('Restrict (Expr: Expr, Type: Type))
Retract	:= ('RetractTo (Expr: Expr, Type: Type))
Return	:= ('Return (Expr: ? Expr, From: ? Id))
ReturnTyped	:= ('returntyped (type body))
Rule	:= ('Rule (lhsitems, rhsitems))
Sequence	:= ('Sequence (Args: [Expr]))
Suchthat	:= ('Suchthat (Cond: Expr))
Symb	:= if leaf then symbol else expression
Tagged	:= ('Tagged (Tag: Expr, Expr: Expr))
TLambda	:= ('TLambda (Args: [Typed], Rets: ReturnedTyped Type, Body: Expr))
Tuple	:= ('Tuple (Parts: [Expr]))
Typed	:= ('Typed (Id: Id, Type: ? Type))
Typing	:= ('Typing (Items: [Typed]))
Until	:= ('Until (Cond: Expr)) NOT USED
WDeclare	:= ('WDeclare (Signature: Typed, Doc: ? Document))
Where	:= ('Where (Context: [DeclPart], Expr: Expr))
While	:= ('While (Cond: Expr))
With	:= ('With (Base: [Typed], Within: [WithPart]))
Wif	:= ('Wif (Cond: Primary, Then: [WithPart], Else: [WithPart]))
Wrong	:= ('Wrong (Why: Document, Rubble: [Expr]))

Special cases of expression nodes are:

- Application. The Op parameter is one of `and`, `or`, `λ`, `|`, `{}`, `[]`, `{|}|`, `[|]|`
- DeclPart. The comment is attached to all signatutres in Typing, Import, Definition, Sequence, DWhere, Macro nodes
- EnSequence. This is either a Tuple or Sequence depending on the argument
- Literal. One of integer symbol expression one zero char string float of the form ('expression expr position)

8.2 Structure handlers

8.2.1 defun pfGlobalLinePosn

[poGlobalLinePosn p73]

— defun pfGlobalLinePosn —

```
(defun |pfGlobalLinePosn| (posn)
  (|poGlobalLinePosn| posn))
```

—————

8.2.2 defun pfCharPosn

[poCharPosn p379]

— defun pfCharPosn —

```
(defun |pfCharPosn| (posn)
  (|poCharPosn| posn))
```

—————

8.2.3 defun pfLinePosn

[poLinePosn p363]

— defun pfLinePosn —

```
(defun |pfLinePosn| (posn)
  (|poLinePosn| posn))
```

8.2.4 defun pfFileName

```
[poFileName p362]
```

— defun pfFileName —

```
(defun |pfFileName| (posn)
  (|poFileName| posn))
```

8.2.5 defun pfCopyWithPos

```
[pfLeaf? p249]
[pfLeaf p249]
[pfAbSynOp p414]
[tokPart p415]
[pfTree p254]
[pfParts p251]
[pfCopyWithPos p238]
```

— defun pfCopyWithPos —

```
(defun |pfCopyWithPos| (pform pos)
  (if (|pfLeaf?| pform)
      (|pfLeaf| (|pfAbSynOp| pform) (|tokPart| pform) pos)
      (|pfTree| (|pfAbSynOp| pform)
                (loop for p in (|pfParts| pform)
                      collect (|pfCopyWithPos| p pos))))))
```

8.2.6 defun pfMapParts

```
[pfLeaf? p249]
[pfParts p251]
[pfTree p254]
```

[pfAbSynOp p414]

— **defun pfMapParts** —

```
(defun |pfMapParts| (f pform)
  (let (same parts1 parts0)
    (if (|pfLeaf?| pform)
        pform
        (progn
         (setq parts0 (|pfParts| pform))
         (setq parts1 (loop for p in parts0 collect (funcall f p)))
         (if (reduce #'(lambda (u v) (and u v)) (mapcar #'eq parts0 parts1))
             pform
             (|pfTree| (|pfAbSynOp| pform) parts1))))))
```

—

8.2.7 defun pf0ApplicationArgs

[pf0FlattenSyntacticTuple p239]

[pfApplicationArg p256]

— **defun pf0ApplicationArgs** —

```
(defun |pf0ApplicationArgs| (pform)
  (|pf0FlattenSyntacticTuple| (|pfApplicationArg| pform)))
```

—

8.2.8 defun pf0FlattenSyntacticTuple

[pfTuple? p294]

[pf0FlattenSyntacticTuple p239]

[pf0TupleParts p295]

— **defun pf0FlattenSyntacticTuple** —

```
(defun |pf0FlattenSyntacticTuple| (pform)
  (if (null (|pfTuple?| pform))
      (list pform)
      ; [:pf0FlattenSyntacticTuple p for p in pf0TupleParts pform]
      ((lambda (arg0 arg1 p)
        (loop
         (cond
          ((or (atom arg1) (progn (setq p (car arg1)) nil))
```



```

      (return (nreverse arg0)))
    (t
      (setq arg0 (append (reverse (|pf0FlattenSyntacticTuple| p)) arg0)))
      (setq arg1 (cdr arg1)))
    nil (|pf0TupleParts| pform) nil)))

```

8.2.9 defun pfSourcePosition

[pfLeaf? p249]
 [pfLeafPosition p250]
 [poNoPosition? p415]
 [pfSourcePosition p240]
 [pfParts p251]
 [\$nopus p28]

— defun pfSourcePosition —

```

(defun |pfSourcePosition| (form)
  (let (pos)
    (declare (special |$nopus|))
    (cond
      ((|pfLeaf?| form) (|pfLeafPosition| form))
      (t
        (setq pos |$nopus|)
        ((lambda (theparts p) ; for p in parts while poNoPosition? pos repeat
          (loop
            (cond
              ((or (atom theparts)
                 (progn (setq p (car theparts)) nil)
                 (not (|poNoPosition?| pos))))
               (return nil))
              (t (setq pos (|pfSourcePosition| p))))
            (setq theparts (cdr theparts))))
          (|pfParts| form) nil)
        pos))))

```

8.2.10 defun Convert a Sequence node to a list

[pfSequence? p289]
 [pfSequenceArgs p289]
 [pfListOf p247]

— defun pfSequenceToList —

```
(defun |pfSequenceToList| (x)
  (if (|pfSequence?| x)
      (|pfSequenceArgs| x)
      (|pfListOf| (list x))))
```

—————

8.2.11 defun pfSpread

[pfTyped p292]

— defun pfSpread —

```
(defun |pfSpread| (arg1 arg2)
  (mapcar #'(lambda (i) (|pfTyped| i arg2)) arg1))
```

—————

8.2.12 defun Deconstruct nodes to lists

```
[pfTagged? p291]
[pfTaggedExpr p291]
[pfNothing p247]
[pfTaggedTag p291]
[pfId? p248]
[pfListOf p247]
[pfTyped p292]
[pfCollect1? p244]
[pfCollectVariable1 p244]
[pfTuple? p294]
[pf0TupleParts p295]
[pfTaggedToTyped p291]
[pfDefinition? p263]
[pfApplication? p257]
[pfFlattenApp p243]
[pfTaggedToTyped1 p246]
[pfTransformArg p246]
[npTrapForm p214]
```

— defun pfCheckItOut —

```

(defun |pfCheckItOut| (x)
  (let (args op ls form rt result)
    (if (|pfTagged?| x)
      (setq rt (|pfTaggedExpr| x))
      (setq rt (|pfNothing|)))
    (if (|pfTagged?| x)
      (setq form (|pfTaggedTag| x))
      (setq form x))
    (cond
      ((|pfId?| form)
       (list (|pfListOf| (list (|pfTyped| form rt))) nil rt))
      ((|pfCollect1?| form)
       (list (|pfListOf| (list (|pfCollectVariable1| form))) nil rt))
      ((|pfTuple?| form)
       (list (|pfListOf|
              (dolist (part (|pf0TupleParts| form) (nreverse result))
                (push (|pfTaggedToTyped| part) result)))
              nil rt))
      ((|pfDefinition?| form)
       (list (|pfListOf| (list (|pfTyped| form (|pfNothing|)))) nil rt))
      ((|pfApplication?| form)
       (setq ls (|pfFlattenApp| form))
       (setq op (|pfTaggedToTyped1| (car ls)))
       (setq args
        (dolist (part (cdr ls) (nreverse result))
          (push (|pfTransformArg| part) result)))
       (list (|pfListOf| (list op)) args rt))
      (t (|npTrapForm| form)))))

```

8.2.13 defun pfCheckMacroOut

[pfId? p248]
 [pfApplication? p257]
 [pfFlattenApp p243]
 [pfCheckId p243]
 [pfCheckArg p243]
 [npTrapForm p214]

— defun pfCheckMacroOut —

```

(defun |pfCheckMacroOut| (form)
  (let (args op ls)
    (cond
      ((|pfId?| form) (list form nil))
      ((|pfApplication?| form)

```

```

(setq ls (|pfFlattenApp| form))
(setq op (|pfCheckId| (car ls)))
(setq args (mapcar #'|pfCheckArg| (cdr ls)))
(list op args))
(t (|npTrapForm| form))))

```

8.2.14 defun pfCheckArg

[pfTuple? p294]
 [pf0TupleParts p295]
 [pfListOf p247]
 [pfCheckId p243]

— defun pfCheckArg —

```

(defun |pfCheckArg| (args)
  (let (arg1)
    (if (|pfTuple?| args)
        (setq arg1 (|pf0TupleParts| args))
        (setq arg1 (list args)))
    (|pfListOf| (mapcar #'|pfCheckId| arg1))))

```

8.2.15 defun pfCheckId

[pfId? p248]
 [npTrapForm p214]

— defun pfCheckId —

```

(defun |pfCheckId| (form)
  (if (null (|pfId?| form))
      (|npTrapForm| form)
      form))

```

8.2.16 defun pfFlattenApp

[pfApplication? p257]
 [pfCollect1? p244]

[pfFlattenApp p243]
 [pfApplicationOp p256]
 [pfApplicationArg p256]

— **defun pfFlattenApp** —

```
(defun |pfFlattenApp| (x)
  (cond
    ((|pfApplication?| x)
     (cond
       ((|pfCollect1?| x) (LIST x))
       (t
        (append (|pfFlattenApp| (|pfApplicationOp| x))
                  (|pfFlattenApp| (|pfApplicationArg| x))))))
    (t (list x))))
```

8.2.17 defun pfCollect1?

[pfApplication? p257]
 [pfApplicationOp p256]
 [pfId? p248]
 [pfIdSymbol p249]

— **defun pfCollect1?** —

```
(defun |pfCollect1?| (x)
  (let (a)
    (when (|pfApplication?| x)
      (setq a (|pfApplicationOp| x))
      (when (|pfId?| a) (eq (|pfIdSymbol| a) '|\\|))))))
```

8.2.18 defun pfCollectVariable1

[pfApplicationArg p256]
 [pf0TupleParts p295]
 [pfTaggedToTyped p291]
 [pfTyped p292]
 [pfSuch p246]
 [pfTypedId p293]
 [pfTypedType p293]

— **defun pfCollectVariable1** —

```
(defun |pfCollectVariable1| (x)
  (let (id var a)
    (setq a (|pfApplicationArg| x))
    (setq var (car (|pf0TupleParts| a)))
    (setq id (|pfTaggedToTyped| var))
    (|pfTyped|
     (|pfSuch| (|pfTypedId| id) (cadr (|pf0TupleParts| a)))
     (|pfTypedType| id))))
```

8.2.19 defun pfPushMacroBody

[pfMLambda p280]
[pfPushMacroBody p245]

— **defun pfPushMacroBody** —

```
(defun |pfPushMacroBody| (args body)
  (if (null args)
      body
      (|pfMLambda| (car args) (|pfPushMacroBody| (cdr args) body))))
```

8.2.20 defun pfSourceStok

[pfLeaf? p249]
[pfParts p251]
[pfSourceStok p245]
[pfFirst p266]

— **defun pfSourceStok** —

```
(defun |pfSourceStok| (x)
  (cond
   ((|pfLeaf?| x) x)
   ((null (|pfParts| x)) '|NoToken|)
   (t (|pfSourceStok| (|pfFirst| x)))))
```

8.2.21 defun pfTransformArg

[pfTuple? p294]
 [pf0TupleParts p295]
 [pfListOf p247]
 [pfTaggedToTyped1 p246]

— defun pfTransformArg —

```
(defun |pfTransformArg| (args)
  (let (arglist result)
    (if (|pfTuple?| args)
      (setq arglist (|pf0TupleParts| args))
      (setq arglist (list args)))
    (|pfListOf|
     (dolist (|i| arglist (nreverse result))
       (push (|pfTaggedToTyped1| |i|) result))))))
```

—————

8.2.22 defun pfTaggedToTyped1

[pfCollect1? p244]
 [pfCollectVariable1 p244]
 [pfDefinition? p263]
 [pfTyped p292]
 [pfNothing p247]
 [pfTaggedToTyped p291]

— defun pfTaggedToTyped1 —

```
(defun |pfTaggedToTyped1| (arg)
  (cond
    ((|pfCollect1?| arg) (|pfCollectVariable1| arg))
    ((|pfDefinition?| arg) (|pfTyped| arg (|pfNothing|)))
    (t (|pfTaggedToTyped| arg))))
```

—————

8.2.23 defun pfSuch

[pfInfApplication p273]
 [pfId p248]

— defun pfSuch —

```
(defun |pfSuch| (x y)
  (|pfInfApplication| (|pfId| '|\|) x y))
```

8.3 Special Nodes

8.3.1 defun Create a Listof node

[pfTree p254]

— defun pfListOf —

```
(defun |pfListOf| (x)
  (|pfTree| 'listOf x))
```

8.3.2 defun pfNothing

[pfTree p254]

— defun pfNothing —

```
(defun |pfNothing| ()
  (|pfTree| 'nothing nil))
```

8.3.3 defun Is this a Nothing node?

[pfAbSynOp? p414]

— defun pfNothing? —

```
(defun |pfNothing?| (form)
  (|pfAbSynOp?| form 'nothing))
```

8.4 Leaves

8.4.1 defun Create a Document node

[pfLeaf p249]

— defun pfDocument —

```
(defun |pfDocument| (strings)
  (|pfLeaf| ' |Document| strings))
```

—————

8.4.2 defun Construct an Id node

[pfLeaf p249]

— defun pfId —

```
(defun |pfId| (expr)
  (|pfLeaf| ' |id| expr))
```

—————

8.4.3 defun Is this an Id node?

[pfAbSynOp? p414]

— defun pfId? —

```
(defun |pfId?| (form)
  (or (|pfAbSynOp?| form ' |id|) (|pfAbSynOp?| form ' |idsy|)))
```

—————

8.4.4 defun Construct an Id leaf node

[pfLeaf p249]

— defun pfIdPos —

```
(defun |pfIdPos| (expr pos)
  (|pfLeaf| ' |id| expr pos))
```

8.4.5 defun Return the Id part

[tokPart p415]

— defun pfIdSymbol —

```
(defun |pfIdSymbol| (form)
  (|tokPart| form))
```

8.4.6 defun Construct a Leaf node

[tokConstruct p413]

[ifcar p??]

[pfNoPosition p416]

— defun pfLeaf —

```
(defun |pfLeaf| (x y &rest z)
  (|tokConstruct| x y (or (ifcar z) (|pfNoPosition|))))
```

8.4.7 defun Is this a leaf node?

[pfAbSynOp p414]

— defun pfLeaf? —

```
(defun |pfLeaf?| (form)
  (member (|pfAbSynOp| form)
    '(|id| |idsy| |symbol| |string| |char| |float| |expression|
      |integer| |Document| |error|)))
```

8.4.8 defun Return the token position of a leaf node

[tokPosn p415]

— defun pfLeafPosition —

```
(defun |pfLeafPosition| (form)
  (|tokPosn| form))
```

—————

8.4.9 defun Return the Leaf Token

[tokPart p415]

— defun pfLeafToken —

```
(defun |pfLeafToken| (form)
  (|tokPart| form))
```

—————

8.4.10 defun Is this a Literal node?

[pfAbSynOp p414]

— defun pfLiteral? 0 —

```
(defun |pfLiteral?| (form)
  (member (|pfAbSynOp| form)
    '(|integer| |symbol| |expression| |one| |zero| |char| |string| |float|)))
```

—————

8.4.11 defun Create a LiteralClass node

[pfAbSynOp p414]

— defun pfLiteralClass —

```
(defun |pfLiteralClass| (form)
  (|pfAbSynOp| form))
```

—————

8.4.12 defun Return the LiteralString

[tokPart p415]

— defun pfLiteralString —

```
(defun |pfLiteralString| (form)
  (|tokPart| form))
```

—————

8.4.13 defun Return the parts of a tree node

— defun pfParts 0 —

```
(defun |pfParts| (form)
  (cdr form))
```

—————

8.4.14 defun Return the argument unchanged

— defun pfPile 0 —

```
(defun |pfPile| (part)
  part)
```

—————

8.4.15 defun pfPushBody

```
[pfLambda p275]
[pfNothing p247]
[pfPushBody p251]
```

— defun pfPushBody —

```
(defun |pfPushBody| (rt args body)
  (cond
    ((null args) body)
```

```
((null (cdr args)) (|pfLambda| (car args) rt body))
(t
  (|pfLambda| (car args) (|pfNothing|)
    (|pfPushBody| rt (cdr args) body))))
```

8.4.16 defun An S-expression which people can read.

[pfSexpr,strip p252]

— defun pfSexpr —

```
(defun |pfSexpr| (pform)
  (|pfSexpr,strip| pform))
```

8.4.17 defun Create a human readable S-expression

[pfId? p248]
 [pfIdSymbol p249]
 [pfLiteral? p250]
 [pfLiteralString p251]
 [pfLeaf? p249]
 [tokPart p415]
 [pfApplication? p257]
 [pfApplicationArg p256]
 [pfTuple? p294]
 [pf0TupleParts p295]
 [pfApplicationOp p256]
 [pfSexpr,strip p252]
 [pfAbSynOp p414]
 [pfParts p251]

— defun pfSexpr,strip —

```
(defun |pfSexpr,strip| (pform)
  (let (args a result)
    (cond
      ((|pfId?| pform)      (|pfIdSymbol| pform))
      ((|pfLiteral?| pform) (|pfLiteralString| pform))
      ((|pfLeaf?| pform)    (|tokPart| pform))
      ((|pfApplication?| pform)
```

```

(setq a (|pfApplicationArg| pform))
(if (|pfTuple?| a)
  (setq args (|pf0TupleParts| a))
  (setq args (list a)))
(dolist (p (cons (|pfApplicationOp| pform) args) (nreverse result))
  (push (|pfSexpr,strip| p) result)))
(t
  (cons (|pfAbSynOp| pform)
    (dolist (p (|pfParts| pform) (nreverse result))
      (push (|pfSexpr,strip| p) result))))))

```

8.4.18 defun Construct a Symbol or Expression node

```

[pfLeaf? p249]
[pfSymbol p253]
[tokPart p415]
[ifcar p??]
[pfExpression p266]
[pfSexpr p252]

```

— defun pfSymb —

```

(defun |pfSymb| (expr &REST optpos)
  (if (|pfLeaf?| expr)
    (|pfSymbol| (|tokPart| expr) (ifcar optpos))
    (|pfExpression| (|pfSexpr| expr) (ifcar optpos))))

```

8.4.19 defun Construct a Symbol leaf node

```

[pfLeaf p249]
[ifcar p??]

```

— defun pfSymbol —

```

(defun |pfSymbol| (expr &rest optpos)
  (|pfLeaf| '|symbol| expr (ifcar optpos)))

```

8.4.20 defun Is this a Symbol node?

[pfAbSynOp? p414]

— defun pfSymbol? —

```
(defun |pfSymbol?| (form)
  (|pfAbSynOp?| form '|symbol|))
```

—————

8.4.21 defun Return the Symbol part

[tokPart p415]

— defun pfSymbolSymbol —

```
(defun |pfSymbolSymbol| (form)
  (|tokPart| form))
```

—————

8.5 Trees**8.5.1 defun Construct a tree node**

— defun pfTree 0 —

```
(defun |pfTree| (x y)
  (cons x y)))
```

—————

8.5.2 defun Construct an Add node

[pfNothing p247]

[pfTree p254]

— defun pfAdd —

```
(defun |pfAdd| (pfbase pfaddin &rest addon)
  (let (lhs)
    (if addon
      (setq lhs addon)
      (setq lhs (|pfNothing|)))
    (|pfTree| '|Add| (list pfbase pfaddin lhs))))
```

8.5.3 defun Construct an And node

[pfTree p254]

— defun pfAnd —

```
(defun |pfAnd| (pleft pright)
  (|pfTree| '|And| (list pleft pright)))
```

8.5.4 defun pfAttribute

[pfTree p254]

— defun pfAttribute —

```
(defun |pfAttribute| (pfexpr)
  (|pfTree| '|Attribute| (list pfexpr)))
```

8.5.5 defun Return an Application node

[pfTree p254]

— defun pfApplication —

```
(defun |pfApplication| (pfop pfarg)
  (|pfTree| '|Application| (list pfop pfarg)))
```

8.5.6 defun Return the Arg part of an Application node

— defun pfApplicationArg 0 —

```
(defun |pfApplicationArg| (pf)
  (caddr pf))
```

—————

8.5.7 defun Return the Op part of an Application node

— defun pfApplicationOp 0 —

```
(defun |pfApplicationOp| (pf)
  (cadr pf))
```

—————

8.5.8 defun Is this an And node?

[pfAbSynOp? p414]

— defun pfAnd? —

```
(defun |pfAnd?| (pf)
  (|pfAbSynOp?| pf ' |And|))
```

—————

8.5.9 defun Return the Left part of an And node

— defun pfAndLeft 0 —

```
(defun |pfAndLeft| (pf)
  (cadr pf))
```

—————

8.5.10 defun Return the Right part of an And node

— defun pfAndRight 0 —

```
(defun |pfAndRight| (pf)
  (caddr pf))
```

—————

8.5.11 defun Flatten a list of lists

— defun pfAppend 0 —

```
(defun |pfAppend| (list)
  (apply #'append list))
```

—————

8.5.12 defun Is this an Application node?

[pfAbSynOp? p414]

— defun pfApplication? —

```
(defun |pfApplication?| (pf)
  (|pfAbSynOp?| pf '|Application|))
```

—————

8.5.13 defun Create an Assign node

[pfTree p254]

— defun pfAssign —

```
(defun |pfAssign| (pflhsitems pfrhs)
  (|pfTree| '|Assign| (list pflhsitems pfrhs)))
```

—————

8.5.14 defun Is this an Assign node?

[pfAbSynOp? p414]

— defun pfAssign? —

```
(defun |pfAssign?| (pf)
  (|pfAbSynOp?| pf '|Assign|))
```

—————

8.5.15 defun Return the parts of an LhsItem of an Assign node

[pfParts p251]

[pfAssignLhsItems p258]

— defun pf0AssignLhsItems 0 —

```
(defun |pf0AssignLhsItems| (pf)
  (|pfParts| (|pfAssignLhsItems| pf)))
```

—————

8.5.16 defun Return the LhsItem of an Assign node

— defun pfAssignLhsItems 0 —

```
(defun |pfAssignLhsItems| (pf)
  (cadr pf))
```

—————

8.5.17 defun Return the RHS of an Assign node

— defun pfAssignRhs 0 —

```
(defun |pfAssignRhs| (pf)
  (caddr pf))
```

—————

8.5.18 defun Construct an application node for a brace

```
[pfApplication p255]
[pfIdPos p248]
[tokPosn p415]
```

— **defun pfBrace** —

```
(defun |pfBrace| (a part)
  (|pfApplication| (|pfIdPos| '{' (|tokPosn| a)) part))
```

—————

8.5.19 defun Construct an Application node for brace-bars

```
[pfApplication p255]
[pfIdPos p248]
[tokPosn p415]
```

— **defun pfBraceBar** —

```
(defun |pfBraceBar| (a part)
  (|pfApplication| (|pfIdPos| '|{\|\\|}|' (|tokPosn| a)) part))
```

—————

8.5.20 defun Construct an Application node for a bracket

```
[pfApplication p255]
[pfIdPos p248]
[tokPosn p415]
```

— **defun pfBracket** —

```
(defun |pfBracket| (a part)
  (|pfApplication| (|pfIdPos| '[' (|tokPosn| a)) part))
```

—————

8.5.21 defun Construct an Application node for bracket-bars

```
[pfApplication p255]
[pfIdPos p248]
```

[tokPosn p415]

— defun pfBracketBar —

```
(defun |pfBracketBar| (a part)
  (|pfApplication| (|pfIdPos| '|\|) (|tokPosn| a) part))
```

—————

8.5.22 defun Create a Break node

[pfTree p254]

— defun pfBreak —

```
(defun |pfBreak| (pffrom)
  (|pfTree| '|Break| (list pffrom)))
```

—————

8.5.23 defun Is this a Break node?

[pfAbSynOp? p414]

— defun pfBreak? —

```
(defun |pfBreak?| (pf)
  (|pfAbSynOp?| pf '|Break|))
```

—————

8.5.24 defun Return the From part of a Break node

— defun pfBreakFrom 0 —

```
(defun |pfBreakFrom| (pf)
  (cadr pf))
```

—————

8.5.25 defun Construct a Coerceto node

[pfTree p254]

— defun pfCoerceto —

```
(defun |pfCoerceto| (pfexpr pftype)
  (|pfTree| '|Coerceto| (list pfexpr pftype)))
```

—————

8.5.26 defun Is this a CoerceTo node?

[pfAbSynOp? p414]

— defun pfCoerceto? —

```
(defun |pfCoerceto?| (pf)
  (|pfAbSynOp?| pf '|Coerceto|))
```

—————

8.5.27 defun Return the Expression part of a CoerceTo node

— defun pfCoercetoExpr 0 —

```
(defun |pfCoercetoExpr| (pf)
  (cadr pf))
```

—————

8.5.28 defun Return the Type part of a CoerceTo node

— defun pfCoercetoType 0 —

```
(defun |pfCoercetoType| (pf)
  (caddr pf))
```

—————

8.5.29 defun Return the Body of a Collect node

— defun pfCollectBody 0 —

```
(defun |pfCollectBody| (pf)
  (cadr pf))
```

8.5.30 defun Return the Iterators of a Collect node

— defun pfCollectIterators 0 —

```
(defun |pfCollectIterators| (pf)
  (caddr pf))
```

8.5.31 defun Create a Collect node

[pfTree p254]

— defun pfCollect —

```
(defun |pfCollect| (pfbody pfiterators)
  (|pfTree| '|Collect| (list pfbody pfiterators)))
```

8.5.32 defun Is this a Collect node?

[pfAbSynOp? p414]

— defun pfCollect? —

```
(defun |pfCollect?| (pf)
  (|pfAbSynOp?| pf '|Collect|))
```

8.5.33 defun pfDefinition

[pfTree p254]

— defun pfDefinition —

```
(defun |pfDefinition| (pflhsitems pfrhs)
  (|pfTree| '|Definition| (list pflhsitems pfrhs)))
```

—————

8.5.34 defun Return the Lhs of a Definition node

— defun pfDefinitionLhsItems 0 —

```
(defun |pfDefinitionLhsItems| (pf)
  (cadr pf))
```

—————

8.5.35 defun Return the Rhs of a Definition node

— defun pfDefinitionRhs 0 —

```
(defun |pfDefinitionRhs| (pf)
  (caddr pf))
```

—————

8.5.36 defun Is this a Definition node?

[pfAbSynOp? p414]

— defun pfDefinition? —

```
(defun |pfDefinition?| (pf)
  (|pfAbSynOp?| pf '|Definition|))
```

—————

8.5.37 defun Return the parts of a Definition node

[pfParts p251]

[pfDefinitionLhsItems p263]

— defun pf0DefinitionLhsItems —

```
(defun |pf0DefinitionLhsItems| (pf)
  (|pfParts| (|pfDefinitionLhsItems| pf)))
```

—

8.5.38 defun Create a Do node

[pfTree p254]

— defun pfDo —

```
(defun |pfDo| (pfbody)
  (|pfTree| ' |Do| (list pfbody)))
```

—

8.5.39 defun Is this a Do node?

[pfAbSynOp? p414]

— defun pfDo? —

```
(defun |pfDo?| (pf)
  (|pfAbSynOp?| pf ' |Do|))
```

—

8.5.40 defun Return the Body of a Do node

— defun pfDoBody 0 —

```
(defun |pfDoBody| (pf)
  (cadr pf))
```

—

8.5.41 defun Construct a Sequence node

[pfTuple p294]
 [pfListOf p247]
 [pfSequence p289]

— defun pfEnSequence —

```
(defun |pfEnSequence| (a)
  (cond
    ((null a) (|pfTuple| (|pfListOf| a)))
    ((null (cdr a)) (car a))
    (t (|pfSequence| (|pfListOf| a)))))
```

—————

8.5.42 defun Construct an Exit node

[pfTree p254]

— defun pfExit —

```
(defun |pfExit| (pfcond pfexpr)
  (|pfTree| '|Exit| (list pfcond pfexpr)))
```

—————

8.5.43 defun Is this an Exit node?

[pfAbSynOp? p414]

— defun pfExit? —

```
(defun |pfExit?| (pf)
  (|pfAbSynOp?| pf '|Exit|))
```

—————

8.5.44 defun Return the Cond part of an Exit

— defun pfExitCond 0 —

```
(defun |pfExitCond| (pf)
  (cadr pf))
```

8.5.45 defun Return the Expression part of an Exit

```
— defun pfExitExpr 0 —

(defun |pfExitExpr| (pf)
  (caddr pf))
```

8.5.46 defun Create an Export node

```
[pfTree p254]

— defun pfExport —

(defun |pfExport| (pfitems)
  (|pfTree| '|Export| (list pfitems)))
```

8.5.47 defun Construct an Expression leaf node

```
[pfLeaf p249]
[ifcar p??]

— defun pfExpression —

(defun |pfExpression| (expr &rest optpos)
  (|pfLeaf| '|expression| expr (ifcar optpos)))
```

8.5.48 defun pfFirst

```
— defun pfFirst 0 —
```

```
(defun |pfFirst| (form)
  (cadr form))
```

8.5.49 defun Create an Application Fix node

[pfApplication p255]
[pfId p248]

— defun pfFix —

```
(defun |pfFix| (pf)
  (|pfApplication| (|pfId| 'Y) pf))
```

8.5.50 defun Create a Free node

[pfTree p254]

— defun pfFree —

```
(defun |pfFree| (pfitems)
  (|pfTree| '|Free| (list pfitems)))
```

8.5.51 defun Is this a Free node?

[pfAbSynOp? p414]

— defun pfFree? —

```
(defun |pfFree?| (pf)
  (|pfAbSynOp?| pf '|Free|))
```

8.5.52 defun Return the parts of the Items of a Free node

```
[pfParts p251]
[pfFreeItems p268]

— defun pf0FreeItems —

(defun |pf0FreeItems| (pf)
  (|pfParts| (|pfFreeItems| pf)))
```

8.5.53 defun Return the Items of a Free node

```
— defun pfFreeItems 0 —

(defun |pfFreeItems| (pf)
  (cadr pf))
```

8.5.54 defun Construct a Forin node

```
[pfTree p254]

— defun pfForin —

(defun |pfForin| (pflhs pfwhole)
  (|pfTree| '|Forin| (list pflhs pfwhole)))
```

8.5.55 defun Is this a ForIn node?

```
[pfAbSynOp? p414]

— defun pfForin? —

(defun |pfForin?| (pf)
  (|pfAbSynOp?| pf '|Forin|))
```

8.5.56 defun Return all the parts of the LHS of a ForIn node

[pfParts p251]
 [pfForinLhs p269]

— defun pf0ForinLhs —

```
(defun |pf0ForinLhs| (pf)
  (|pfParts| (|pfForinLhs| pf)))
```

—————

8.5.57 defun Return the LHS part of a ForIn node

— defun pfForinLhs 0 —

```
(defun |pfForinLhs| (pf)
  (cadr pf))
```

—————

8.5.58 defun Return the Whole part of a ForIn node

— defun pfForinWhole 0 —

```
(defun |pfForinWhole| (pf)
  (caddr pf))
```

—————

8.5.59 defun pfFromDom

[pfApplication? p257]
 [pfApplication p255]
 [pfApplicationOp p256]
 [pfApplicationArg p256]
 [pfFromdom p270]

— defun pfFromDom —

```
(defun |pfFromDom| (dom expr)
  (cond
    ((|pfApplication?| expr)
     (|pfApplication|
      (|pfFromDom| (|pfApplicationOp| expr) dom)
      (|pfApplicationArg| expr)))
    (t (|pfFromDom| expr dom))))
```

8.5.60 defun Construct a Fromdom node

[pfTree p254]

— defun pfFromdom —

```
(defun |pfFromDom| (pfwhat pfdomain)
  (|pfTree| ' |Fromdom| (list pfwhat pfdomain)))
```

8.5.61 defun Is this a Fromdom mode?

[pfAbSynOp? p414]

— defun pfFromdom? —

```
(defun |pfFromdom?| (pf)
  (|pfAbSynOp?| pf ' |Fromdom|))
```

8.5.62 defun Return the What part of a Fromdom node

— defun pfFromdomWhat 0 —

```
(defun |pfFromdomWhat| (pf)
  (cadr pf))
```

8.5.63 defun Return the Domain part of a Fromdom node

— defun pfFromdomDomain 0 —

```
(defun |pfFromdomDomain| (pf)
  (caddr pf))
```

8.5.64 defun Construct a Hide node

[pfTree p254]

— defun pfHide —

```
(defun |pfHide| (a part)
  (declare (ignore a))
  (|pfTree| '|Hide| (list part)))
```

8.5.65 defun pfIf

[pfTree p254]

— defun pfIf —

```
(defun |pfIf| (pfcond pfthen pfelse)
  (|pfTree| '|If| (list pfcond pfthen pfelse)))
```

8.5.66 defun Is this an If node?

[pfAbSynOp? p414]

— defun pfIf? —

```
(defun |pfIf?| (pf)
  (|pfAbSynOp?| pf '|If|))
```

8.5.67 defun Return the Cond part of an If

— defun pfIfCond 0 —

```
(defun |pfIfCond| (pf)
  (cadr pf))
```

8.5.68 defun Return the Then part of an If

— defun pfIfThen 0 —

```
(defun |pfIfThen| (pf)
  (caddr pf))
```

8.5.69 defun pfIfThenOnly

```
[pfIf p271]
[pfNothing p247]
```

— defun pfIfThenOnly —

```
(defun |pfIfThenOnly| (pred cararg)
  (|pfIf| pred cararg (|pfNothing|)))
```

8.5.70 defun Return the Else part of an If

— defun pfIfElse 0 —

```
(defun |pfIfElse| (pf)
  (caddr pf))
```

8.5.71 defun Construct an Import node

[pfTree p254]

— defun pfImport —

```
(defun |pfImport| (pfitems)
  (|pfTree| '|Import| (list pfitems)))
```

—————

8.5.72 defun Construct an Iterate node

[pfTree p254]

— defun pfIterate —

```
(defun |pfIterate| (pffrom)
  (|pfTree| '|Iterate| (list pffrom)))
```

—————

8.5.73 defun Is this an Iterate node?

[pfAbSynOp? p414]

— defun pfIterate? —

```
(defun |pfIterate?| (pf)
  (|pfAbSynOp?| pf '|Iterate|))
```

—————

8.5.74 defun Handle an infix application

[pfListOf p247]

[pfIdSymbol p249]

[pfAnd p255]

[pfOr p282]

[pfApplication p255]

[pfTuple p294]

— defun pfInfApplication —

```
(defun |pfInfApplication| (op left right)
  (cond
    ((eq (|pfIdSymbol| op) '|and|) (|pfAnd| left right))
    ((eq (|pfIdSymbol| op) '|or|) (|pfOr| left right))
    (t (|pfApplication| op (|pfTuple| (|pfListOf| (list left right)))))))
```

8.5.75 defun Create an Inline node

[pfTree p254]

— defun pfInline —

```
(defun |pfInline| (pfitems)
  (|pfTree| '|Inline| (list pfitems)))
```

8.5.76 defun pfLam

[pfAbSynOp? p414]
 [pfFirst p266]
 [pfNothing p247]
 [pfSecond p288]
 [pfLambda p275]

— defun pfLam —

```
(defun |pfLam| (variable body)
  (let (bdy rets)
    (if (|pfAbSynOp?| body '|returntyped|)
      (setq rets (|pfFirst| body))
      (setq rets (|pfNothing|)))
    (if (|pfAbSynOp?| body '|returntyped|)
      (setq bdy (|pfSecond| body))
      (setq bdy body))
    (|pfLambda| variable rets bdy)))
```

8.5.77 defun pfLambda

[pfTree p254]

— defun pfLambda —

```
(defun |pfLambda| (pfargs pfrets pfbody)
  (|pfTree| ' |Lambda| (list pfargs pfrets pfbody)))
```

—————

8.5.78 defun Return the Body part of a Lambda node

— defun pfLambdaBody 0 —

```
(defun |pfLambdaBody| (pf)
  (caddr pf))
```

—————

8.5.79 defun Return the Rets part of a Lambda node

— defun pfLambdaRets 0 —

```
(defun |pfLambdaRets| (pf)
  (caddr pf))
```

—————

8.5.80 defun Is this a Lambda node?

[pfAbSynOp? p414]

— defun pfLambda? —

```
(defun |pfLambda?| (pf)
  (|pfAbSynOp?| pf ' |Lambda|))
```

—————

8.5.81 defun Return the Args part of a Lambda node

— defun pfLambdaArgs 0 —

```
(defun |pfLambdaArgs| (pf)
  (cadr pf))
```

8.5.82 defun Return the Args of a Lambda Node

[pfParts p251]
[pfLambdaArgs p276]

— defun pf0LambdaArgs —

```
(defun |pf0LambdaArgs| (pf)
  (|pfParts| (|pfLambdaArgs| pf)))
```

8.5.83 defun Construct a Local node

[pfTree p254]

— defun pfLocal —

```
(defun |pfLocal| (pfitems)
  (|pfTree| '|Local| (list pfitems)))
```

8.5.84 defun Is this a Local node?

[pfAbSynOp? p414]

— defun pfLocal? —

```
(defun |pfLocal?| (pf)
  (|pfAbSynOp?| pf '|Local|))
```

8.5.85 defun Return the parts of Items of a Local node

```
[pfParts p251]
[pfLocalItems p277]
```

— defun pf0LocalItems —

```
(defun |pf0LocalItems| (pf)
  (|pfParts| (|pfLocalItems| pf)))
```

—————

8.5.86 defun Return the Items of a Local node

— defun pfLocalItems 0 —

```
(defun |pfLocalItems| (pf)
  (cadr pf))
```

—————

8.5.87 defun Construct a Loop node

```
[pfTree p254]
```

— defun pfLoop —

```
(defun |pfLoop| (pfiterators)
  (|pfTree| '|Loop| (list pfiterators)))
```

—————

8.5.88 defun pfLoop1

```
[pfLoop p277]
[pfListOf p247]
[pfDo p264]
```

— defun pfLoop1 —

```
(defun |pfLoop1| (body)
  (|pfLoop| (|pfListOf| (list (|pfDo| body))))))
```

8.5.89 defun Is this a Loop node?

[pfAbSynOp? p414]

— defun pfLoop? —

```
(defun |pfLoop?| (pf)
  (|pfAbSynOp?| pf '|Loop|))
```

8.5.90 defun Return the Iterators of a Loop node

— defun pfLoopIterators 0 —

```
(defun |pfLoopIterators| (pf)
  (cadr pf))
```

8.5.91 defun pf0LoopIterators

[pfParts p251]

[pf0LoopIterators p278]

— defun pf0LoopIterators —

```
(defun |pf0LoopIterators| (pf)
  (|pfParts| (|pfLoopIterators| pf)))
```

8.5.92 defun pfLp

[pfLoop p277]

[pfListOf p247]

[pfDo p264]

— defun pfLp —

```
(defun |pfLp| (iterators body)
  (|pfLoop| (|pfListOf| (append iterators (list (|pfDo| body))))))
```

8.5.93 defun Create a Macro node

[pfTree p254]

— defun pfMacro —

```
(defun |pfMacro| (pflhs pfrhs)
  (|pfTree| '|Macro| (list pflhs pfrhs)))
```

8.5.94 defun Is this a Macro node?

[pfAbSynOp? p414]

— defun pfMacro? —

```
(defun |pfMacro?| (pf)
  (|pfAbSynOp?| pf '|Macro|))
```

8.5.95 defun Return the Lhs of a Macro node

— defun pfMacroLhs 0 —

```
(defun |pfMacroLhs| (pf)
  (cadr pf))
```

8.5.96 defun Return the Rhs of a Macro node

— defun pfMacroRhs 0 —


```
(defun |pfMacroRhs| (pf)
  (caddr pf))
```

8.5.97 defun Construct an MLambda node

[pfTree p254]

```
— defun pfMLambda —

(defun |pfMLambda| (pfargs pfbody)
  (|pfTree| ' |MLambda| (list pfargs pfbody)))
```

8.5.98 defun Is this an MLambda node?

[pfAbSynOp? p414]

```
— defun pfMLambda? —

(defun |pfMLambda?| (pf)
  (|pfAbSynOp?| pf ' |MLambda|))
```

8.5.99 defun Return the Args of an MLambda

```
— defun pfMLambdaArgs 0 —

(defun |pfMLambdaArgs| (pf)
  (cadr pf))
```

8.5.100 defun Return the parts of an MLambda argument

[pfParts p251]

```
— defun pf0MLambdaArgs —
```

```
(defun |pf0MLambdaArgs| (pf)
  (|pfParts| (|pfMLambdaArgs| pf)))
```

8.5.101 defun pfMLambdaBody

— defun pfMLambdaBody 0 —

```
(defun |pfMLambdaBody| (pf)
  (caddr pf))
```

8.5.102 defun Is this a Not node?

[pfAbSynOp? p414]

— defun pfNot? —

```
(defun |pfNot?| (pf)
  (|pfAbSynOp?| pf ' |Not|))
```

8.5.103 defun Return the Arg part of a Not node

— defun pfNotArg 0 —

```
(defun |pfNotArg| (pf)
  (cadr pf))
```

8.5.104 defun Construct a NoValue node

[pfTree p254]

— defun pfNovalue —

```
(defun |pfNovalue| (pfexpr)
  (|pfTree| ' |Novalue| (list pfexpr)))
```

8.5.105 defun Is this a Novalue node?

[pfAbSynOp? p414]

— defun pfNovalue? —

```
(defun |pfNovalue?| (pf)
  (|pfAbSynOp?| pf ' |Novalue|))
```

8.5.106 defun Return the Expr part of a Novalue node

— defun pfNovalueExpr 0 —

```
(defun |pfNovalueExpr| (pf)
  (cadr pf))
```

8.5.107 defun Construct an Or node

[pfTree p254]

— defun pfOr —

```
(defun |pfOr| (pleft pright)
  (|pfTree| ' |Or| (list pleft pright)))
```

8.5.108 defun Is this an Or node?

[pfAbSynOp? p414]

— defun pfOr? —

```
(defun |pfOr?| (pf)
  (|pfAbSynOp?| pf ' |Or|))
```

8.5.109 defun Return the Left part of an Or node

— defun pfOrLeft 0 —

```
(defun |pfOrLeft| (pf)
  (cadr pf))
```

8.5.110 defun Return the Right part of an Or node

— defun pfOrRight 0 —

```
(defun |pfOrRight| (pf)
  (caddr pf))
```

8.5.111 defun Return the part of a parenthesised expression

— defun pfParen —

```
(defun |pfParen| (a part)
  part)
```

8.5.112 defun pfPretend

[pfTree p254]

— defun pfPretend —

```
(defun |pfPretend| (pfexpr pftype)
  (|pfTree| ' |Pretend| (list pfexpr pftype)))
```

8.5.113 defun Is this a Pretend node?

[pfAbSynOp? p414]

— defun pfPretend? —

```
(defun |pfPretend?| (pf)
  (|pfAbSynOp?| pf ' |Pretend|))
```

8.5.114 defun Return the Expression part of a Pretend node

— defun pfPretendExpr 0 —

```
(defun |pfPretendExpr| (pf)
  (cadr pf))
```

8.5.115 defun Return the Type part of a Pretend node

— defun pfPretendType 0 —

```
(defun |pfPretendType| (pf)
  (caddr pf))
```

8.5.116 defun Construct a QualType node

[pfTree p254]

— defun pfQualType —

```
(defun |pfQualType| (pftype pfqual)
  (|pfTree| ' |QualType| (list pftype pfqual)))
```

8.5.117 defun Construct a Restrict node

[pfTree p254]

— defun pfRestrict —

```
(defun |pfRestrict| (pfexpr pftype)
  (|pfTree| ' |Restrict| (list pfexpr pftype)))
```

8.5.118 defun Is this a Restrict node?

[pfAbSynOp? p414]

— defun pfRestrict? —

```
(defun |pfRestrict?| (pf)
  (|pfAbSynOp?| pf ' |Restrict|))
```

8.5.119 defun Return the Expr part of a Restrict node

— defun pfRestrictExpr 0 —

```
(defun |pfRestrictExpr| (pf)
  (cadr pf))
```

8.5.120 defun Return the Type part of a Restrict node

— defun pfRestrictType 0 —

```
(defun |pfRestrictType| (pf)
  (caddr pf))
```

8.5.121 defun Construct a RetractTo node

[pfTree p254]

— defun pfRetractTo —

```
(defun |pfRetractTo| (pfexpr pftype)
  (|pfTree| ' |RetractTo| (list pfexpr pftype)))
```

8.5.122 defun Construct a Return node

[pfTree p254]

— defun pfReturn —

```
(defun |pfReturn| (pfexpr pffrom)
  (|pfTree| ' |Return| (list pfexpr pffrom)))
```

8.5.123 defun Is this a Return node?

[pfAbSynOp? p414]

— defun pfReturn? —

```
(defun |pfReturn?| (pf)
  (|pfAbSynOp?| pf ' |Return|))
```

8.5.124 defun Return the Expr part of a Return node

— defun pfReturnExpr 0 —

```
(defun |pfReturnExpr| (pf)
  (cadr pf))
```

8.5.125 defun pfReturnNoName

[pfReturn p286]
[pfNothing p247]

— defun pfReturnNoName —

```
(defun |pfReturnNoName| (|value|)
  (|pfReturn| |value| (|pfNothing|)))
```

8.5.126 defun Construct a ReturnTyped node

[pfTree p254]

— defun pfReturnTyped —

```
(defun |pfReturnTyped| (type body)
  (|pfTree| '|returntyped| (list type body)))
```

8.5.127 defun Construct a Rule node

[pfTree p254]

— defun pfRule —

```
(defun |pfRule| (pflhsitems pfrhs)
  (|pfTree| '|Rule| (list pflhsitems pfrhs)))
```

8.5.128 defun Return the Lhs of a Rule node

— defun pfRuleLhsItems 0 —

```
(defun |pfRuleLhsItems| (pf)
  (cadr pf))
```

8.5.129 defun Return the Rhs of a Rule node

— defun pfRuleRhs 0 —

```
(defun |pfRuleRhs| (pf)
  (caddr pf))
```

8.5.130 defun Is this a Rule node?

[pfAbSynOp? p414]

— defun pfRule? —

```
(defun |pfRule?| (pf)
  (|pfAbSynOp?| pf ' |Rule|))
```

8.5.131 defun pfSecond

— defun pfSecond 0 —

```
(defun |pfSecond| (form)
  (caddr form))
```

8.5.132 defun Construct a Sequence node

[pfTree p254]

— defun pfSequence —

```
(defun |pfSequence| (pfargs)
  (|pfTree| ' |Sequence| (list pfargs)))
```

—————

8.5.133 defun Return the Args of a Sequence node

— defun pfSequenceArgs 0 —

```
(defun |pfSequenceArgs| (pf)
  (cadr pf))
```

—————

8.5.134 defun Is this a Sequence node?

[pfAbSynOp? p414]

— defun pfSequence? —

```
(defun |pfSequence?| (pf)
  (|pfAbSynOp?| pf ' |Sequence|))
```

—————

8.5.135 defun Return the parts of the Args of a Sequence node

[pfParts p251]

[pfSequenceArgs p289]

— defun pf0SequenceArgs —

```
(defun |pf0SequenceArgs| (pf)
  (|pfParts| (|pfSequenceArgs| pf)))
```

—————

8.5.136 defun Create a Suchthat node

[pfTree p254]

```

— defun pfSuchthat —

(defun |pfSuchthat| (pfcond)
  (|pfTree| '|Suchthat| (list pfcond)))

```

8.5.137 defun Is this a SuchThat node?

[pfAbSynOp? p414]

```

— defun pfSuchthat? —

(defun |pfSuchthat?| (pf)
  (|pfAbSynOp?| pf '|Suchthat|))

```

8.5.138 defun Return the Cond part of a SuchThat node

```

— defun pfSuchthatCond 0 —

(defun |pfSuchthatCond| (pf)
  (cadr pf))

```

8.5.139 defun Create a Tagged node

[pfTree p254]

```

— defun pfTagged —

(defun |pfTagged| (pftag pfexpr)
  (|pfTree| '|Tagged| (list pftag pfexpr)))

```

8.5.140 defun Is this a Tagged node?

[pfAbSynOp? p414]

— defun pfTagged? —

```
(defun |pfTagged?| (pf)
  (|pfAbSynOp?| pf '|Tagged|))
```

—————

8.5.141 defun Return the Expression portion of a Tagged node

— defun pfTaggedExpr 0 —

```
(defun |pfTaggedExpr| (pf)
  (caddr pf))
```

—————

8.5.142 defun Return the Tag of a Tagged node

— defun pfTaggedTag 0 —

```
(defun |pfTaggedTag| (pf)
  (cadr pf))
```

—————

8.5.143 defun pfTaggedToTyped

```
[pfTagged? p291]
[pfTaggedExpr p291]
[pfNothing p247]
[pfTaggedTag p291]
[pfld? p248]
[pfld p248]
[pfTyped p292]
[pfSuch p246]
[pfInfApplication p273]
```

— defun pfTaggedToTyped —

```
(defun |pfTaggedToTyped| (arg)
  (let (a form rt)
    (if (|pfTagged?| arg)
      (setq rt (|pfTaggedExpr| arg))
      (setq rt (|pfNothing|)))
    (if (|pfTagged?| arg)
      (setq form (|pfTaggedTag| arg))
      (setq form arg))
    (cond
      ((null (|pfId?| form))
       (setq a (|pfId| (gensym)))
       (|pfTyped| (|pfSuch| a (|pfInfApplication| (|pfId| '=) a form)) rt))
      (t (|pfTyped| form rt)))))
```

8.5.144 defun pfTweakIf

[pfIfElse p272]
 [pfNothing? p247]
 [pfListOf p247]
 [pfTree p254]
 [pfIfCond p272]
 [pfIfThen p272]

— defun pfTweakIf —

```
(defun |pfTweakIf| (form)
  (let (b a)
    (setq a (|pfIfElse| form))
    (setq b (if (|pfNothing?| a) (|pfListOf| NIL) a))
    (|pfTree| '|WIf| (list (|pfIfCond| form) (|pfIfThen| form) b))))
```

8.5.145 defun Construct a Typed node

[pfTree p254]

— defun pfTyped —

```
(defun |pfTyped| (pfid pftype)
```

```
(|pfTree| ' |Typed| (list pfid pftype)))
```

8.5.146 defun Is this a Typed node?

[pfAbSynOp? p414]

— defun pfTyped? —

```
(defun |pfTyped?| (pf)
  (|pfAbSynOp?| pf ' |Typed|))
```

8.5.147 defun Return the Type of a Typed node

— defun pfTypedType 0 —

```
(defun |pfTypedType| (pf)
  (caddr pf))
```

8.5.148 defun Return the Id of a Typed node

— defun pfTypedId 0 —

```
(defun |pfTypedId| (pf)
  (cadr pf))
```

8.5.149 defun Construct a Typing node

[pfTree p254]

— defun pfTyping —

```
(defun |pfTyping| (pfitems)
  (|pfTree| '|Typing| (list pfitems)))
```

8.5.150 defun Return a Tuple node

[pfTree p254]

— defun pfTuple —

```
(defun |pfTuple| (pfparts)
  (|pfTree| '|Tuple| (list pfparts)))
```

8.5.151 defun Return a Tuple from a List

[pfTuple p294]

[pfListOf p247]

— defun pfTupleListOf —

```
(defun |pfTupleListOf| (pfparts)
  (|pfTuple| (|pfListOf| pfparts)))
```

8.5.152 defun Is this a Tuple node?

[pfAbSynOp? p414]

— defun pfTuple? —

```
(defun |pfTuple?| (pf)
  (|pfAbSynOp?| pf '|Tuple|))
```

8.5.153 defun Return the Parts of a Tuple node

— defun pfTupleParts 0 —

```
(defun |pfTupleParts| (pf)
  (cadr pf))
```

—————

8.5.154 defun Return the parts of a Tuple

```
[pfParts p251]
[pfTupleParts p295]
```

— defun pf0TupleParts —

```
(defun |pf0TupleParts| (pf)
  (|pfParts| (|pfTupleParts| pf)))
```

—————

8.5.155 defun Return a list from a Sequence node

```
[pfSequence? p289]
[pfAppend p257]
[pf0SequenceArgs p289]
[pfListOf p247]
```

— defun pfUnSequence —

```
(defun |pfUnSequence| (x)
  (if (|pfSequence?| x)
      (|pfListOf| (|pfAppend| (|pf0SequenceArgs| x)))
      (|pfListOf| x)))
```

—————

8.5.156 defun The comment is attached to all signatutres

```
[pfWDeclare p296]
[pfParts p251]
```


— defun pfWDec —

```
(defun |pfWDec| (doc name)
  (mapcar #'(lambda (i) (|pfWDeclare| i doc)) (|pfParts| name)))
```

8.5.157 defun Construct a WDeclare node

[pfTree p254]

— defun pfWDeclare —

```
(defun |pfWDeclare| (pfsignature pfdoc)
  (|pfTree| '|WDeclare| (list pfsignature pfdoc)))
```

8.5.158 defun Construct a Where node

[pfTree p254]

— defun pfWhere —

```
(defun |pfWhere| (pfcontext pfexpr)
  (|pfTree| '|Where| (list pfcontext pfexpr)))
```

8.5.159 defun Is this a Where node?

[pfAbSynOp? p414]

— defun pfWhere? —

```
(defun |pfWhere?| (pf)
  (|pfAbSynOp?| pf '|Where|))
```

8.5.160 defun Return the parts of the Context of a Where node

[pfParts p251]

[pfWhereContext p297]

— defun pf0WhereContext —

```
(defun |pf0WhereContext| (pf)
  (|pfParts| (|pfWhereContext| pf)))
```

—————

8.5.161 defun Return the Context of a Where node

— defun pfWhereContext 0 —

```
(defun |pfWhereContext| (pf)
  (cadr pf))
```

—————

8.5.162 defun Return the Expr part of a Where node

— defun pfWhereExpr 0 —

```
(defun |pfWhereExpr| (pf)
  (caddr pf))
```

—————

8.5.163 defun Construct a While node

[pfTree p254]

— defun pfWhile —

```
(defun |pfWhile| (pfcond)
  (|pfTree| '|While| (list pfcond)))
```

—————

8.5.164 defun Is this a While node?

[pfAbSynOp? p414]

— defun pfWhile? —

```
(defun |pfWhile?| (pf)
  (|pfAbSynOp?| pf '|While|))
```

—————

8.5.165 defun Return the Cond part of a While node

— defun pfWhileCond 0 —

```
(defun |pfWhileCond| (pf)
  (cadr pf))
```

—————

8.5.166 defun Construct a With node

[pfTree p254]

— defun pfWith —

```
(defun |pfWith| (pfbase pfwithin pfwithon)
  (|pfTree| '|With| (list pfbase pfwithin pfwithon)))
```

—————

8.5.167 defun Create a Wrong node

[pfTree p254]

— defun pfWrong —

```
(defun |pfWrong| (pfwhy pfrubble)
  (|pfTree| '|Wrong| (list pfwhy pfrubble)))
```

—————

8.5.168 defun Is this a Wrong node?

[pfAbSynOp? p414]

— defun pfWrong? —

```
(defun |pfWrong?| (pf)
  (|pfAbSynOp?| pf 'Wrong|))
```

—————

Chapter 9

Pftree to s-expression translation

Pftree to s-expression translation. Used to interface the new parser technology to the interpreter. The input is a parseTree and the output is an old-parser-style s-expression.

9.0.169 defun Pftree to s-expression translation

```
[pf2Sex1 p302]  
[$insideSEQ p??]  
[$insideApplication p??]  
[$insideRule p??]  
[$QuietCommand p47]
```

— defun pf2Sex —

```
(defun |pf2Sex| (pf)  
  (let (|$insideSEQ| |$insideApplication| |$insideRule|)  
    (declare (special |$insideSEQ| |$insideApplication| |$insideRule|  
                      |$QuietCommand|))  
    (setq |$QuietCommand| nil)  
    (setq |$insideRule| nil)  
    (setq |$insideApplication| nil)  
    (setq |$insideSEQ| nil)  
    (|pf2Sex1| pf)))
```

—

9.0.170 defun Pftree to s-expression translation inner function

```

[pfNothing? p247]
[pfSymbol? p254]
[pfSymbolSymbol p254]
[pfLiteral? p250]
[pfLiteral2Sex p306]
[pfIdSymbol p249]
[pfApplication? p257]
[pfApplication2Sex p307]
[pfTuple? p294]
[pf2Sex1 p302]
[pf0TupleParts p295]
[pfIf? p271]
[pfIfCond p272]
[pfIfThen p272]
[pfIfElse p272]
[pfTagged? p291]
[pfTaggedTag p291]
[pfTaggedExpr p291]
[pfCoerceto? p261]
[pfCoercetoExpr p261]
[pfCoercetoType p261]
[pfPretend? p284]
[pfPretendExpr p284]
[pfPretendType p284]
[pfFromdom? p270]
[opTran p325]
[pfFromdomWhat p270]
[pfFromdomDomain p271]
[pfSequence? p289]
[pfSequence2Sex p312]
[pfExit? p265]
[pfExitCond p265]
[pfExitExpr p266]
[pfLoop? p278]
[loopIters2Sex p313]
[pf0LoopIterators p278]
[pfCollect? p262]
[pfCollect2Sex p316]
[pfForin? p268]
[pf0ForinLhs p269]
[pfForinWhole p269]
[pfWhile? p298]
[pfWhileCond p298]
[pfSuchthat? p290]

```

[keyedSystemError p??]
 [pfSuchthatCond p290]
 [pfDo? p264]
 [pfDoBody p264]
 [pfTyped? p293]
 [pfTypedType p293]
 [pfTypedId p293]
 [pfAssign? p258]
 [pf0AssignLhsItems p258]
 [pfAssignRhs p258]
 [pfDefinition? p263]
 [pfDefinition2Sex p317]
 [pfLambda? p275]
 [pfLambda2Sex p319]
 [pfMLambda? p280]
 [pfRestrict? p285]
 [pfRestrictExpr p285]
 [pfRestrictType p285]
 [pfFree? p267]
 [pf0FreeItems p268]
 [pfLocal? p276]
 [pf0LocalItems p277]
 [pfWrong? p299]
 [spadThrow p??]
 [pfAnd? p256]
 [pfAndLeft p256]
 [pfAndRight p257]
 [pfOr? p282]
 [pfOrLeft p283]
 [pfOrRight p283]
 [pfNot? p281]
 [pfNotArg p281]
 [pfNovalue? p282]
 [pfNovalueExpr p282]
 [pfRule? p288]
 [pfRule2Sex p320]
 [pfBreak? p260]
 [pfBreakFrom p260]
 [pfMacro? p279]
 [pfReturn? p286]
 [pfReturnExpr p286]
 [pfIterate? p273]
 [pfWhere? p296]
 [pf0WhereContext p297]
 [pfWhereExpr p297]
 [pfAbSynOp p414]


```
[tokPart p415]
[$insideSEQ p??]
[$insideRule p??]
[$QuietCommand p47]
```

— defun pf2Sex1 —

```
(defun |pf2Sex1| (pf)
  (let (args idList type op tagPart tag s)
    (declare (special |$insideSEQ| |$insideRule| |$QuietCommand|))
    (cond
      ((|pfNothing?| pf) '|noBranch|)
      ((|pfSymbol?| pf)
        (if (eq |$insideRule| '|left|)
          (progn
            (setq s (|pfSymbolSymbol| pf))
            (list '|constant| (list 'quote s)))
          (list 'quote (|pfSymbolSymbol| pf))))
      ((|pfLiteral?| pf) (|pfLiteral2Sex| pf))
      ((|pfId?| pf)
        (if |$insideRule|
          (progn
            (setq s (|pfIdSymbol| pf))
            (if (member s '(|%pi| |%e| |%i|))
              s
              (list 'quote s)))
          (|pfIdSymbol| pf)))
      ((|pfApplication?| pf) (|pfApplication2Sex| pf))
      ((|pfTuple?| pf) (cons '|Tuple| (mapcar #'|pf2Sex1| (|pf0TupleParts| pf))))
      ((|pfIf?| pf)
        (list 'if (|pf2Sex1| (|pfIfCond| pf))
              (|pf2Sex1| (|pfIfThen| pf))
              (|pf2Sex1| (|pfIfElse| pf))))
      ((|pfTagged?| pf)
        (setq tag (|pfTaggedTag| pf))
        (setq tagPart
          (if (|pfTuple?| tag)
            (cons '|Tuple| (mapcar #'|pf2Sex1| (|pf0TupleParts| tag)))
            (|pf2Sex1| tag)))
        (list '|:| tagPart (|pf2Sex1| (|pfTaggedExpr| pf))))
      ((|pfCoerceto?| pf)
        (list '|::| (|pf2Sex1| (|pfCoercetoExpr| pf))
              (|pf2Sex1| (|pfCoercetoType| pf))))
      ((|pfPretend?| pf)
        (list '|pretend| (|pf2Sex1| (|pfPretendExpr| pf))
              (|pf2Sex1| (|pfPretendType| pf))))
      ((|pfFromdom?| pf)
        (setq op (|opTran| (|pf2Sex1| (|pfFromdomWhat| pf))))
        (when (eq op '|braceFromCurly|) (setq op 'seq))
```

```

(list '$elt| (|pf2Sex1| (|pfFromdomDomain| pf)) op))
((|pfSequence?| pf) (|pfSequence2Sex| pf))
((|pfExit?| pf)
 (if |$insideSEQ|
  (list 'exit| (|pf2Sex1| (|pfExitCond| pf))
        (|pf2Sex1| (|pfExitExpr| pf))))
  (list 'if (|pf2Sex1| (|pfExitCond| pf))
        (|pf2Sex1| (|pfExitExpr| pf)) 'noBranch|)))
((|pfLoop?| pf) (cons 'repeat (|loopIters2Sex| (|pf0LoopIterators| pf))))
((|pfCollect?| pf) (|pfCollect2Sex| pf))
((|pfForin?| pf)
 (cons 'in
  (append (mapcar #'|pf2Sex1| (|pf0ForinLhs| pf))
    (list (|pf2Sex1| (|pfForinWhole| pf))))))
((|pfWhile?| pf) (list 'while (|pf2Sex1| (|pfWhileCond| pf))))
((|pfSuchthat?| pf)
 (if (eq |$insideRule| '|left|)
  (|keyedSystemError| "S2GE0017" (list "pf2Sex1: pfSuchThat"))
  (list '|\\| (|pf2Sex1| (|pfSuchthatCond| pf)))))
((|pfDo?| pf) (|pf2Sex1| (|pfDoBody| pf)))
((|pfTyped?| pf)
 (setq type (|pfTypedType| pf))
 (if (|pfNothing?| type)
  (|pf2Sex1| (|pfTypedId| pf))
  (list '|:| (|pf2Sex1| (|pfTypedId| pf)) (|pf2Sex1| (|pfTypedType| pf)))))
((|pfAssign?| pf)
 (setq idList (mapcar #'|pf2Sex1| (|pf0AssignLhsItems| pf)))
 (if (not (eql (length idList) 1))
  (setq idList (cons '|Tuple| idList))
  (setq idList (car idList)))
 (list 'let idList (|pf2Sex1| (|pfAssignRhs| pf))))
((|pfDefinition?| pf) (|pfDefinition2Sex| pf))
((|pfLambda?| pf) (|pfLambda2Sex| pf))
((|pfMLambda?| pf) '|/throwAway|)
((|pfRestrict?| pf)
 (list '@ (|pf2Sex1| (|pfRestrictExpr| pf))
        (|pf2Sex1| (|pfRestrictType| pf))))
((|pfFree?| pf) (cons '|free| (mapcar #'|pf2Sex1| (|pf0FreeItems| pf))))
((|pfLocal?| pf) (cons '|local| (mapcar #'|pf2Sex1| (|pf0LocalItems| pf))))
((|pfWrong?| pf) (|spadThrow|))
((|pfAnd?| pf)
 (list '|and| (|pf2Sex1| (|pfAndLeft| pf))
        (|pf2Sex1| (|pfAndRight| pf))))
((|pfOr?| pf)
 (list '|or| (|pf2Sex1| (|pfOrLeft| pf))
        (|pf2Sex1| (|pfOrRight| pf))))
((|pfNot?| pf) (list '|not| (|pf2Sex1| (|pfNotArg| pf))))
((|pfNovalue?| pf)
 (setq |$QuietCommand| t)
 (list 'seq (|pf2Sex1| (|pfNovalueExpr| pf))))

```

```

((|pfRule?| pf) (|pfRule2Sex| pf))
((|pfBreak?| pf) (list '|break| (|pfBreakFrom| pf)))
((|pfMacro?| pf) '|/throwAway|)
((|pfReturn?| pf) (list '|return| (|pf2Sex1| (|pfReturnExpr| pf))))
((|pfIterate?| pf) (list '|iterate|))
((|pfWhere?| pf)
 (setq args (mapcar #'|pf2Sex1| (|pf0WhereContext| pf)))
 (if (eql (length args) 1)
     (cons '|where| (cons (|pf2Sex1| (|pfWhereExpr| pf)) args))
     (list '|where| (|pf2Sex1| (|pfWhereExpr| pf)) (cons 'seq args))))
; -- under strange circumstances/piling, system commands can wind
; -- up in expressions. This just passes it through as a string for
; -- the user to figure out what happened.
((eq (|pfAbSynOp| pf) '|command|) (|tokPart| pf))
(t (|keyedSystemError| "S2GE0017" (list "pf2Sex1"))))

```

9.0.171 defun Convert a Literal to an S-expression

```

[|pfLiteralClass| p250]
[|pfLiteralString| p251]
[|float2Sex| p307]
[|pfSymbolSymbol| p254]
[|pfLeafToken| p250]
[|keyedSystemError| p??]
[|$insideRule| p??]

```

— defun `pfLiteral2Sex` —

```

(defun |pfLiteral2Sex| (pf)
  (let (s type)
    (declare (special |$insideRule|))
    (setq type (|pfLiteralClass| pf))
    (cond
      ((eq type '|integer|) (read-from-string (|pfLiteralString| pf)))
      ((or (eq type '|string|) (eq type '|char|))
       (|pfLiteralString| pf))
      ((eq type '|float|) (|float2Sex| (|pfLiteralString| pf)))
      ((eq type '|symbol|)
       (if |$insideRule|
          (progn
            (setq s (|pfSymbolSymbol| pf))
            (list 'quote s))
          (|pfSymbolSymbol| pf)))
      ((eq type '|expression|) (list 'quote (|pfLeafToken| pf)))
    )
    (t

```

```
(|keyedSystemError| 'S2GE0017 (list "pfLiteral2Sex: unexpected form"))))))
```

9.0.172 defun Convert a float to an S-expression

[*\$useBFasDefault* p??]

— **defun float2Sex** —

```
(defun |float2Sex| (num)
  (let (exp frac bfForm fracPartString intPart dotIndex expPart mantPart eIndex)
    (declare (special |$useBFasDefault|))
    (setq eIndex (search "e" num))
    (if eIndex
      (setq mantPart (subseq num 0 eIndex))
      (setq mantPart num))
    (if eIndex
      (setq expPart (read-from-string (subseq num (+ eIndex 1))))
      (setq expPart 0))
    (setq dotIndex (search "." mantPart))
    (if dotIndex
      (setq intPart (read-from-string (subseq mantPart 0 dotIndex)))
      (setq intPart (read-from-string mantPart)))
    (if dotIndex
      (setq fracPartString (subseq mantPart (+ dotIndex 1)))
      (setq fracPartString 0))
    (setq bfForm
      (make-float intPart (read-from-string fracPartString)
                   (length fracPartString) expPart))
    (if |$useBFasDefault|
      (progn
        (setq frac (cadr bfForm))
        (setq exp (caddr bfForm))
        (list (list '|$elt| (list '|Float|) '|float|) frac exp 10))
      bfForm)))
```

9.0.173 defun Change an Application node to an S-expression

[*pfOp2Sex* p310]
 [*pfApplicationOp* p256]
 [*opTran* p325]
 [*pf0TupleParts* p295]

```
[pfApplicationArg p256]
[pfTuple? p294]
[pf2Sex1 p302]
[pf2Sex p301]
[pfSuchThat2Sex p309]
[hasOptArgs? p311]
[$insideApplication p??]
[$insideRule p??]
```

— defun pfApplication2Sex —

```
(defun |pfApplication2Sex| (pf)
  (let (|$insideApplication| x val realOp tmp1 qt argSex typeList args op)
    (declare (special |$insideApplication| |$insideRule|))
    (setq |$insideApplication| t)
    (setq op (|pfOp2Sex| (|pfApplicationOp| pf)))
    (setq op (|opTran| op))
    (cond
      ((eq op '->)
        (setq args (|pf0TupleParts| (|pfApplicationArg| pf)))
        (if (|pfTuple?| (car args))
          (setq typeList (mapcar #'|pf2Sex1| (|pf0TupleParts| (car args))))
          (setq typeList (list (|pf2Sex1| (car args)))))
        (setq args (cons (|pf2Sex1| (cadr args)) typeList))
        (cons '|Mapping| args))
      ((and (eq op '|:|) (eq |$insideRule| '|left|))
        (list '|multiple| (|pf2Sex| (|pfApplicationArg| pf))))
      ((and (eq op '?') (eq |$insideRule| '|left|))
        (list '|optional| (|pf2Sex| (|pfApplicationArg| pf))))
      (t
        (setq args (|pfApplicationArg| pf))
        (cond
          ((|pfTuple?| args)
            (if (and (eq op '|\\|') (eq |$insideRule| '|left|))
              (|pfSuchThat2Sex| args)
              (progn
                (setq argSex (cdr (|pf2Sex1| args)))
                (cond
                  ((eq op '>') (list '<' (cadr argSex) (car argSex)))
                  ((eq op '>=') (list '|not| (list '<' (car argSex) (cadr argSex))))
                  ((eq op '<=') (list '|not| (list '<' (cadr argSex) (car argSex))))
                  ((eq op 'and') (list '|and| (car argSex) (cadr argSex)))
                  ((eq op 'or') (list '|or| (car argSex) (cadr argSex)))
                  ((eq op '|Iterate|) (list '|iterate|))
                  ((eq op '|by|) (cons 'by argSex))
                  ((eq op '|braceFromCurly|)
                    (if (and (consp argSex) (eq (car argSex) 'seq))
                      argSex
                      (cons 'seq argSex))))
                (cons 'seq argSex))))
          (t
            (if (and (consp argSex) (eq (car argSex) 'seq))
              argSex
              (cons 'seq argSex))))
        (cons 'seq argSex)))
```

```

((and (consp op)
      (progn
        (setq qt (car op))
        (setq tmp1 (cdr op))
        (and (consp tmp1)
              (eq (cdr tmp1) nil)
              (progn
                (setq realOp (car tmp1))
                t))))
      (eq qt 'quote))
      (cons '|applyQuote| (cons op argSex)))
((setq val (|hasOptArgs?| argSex)) (cons op val))
(t (cons op argSex))))))

((and (consp op)
      (progn
        (setq qt (car op))
        (setq tmp1 (cdr op))
        (and (consp tmp1)
              (eq (cdr tmp1) NIL)
              (progn
                (setq realOp (car tmp1))
                t))))
      (eq qt 'quote))
      (list '|applyQuote| op (|pf2Sex1| args)))
((eq op '|braceFromCurly|)
 (setq x (|pf2Sex1| args))
 (if (and (consp x) (eq (car x) 'seq))
     x
     (list 'seq x)))
((eq op '|by|) (list 'by (|pf2Sex1| args)))
(t (list op (|pf2Sex1| args))))))

```

9.0.174 defun Convert a SuchThat node to an S-expression

```

[pf0TupleParts p295]
[pf2Sex1 p302]
[pf2Sex p301]
[$predicateList p??]

```

— defun pfSuchThat2Sex —

```

(defun |pfSuchThat2Sex| (args)
  (let (rhsSex lhsSex argList name)
    (declare (special |$predicateList|))
    (setq name (gentemp))

```

```

(setq argList (|pf0TupleParts| args))
(setq lhsSex (|pf2Sex1| (car argList)))
(setq rhsSex (|pf2Sex| (cadr argList)))
(setq |$predicateList|
  (cons (cons name (cons lhsSex rhsSex)) |$predicateList|))
name))

```

9.0.175 defun pfOp2Sex

```

[|pf2Sex1| p302]
[|pmDontQuote?| p311]
[|pfSymbol?| p254]
[$quotedOpList p??]
[$insideRule p??]

```

— defun pfOp2Sex —

```

(defun |pfOp2Sex| (pf)
  (let (realOp tmp1 op alreadyQuoted)
    (declare (special |$quotedOpList| |$insideRule|))
    (setq alreadyQuoted (|pfSymbol?| pf))
    (setq op (|pf2Sex1| pf))
    (cond
      ((and (consp op)
            (eq (car op) 'quote)
            (progn
              (setq tmp1 (cdr op))
              (and (consp tmp1)
                    (eq (cdr tmp1) nil)
                    (progn
                     (setq realOp (car tmp1)) t))))
        (cond
          ((eq |$insideRule| '|left|) realOp)
          ((eq |$insideRule| '|right|)
            (cond
              ((|pmDontQuote?| realOp) realOp)
              (t
               (setq |$quotedOpList| (cons op |$quotedOpList|))
               op)))
          ((eq realOp '|\\|) realOp)
          ((eq realOp '|:|) realOp)
          ((eq realOp '|?) realOp)
          (t op)))
      (t op))))

```

9.0.176 defun pmDontQuote?

— defun pmDontQuote? 0 —

```
(defun |pmDontQuote?| (sy)
  (member sy
    '(+ - * ** ^ / |log| |exp| |pi| |sqrt| |ei| |li| |erf| |ci|
      |si| |dilog| |sin| |cos| |tan| |cot| |sec| |csc| |asin|
      |acos| |atan| |acot| |asec| |acsc| |sinh| |cosh| |tanh|
      |coth| |sech| |csch| |asinh| |acosh| |atanh| |acoth|
      |asech| |acsc|)))
```

9.0.177 defun hasOptArgs?

— defun hasOptArgs? 0 —

```
(defun |hasOptArgs?| (argSex)
  (let (rhs lhs opt nonOpt tmp1 tmp2)
    (dolist (arg argSex)
      (cond
        ((and (consp arg)
              (eq (car arg) 'optarg)
              (progn
                (setq tmp1 (cdr arg))
                (and (consp tmp1)
                     (progn
                      (setq lhs (car tmp1))
                      (setq tmp2 (cdr tmp1))
                      (and (consp tmp2)
                          (eq (cdr tmp2) nil)
                          (progn
                           (setq rhs (car tmp2))
                           t)))))))
          (setq opt (cons (list lhs rhs) opt)))
        (t (setq nonOpt (cons arg nonOpt)))))
    (when opt
      (nconc (nreverse nonOpt) (list (cons '|construct| (nreverse opt)))))))
```

9.0.178 defun Convert a Sequence node to an S-expression

[pf2Sex1 p302]

[pf0SequenceArgs p289]

[\$insideSEQ p??]

— defun pfSequence2Sex —

```

(defun |pfSequence2Sex| (pf)
  (let (|$insideSEQ| tmp1 ruleList seq)
    (declare (special |$insideSEQ|))
    (setq |$insideSEQ| t)
    (setq seq (|pfSequence2Sex0| (mapcar #'|pf2Sex1| (|pf0SequenceArgs| pf))))
    (cond
      ((and (consp seq)
            (eq (car seq) 'seq)
            (progn (setq ruleList (cdr seq)) 't)
            (consp ruleList)
            (progn
              (setq tmp1 (car ruleList))
              (and (consp tmp1) (eq (car tmp1) '|rule|))))
        (list '|ruleset| (cons '|construct| ruleList)))
      (t seq))))

```

9.0.179 defun pfSequence2Sex0

TPDHERE: rewrite this using (dolist (item seqList)...)

```

;pfSequence2Sex0 seqList ==
; null seqList => "noBranch"
; seqTranList := []
; while seqList ^= nil repeat
;   item := first seqList
;   item is ["exit", cond, value] =>
;     item := ["IF", cond, value, pfSequence2Sex0 rest seqList]
;     seqTranList := [item, :seqTranList]
;   seqList := nil
;   seqTranList := [item, :seqTranList]
;   seqList := rest seqList
; #seqTranList = 1 => first seqTranList
; ["SEQ", :nreverse seqTranList]

```

[pfSequence2Sex0 p312]

— defun pfSequence2Sex0 —

```

(defun |pfSequence2Sex0| (seqList)
  (let (value tmp2 cond tmp1 item seqTranList)
    (if (null seqList)
      '|noBranch|
      (progn
        ((lambda ()
          (loop
            (if (not seqList)
              (return nil)
              (progn
                (setq item (car seqList))
                (cond
                  ((and (consp item)
                       (eq (car item) '|exit|))
                   (progn
                     (setq tmp1 (cdr item))
                     (and (consp tmp1)
                        (progn
                          (setq cond (car tmp1))
                          (setq tmp2 (cdr tmp1))
                          (and (consp tmp2)
                             (eq (cdr tmp2) nil)
                             (progn
                               (setq value (car tmp2))
                               t))))))
                  (t
                   (setq item
                        (list 'if cond value (|pfSequence2Sex0| (cdr seqList))))
                   (setq seqTranList (cons item seqTranList))
                   (setq seqList nil))
                  (t
                   (progn
                     (setq seqTranList (cons item seqTranList))
                     (setq seqList (cdr seqList))))))))))
            (if (eql (length seqTranList) 1)
              (car seqTranList)
              (cons 'seq (nreverse seqTranList))))))

```

9.0.180 defun Convert a loop node to an S-expression

TPDHERE: rewrite using dsetq

```

;loopIters2Sex iterList ==
; result := nil
; for iter in iterList repeat
;   sex := pf2Sex1 iter
;   sex is ['IN, var, ['SEGMENT, i, ["BY", incr]] =>

```

[pf2Sex1 p302]

```
(defun |loopIters2Sex| (iterList)
(let (j incr i var sex result tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 tmp7 tmp8)
(dolist (iter iterList (nreverse result))
(setq sex (|pf2Ssex| iter))
(cond
((and (consp sex)
(eq (car sex) 'in)
(progn
(setq tmp1 (cdr sex))
(and (consp tmp1)
(progn
(setq var (car tmp1))
(setq tmp2 (cdr tmp1))
(and (consp tmp2)
(eq (cdr tmp2) nil)
(progn
(setq tmp3 (car tmp2))
(and (consp tmp3)
(eq (car tmp3) 'segment)
(progn
(setq tmp4 (cdr tmp3))
(and (consp tmp4)
(progn
(setq i (car tmp4))
(setq tmp5 (cdr tmp4))
(and (consp tmp5)
(eq (cdr tmp5) nil)
(progn
(setq tmp6 (car tmp5))
(and (consp tmp6)
(eq (car tmp6) 'by)
(progn
(setq tmp7 (cdr tmp6))
(and (consp tmp7)
(eq (cdr tmp7) nil)
(progn
(setq incr (car tmp7))
t)))))))))))))))
```



```

(eq (cdr tmp2) nil)
(progn
  (setq tmp3 (car tmp2))
  (and (consp tmp3)
    (eq (car tmp3) 'segment)
    (progn
      (setq tmp4 (cdr tmp3))
      (and (consp tmp4)
        (progn
          (setq i (car tmp4))
          (setq tmp5 (cdr tmp4))
          (and (consp tmp5)
            (eq (cdr tmp5) nil)
            (progn
              (setq j (car tmp5))
              t))))))))))
  (setq result (cons (list 'step var i 1 j) result)))
(t (setq result (cons sex result))))))

```

9.0.181 defun Change a Collect node to an S-expression

```

[loopIters2Sex p313]
[pfParts p251]
[pfCollectIterators p262]
[pf2Sex1 p302]
[pfCollectBody p262]

```

— defun pfCollect2Sex —

```

(defun |pfCollect2Sex| (pf)
  (let (var cond sex tmp1 tmp2 tmp3 tmp4)
    (setq sex
      (cons 'collect
        (append (|loopIters2Sex| (|pfParts| (|pfCollectIterators| pf)))
          (list (|pf2Sex1| (|pfCollectBody| pf))))))
    (cond
      ((and (consp sex)
        (eq (car sex) 'collect)
        (progn
          (setq tmp1 (cdr sex))
          (and (consp tmp1)
            (progn
              (setq tmp2 (car tmp1))
              (and (consp tmp2)
                (eq (car tmp2) '|\|))

```

```

      (progn
        (setq tmp3 (cdr tmp2))
        (and (consp tmp3)
              (eq (cdr tmp3) nil)
              (progn
                (setq cond (car tmp3))
                t))))))
    (progn
      (setq tmp4 (cdr tmp1))
      (and (consp tmp4)
            (eq (cdr tmp4) nil)
            (progn (setq var (car tmp4)) t))))))
  (symbolp var))
(list '|\\| var cond))
(t sex))))

```

9.0.182 defun Convert a Definition node to an S-expression

```

[pf2Sex1 p302]
[pf0DefinitionLhsItems p264]
[pfDefinitionRhs p263]
[systemError p??]
[pfLambdaTran p318]
[$insideApplication p??]

```

— defun pfDefinition2Sex —

```

(defun |pfDefinition2Sex| (pf)
  (let (body argList tmp1 rhs id idList)
    (declare (special |$insideApplication|))
    (if |$insideApplication|
        (list 'optarg
              (|pf2Sex1| (car (|pf0DefinitionLhsItems| pf)))
              (|pf2Sex1| (|pfDefinitionRhs| pf)))
        (progn
          (setq idList (mapcar #'|pf2Sex1| (|pf0DefinitionLhsItems| pf)))
          (if (not (eql (length idList) 1))
              (|systemError|
               "lhs of definition must be a single item in the interpreter")
              (progn
                (setq id (car idList))
                (setq rhs (|pfDefinitionRhs| pf))
                (setq tmp1 (|pfLambdaTran| rhs))
                (setq argList (car tmp1))
                (setq body (cdr tmp1))

```

```

(cons 'def
  (cons
    (if (eq argList 'id)
      id
      (cons id argList))
    body))))))

```

9.0.183 defun Convert a Lambda node to an S-expression

```

[pfLambda? p275]
[pf0LambdaArgs p276]
[pfTyped? p293]
[pfCollectArgTran p319]
[pfTypedId p293]
[pfNothing? p247]
[pfTypedType p293]
[pf2Sex1 p302]
[systemError p??]
[pfLambdaRets p275]
[pfLambdaBody p275]

```

— defun pfLambdaTran —

```

(defun |pfLambdaTran| (pf)
  (let (retType argList argTypeList)
    (cond
      ((|pfLambda?| pf)
       (dolist (arg (|pf0LambdaArgs| pf))
         (if (|pfTyped?| arg)
             (progn
              (setq argList
                (cons (|pfCollectArgTran| (|pfTypedId| arg)) argList))
              (if (|pfNothing?| (|pfTypedType| arg))
                  (setq argTypeList (cons nil argTypeList))
                  (setq argTypeList
                    (cons (|pf2Sex1| (|pfTypedType| arg)) argTypeList))))
              (|systemError| "definition args should be typed"))
             (setq argList (nreverse argList)))
         (unless (|pfNothing?| (|pfLambdaRets| pf))
           (setq retType (|pf2Sex1| (|pfLambdaRets| pf))))
         (setq argTypeList (cons retType (nreverse argTypeList)))
         (cons argList
              (list argTypeList
                    (mapcar #'(lambda (x) (declare (ignore x)) nil) argTypeList)
                    (|pf2Sex1| (|pfLambdaBody| pf))))))
    )
  )

```

```
(t (cons 'id| (list '(nil) '(nil) (|pf2Sex1| pf))))))
```

9.0.184 defun pfCollectArgTran

```
[pfCollect? p262]
[pf2sex1 p??]
[pfParts p251]
[pfCollectIterators p262]
[pfCollectBody p262]
```

— defun pfCollectArgTran —

```
(defun |pfCollectArgTran| (pf)
  (let (cond tmp2 tmp1 id conds)
    (cond
      ((|pfCollect?| pf)
       (setq conds (mapcar #'|pf2sex1| (|pfParts| (|pfCollectIterators| pf))))
       (setq id (|pf2Sex1| (|pfCollectBody| pf)))
       (cond
         ((and (consp conds) ; conds is [ "|", cond ]
              (eq (cdr conds) nil))
          (progn
            (setq tmp1 (car conds))
            (and (consp tmp1)
                 (eq (car tmp1) '|\\|))
            (progn
              (setq tmp2 (cdr tmp1))
              (and (consp tmp2)
                   (eq (cdr tmp2) nil))
              (progn
                (setq cond (car tmp2))
                t))))))
          (list '|\\| id cond))
        (t (cons id conds))))
    (t (|pf2Sex1| pf))))))
```

9.0.185 defun Convert a Lambda node to an S-expression

```
[pfLambdaTran p318]
```

— defun pfLambda2Sex —


```
(defun |pfLambda2Sex| (pf)
  (let (body argList tmp1)
    (setq tmp1 (|pfLambdaTran| pf))
    (setq argList (car tmp1))
    (setq body (cdr tmp1))
    (cons 'adeft (cons argList body))))
```

9.0.186 defun Convert a Rule node to an S-expression

```
[pfLhsRule2Sex p320]
[pfRuleLhsItems p288]
[pfRhsRule2Sex p321]
[pfRuleRhs p288]
[ruleLhsTran p324]
[rulePredicateTran p321]
[$multiVarPredicateList p??]
[$predicateList p??]
[$quotedOpList p??]
```

— defun pfRule2Sex —

```
(defun |pfRule2Sex| (pf)
  (let (|$multiVarPredicateList| |$predicateList| |$quotedOpList| rhs lhs)
    (declare (special |$multiVarPredicateList| |$predicateList| |$quotedOpList|))
    (setq |$quotedOpList| nil)
    (setq |$predicateList| nil)
    (setq |$multiVarPredicateList| nil)
    (setq lhs (|pfLhsRule2Sex| (|pfRuleLhsItems| pf)))
    (setq rhs (|pfRhsRule2Sex| (|pfRuleRhs| pf)))
    (setq lhs (|ruleLhsTran| lhs))
    (|rulePredicateTran|
     (if |$quotedOpList|
        (list '|rule| lhs rhs (cons '|construct| |$quotedOpList|))
        (list '|rule| lhs rhs)))))
```

9.0.187 defun Convert the Lhs of a Rule to an S-expression

```
[pf2Sex1 p302]
[$insideRule p??]
```

— defun pfLhsRule2Sex —

```
(defun |pfLhsRule2Sex| (lhs)
  (let (|$insideRule|)
    (declare (special |$insideRule|))
    (setq |$insideRule| '|left|)
    (|pf2Sex1| lhs)))
```

9.0.188 defun Convert the Rhs of a Rule to an S-expression

```
[pf2Sex1 p302]
|$insideRule p??]
```

— defun pfRhsRule2Sex —

```
(defun |pfRhsRule2Sex| (rhs)
  (let (|$insideRule|)
    (declare (special |$insideRule|))
    (setq |$insideRule| '|right|)
    (|pf2Sex1| rhs)))
```

9.0.189 defun Convert a Rule predicate to an S-expression

```
;rulePredicateTran rule ==
; null $multiVarPredicateList => rule
; varList := patternVarsOf [rhs for [.,.,:rhs] in $multiVarPredicateList]
; predBody :=
;   CDR $multiVarPredicateList =>
;     ['AND, :[:pvarPredTran(rhs, varList) for [.,.,:rhs] in
;       $multiVarPredicateList]]
;   [ [.,.,:rhs],:] := $multiVarPredicateList
;   pvarPredTran(rhs, varList)
; ['suchThat, rule,
;   ['construct, :[ ["QUOTE", var] for var in varList]],
;   ['ADEF, '(predicateVariable),
;   '((Boolean) (List (Expression (Integer))))), '(() ()),
;   predBody]]
```

```
[patternVarsOf p323]
[pvarPredTran p324]
[$multiVarPredicateList p??]
```

— defun rulePredicateTran —

```

(defun |rulePredicateTran| (rule)
  (let (predBody varList rhs tmp1 result)
    (declare (special |$multiVarPredicateList|))
    (if (null |$multiVarPredicateList|)
      rule
      (progn
        (setq varList
          (|patternVarsOf|
            ((lambda (t1 t2 t3)
              (loop
                (cond
                  ((or (atom t2)
                     (progn
                      (setq t3 (car t2))
                      nil))
                 (return (nreverse t1))))
              (t
                (and (consp t3)
                  (progn
                    (setq tmp1 (cdr t3))
                    (and (consp tmp1)
                      (progn
                        (setq rhs (cdr tmp1))
                        t))))
                  (setq t1 (cons rhs t1))))))
            (setq t2 (cdr t2))))
          nil |$multiVarPredicateList| nil)))
    (setq predBody
      (cond
        ((cdr |$multiVarPredicateList|)
         (cons 'and
           ((lambda (t4 t5 t6)
             (loop
               (cond
                 ((or (atom t5)
                    (progn
                     (setq t6 (car t5))
                     nil))
                  (return (nreverse t4))))
             (t
               (and (consp t6)
                 (progn
                   (setq tmp1 (cdr t6))
                   (and (consp tmp1)
                     (progn
                       (setq rhs (cdr tmp1))
                       t))))
                 (setq t4
                   (append (reverse (|pvarPredTran| rhs varList))
                     t4))))))
         ))
      ))

```

```

      (setq t5 (cdr t5)))
    nil |$multiVarPredicateList| nil)))
  (t
   (progn
    (setq rhs (cddar |$multiVarPredicateList|))
    (|pvarPredTran| rhs varList))))
  (dolist (var varList) (push (list 'quote var) result))
  (list '|suchThat| rule
   (cons '|construct| (nreverse result))
   (list 'adeft '(|predicateVariable|
                  '((|Boolean|
                     (|List| (|Expression| (|Integer|))))
                    '(nil nil) predBody))))))

```

9.0.190 defun patternVarsOf

[patternVarsOf1 p323]

— defun patternVarsOf —

```

(defun |patternVarsOf| (expr)
  (|patternVarsOf1| expr nil))

```

9.0.191 defun patternVarsOf1

[patternVarsOf1 p323]

— defun patternVarsOf1 —

```

(defun |patternVarsOf1| (expr varList)
  (let (arg1 op)
    (cond
     ((null expr) varList)
     ((atom expr)
      (cond
       ((null (symbolp expr)) varList)
       ((member expr varList) varList)
       (t (cons expr varList))))
     ((and (consp expr)
            (progn
             (setq op (car expr))

```

```

      (setq argl (cdr expr))
      t))
    (progn
      (dolist (arg argl)
        (setq varList (|patternVarsOf1| arg varList)))
        varList))
    (t varList))))

```

9.0.192 defun pvarPredTran

— defun pvarPredTran —

```

(defun |pvarPredTran| (rhs varList)
  (let ((i 0))
    (dolist (var varList rhs)
      (setq rhs (nsbst (list '|elt| '|predicateVariable| (incf i)) var rhs))))))

```

9.0.193 defun Convert the Lhs of a Rule node to an S-expression

```

[patternVarsOf p323]
[nsbst p??]
[$predicateList p??]
[$multiVarPredicateList p??]

```

— defun ruleLhsTran —

```

(defun |ruleLhsTran| (ruleLhs)
  (let (predicate var vars predRhs predLhs name)
    (declare (special |$predicateList| |$multiVarPredicateList|))
    (dolist (pred |$predicateList|)
      (setq name (car pred))
      (setq predLhs (cadr pred))
      (setq predRhs (cddr pred))
      (setq vars (|patternVarsOf1| predRhs))
      (cond
        ((cdr vars)
         (setq ruleLhs (nsbst predLhs name ruleLhs))
         (setq |$multiVarPredicateList| (cons pred |$multiVarPredicateList|)))
        (t
         (setq var (cadr predLhs))

```

```

(setq predicate
  (list '|suchThat| predLhs (list 'adeq (list var)
    '((|Boolean|) (|Expression| (|Integer|))) '(nil nil) predRhs)))
(setq ruleLhs (nsbst predicate name ruleLhs))))
ruleLhs))

```

9.0.194 defvar \$dotdot

— initvars —

```
(defvar |$dotdot| '|...|)
```

9.0.195 defun Translate ops into internal symbols

[[\\$dotdot p325](#)]

— defun opTran 0 —

```

(defun |opTran| (op)
  (declare (special |$dotdot|))
  (cond
    ((equal op |$dotdot|) 'segment)
    ((eq op '[]) '|construct|)
    ((eq op '{}) '|braceFromCurly|)
    ((eq op 'is) '|is|)
    (t op)))

```

Chapter 10

Keyed Message Handling

Throughout the interpreter there are messages printed using a symbol for a database lookup. This was done to enable translation of these messages languages other than English.

Axiom messages are read from a flat file database and returned as one long string. They are preceded in the database by a key and this is how they are referenced from code. For example, one key is S2IL0001 which means:

S2	Scratchpad II designation
I	from the interpreter
L	originally from LISPLIB B00T
0001	a sequence number

Each message may contain formatting codes and and parameter codes. The formatting codes are:

%b	turn on bright printing
%ceoff	turn off centering
%ceon	turn on centering
%d	turn off bright printing
%f	user defined printing
%i	start indentation of 3 more spaces
%l	start a new line
%m	math-print an expression
%rjoff	turn off right justification (actually ragged left)
%rjon	turn on right justification (actually ragged left)
%s	pretty-print as an S-expression
%u	unindent 3 spaces
%x#	insert # spaces

The parameter codes look like %1, %2b, %3p, %4m, %5bp, %6s where the digit is the parameter number and the letters following indicate additional formatting. You can indicate as many additional formatting qualifiers as you like, to the degree they make sense.

- The “p” code means to call `prefix2String` on the parameter, a standard way of printing abbreviated types.
- The “P” operator maps `prefix2String` over its arguments.
- The “o” operation formats the argument as an operation name.
- The “b” means to print that parameter in a bold (bright) font.
- The “c” means to center that parameter on a new line.
- The “r” means to right justify (ragged left) the argument.
- The “f” means that the parameter is a list `[fn, :args]` and that “fn” is to be called on “args” to get the text.

Look in the file with the name defined in `$defaultMsgDatabaseName` above for examples.

10.0.196 `defvar $cacheMessages`

This is used for debugging

— **initvars** —

```
(defvar |$cacheMessages| t)
```

—————

10.0.197 `defvar $msgAlist`

— **initvars** —

```
(defvar |$msgAlist| nil)
```

—————

10.0.198 `defvar $msgDatabaseName`

— **initvars** —

```
(defvar |$msgDatabaseName| nil)
```

—————

10.0.199 defvar \$testingErrorPrefix

— initvars —

```
(defvar |$testingErrorPrefix| "Daly Bug")
```

10.0.200 defvar \$texFormatting

— initvars —

```
(defvar |$texFormatting| nil)
```

10.0.201 defvar \$*msghash*

— initvars —

```
(defvar *msghash* nil "hash table keyed by msg number")
```

10.0.202 defvar \$msgddbPrims

— initvars —

```
(defvar |$msgddbPrims|
  '(|%b| |%d| |%l| |%i| |%u| %U |%n| |%x| |%ce| |%rj| "%U" "%b" "%d"
    "%l" "%i" "%u" "%U" "%n" "%x" "%ce" "%rj"))
```

10.0.203 defvar \$msgddbPunct

— initvars —

```
(defvar |$msgdbPunct|
  '(|.| |,| ! |:| |;| ? |)| | "." ", " "!" ":" "; " "?" "]" " ") )
```

10.0.204 defvar \$msgdbNoBlanksBeforeGroup

— initvars —

```
(defvar |$msgdbNoBlanksBeforeGroup|
  '(" " | | "%" % ,@|$msgdbPrims| ,@|$msgdbPunct|))
```

10.0.205 defvar \$msgdbNoBlanksAfterGroup

— initvars —

```
(defvar |$msgdbNoBlanksAfterGroup|
  '(" " | | "%" % ,@|$msgdbPrims| [ |(| "[" "("))
```

10.0.206 defun Fetch a message from the message database

If the `*msghash*` hash table is empty we call `cacheKeyedMsg` to fill the table, otherwise we do a key lookup in the hash table. [object2Identifier p??]

```
[cacheKeyedMsg p331]
[$defaultMsgDatabaseName p8]
[*msghash* p329]
```

— defun fetchKeyedMsg —

```
(defun |fetchKeyedMsg| (key ignore)
  (declare (ignore ignore) (special *msghash* |$defaultMsgDatabaseName|))
  (setq key (|object2Identifier| key))
  (unless *msghash*
    (setq *msghash* (make-hash-table))
    (cacheKeyedMsg |$defaultMsgDatabaseName|))
  (gethash key *msghash*))
```

10.0.207 defun Cache messages read from message database

```
[done p??]
[done p??]
[*msghash* p329]
```

— defun cacheKeyedMsg —

```
(defun cacheKeyedMsg (file)
  (let ((line "") (msg "") key)
    (declare (special *msghash*))
    (with-open-file (in file)
      (catch 'done
        (loop
          (setq line (read-line in nil nil))
          (cond
            ((null line)
             (when key (setf (gethash key *msghash*) msg))
             (throw 'done nil))
            ((= (length line) 0))
            ((char= (schar line 0) #\S)
             (when key (setf (gethash key *msghash*) msg))
             (setq key (intern line "BOOT"))
             (setq msg ""))
            ('else
             (setq msg (concatenate 'string msg line))))))))))
```

10.0.208 defun getKeyedMsg

```
[fetchKeyedMsg p330]
```

— defun getKeyedMsg —

```
(defun |getKeyedMsg| (key) (|fetchKeyedMsg| key nil))
```

10.0.209 defun Say a message using a keyed lookup

```
[sayKeyedMsgLocal p332]
[$texFormatting p329]
```

— **defun sayKeyedMsg** —

```
(defun |sayKeyedMsg| (key args)
  (let (|$texFormatting|)
    (declare (special |$texFormatting|))
    (setq |$texFormatting| nil)
    (|sayKeyedMsgLocal| key args)))
```

—————

10.0.210 **defun** Handle msg formatting and print to file

```
[segmentKeyedMsg p332]
[getKeyedMsg p331]
[substituteSegmentedMsg p??]
[flowSegmentedMsg p??]
[sayMSG2File p333]
[sayMSG p333]
[$printMsgsToFile p717]
[$linelength p751]
[$margin p751]
[$displayMsgNumber p722]
```

— **defun sayKeyedMsgLocal** —

```
(defun |sayKeyedMsgLocal| (key args)
  (let (msg msgp)
    (declare (special |$printMsgsToFile| $linelength $margin |$displayMsgNumber|))
    (setq msg (|segmentKeyedMsg| (|getKeyedMsg| key)))
    (setq msg (|substituteSegmentedMsg| msg args))
    (when |$displayMsgNumber| (setq msg '("%b" ,key |:| "%d" . ,msg)))
    (setq msgp (|flowSegmentedMsg| msg $linelength $margin))
    (when |$printMsgsToFile| (|sayMSG2File| msgp))
    (|sayMSG| msgp)))
```

—————

10.0.211 **defun** Break a message into words

```
[string2Words p??]
```

— **defun segmentKeyedMsg** —

```
(defun |segmentKeyedMsg| (msg) (|string2Words| msg))
```

10.0.212 defun Write a msg into spadmsg.listing file

```
[makePathname p998]
[defiostream p938]
[sayBrightly1 p1005]
[shut p938]
```

— defun sayMSG2File —

```
(defun |sayMSG2File| (msg)
  (let (file str)
    (setq file (|makePathname| '|spadmsg| '|listing| 'a))
    (setq str (defiostream '((mode . output) (file . ,file)) 255 0))
    (sayBrightly1 msg str)
    (shut str)))
```

10.0.213 defun sayMSG

```
[saybrightly1 p??]
[$algebraOutputStream p739]
```

— defun sayMSG —

```
(defun |sayMSG| (x)
  (declare (special |$algebraOutputStream|))
  (when x (sayBrightly1 x |$algebraOutputStream|)))
```

Chapter 11

Stream Utilities

The input stream is parsed into a large s-expression by repeated calls to Delay. Delay takes a function f and an argument x and returns a list consisting of ("nonnullstream" f x). Eventually multiple calls are made and a large list structure is created that consists of ("nonnullstream" f x ("nonnullstream" f1 x1 ("nonnullstream" f2 x2...

This delay structure is given to StreamNull which walks along the list looking at the head. If the head is "nonnullstream" then the function is applied to the argument.

So, in effect, the input is "zipped up" into a Delay data structure which is then evaluated by calling StreamNull. This "zippered stream" parser was a research project at IBM and Axiom was the testbed (which explains the strange parsing technique).

11.0.214 defun npNull

[StreamNull p335]

— defun npNull —

```
(defun |npNull| (x) (|StreamNull| x))
```

—————

11.0.215 defun StreamNull

[eqcar p??]

— defun StreamNull 0 —

```
(defun |StreamNull| (x)
```



```
(let (st)
  (cond
    ((or (null x) (eqcar x '|nullstream|)) t)
    (t
     ((lambda nil
        (loop
         (cond
           ((not (eqcar x '|nonnullstream|)) (return nil))
           (t
            (setq st (apply (cadr x) (cddr x)))
            (rplaca x (car st))
            (rplacd x (cdr st))))))
      (eqcar x '|nullstream|))))))
```

Chapter 12

Code Piles

The `insertpile` function converts a line-list to a line-forest where a line is a token-dequeue and has a column which is an integer. An A-forest is an A-tree-list. An A-tree has a root which is an A, and subtrees which is an A-forest.

A forest with more than one tree corresponds to a Scratchpad pile structure $(t_1;t_2;t_3;\dots;t_n)$, and a tree corresponds to a pile item. The `(;` and `)` tokens are inserted into a `l`-forest, otherwise the root of the first tree is concatenated with its forest. column `t` is the number of spaces before the first non-space in line `t`.

12.0.216 `defun insertpile`

[npNull p335]

[pilePlusComment p338]

[pilePlusComments p338]

[pileTree p339]

[pileCforest p342]

— `defun insertpile` —

```
(defun |insertpile| (s)
  (let (stream a t1 h1 t2 h tmp1)
    (cond
      ((|npNull| s) (list nil 0 nil s))
      (t
       (setq tmp1 (list (car s) (cdr s)))
       (setq h (car tmp1))
       (setq t2 (cadr tmp1))
       (cond
         ((|pilePlusComment| h)
          (setq tmp1 (|pilePlusComments| s))
          (setq h1 (car tmp1))
```

```

      (setq t1 (cadr tmp1))
      (setq a (|pileTree| (- 1) t1))
      (cons (list (|pileCforest|
                    (append h1 (cons (elt a 2) nil))))
              (elt a 3)))
    (t
      (setq stream (cadar s))
      (setq a (|pileTree| -1 s))
      (cons (list (list (elt a 2) stream)) (elt a 3))))))

```

12.0.217 defun pilePlusComment

[tokType p415]
 [npNull p335]
 [pilePlusComment p338]
 [pilePlusComments p338]

— defun pilePlusComment —

```

(defun |pilePlusComment| (arg)
  (eq (|tokType| (caar arg)) '|comment|))

```

12.0.218 defun pilePlusComments

— defun pilePlusComments —

```

(defun |pilePlusComments| (s)
  (let (t1 h1 t2 h tmp1)
    (cond
      ((|npNull| s) (list nil s))
      (t
        (setq tmp1 (list (car s) (cdr s)))
        (setq h (car tmp1))
        (setq t2 (cadr tmp1))
        (cond
          ((|pilePlusComment| h)
           (setq tmp1 (|pilePlusComments| t2))
           (setq h1 (car tmp1))
           (setq t1 (cadr tmp1))
           (list (cons h h1) t1))

```

```
(t
  (list nil s))))))
```

12.0.219 defun pileTree

```
[npNull p335]
[pileColumn p339]
[pileForests p339]
```

— defun pileTree —

```
(defun |pileTree| (n s)
  (let (hh t1 h tmp1)
    (cond
      ((|npNull| s) (list nil n nil s))
      (t
       (setq tmp1 (list (car s) (cdr s)))
       (setq h (car tmp1))
       (setq t1 (cadr tmp1))
       (setq hh (|pileColumn| (car h)))
       (cond
         ((< n hh) (|pileForests| (car h) hh t1))
         (t (list nil n nil s)))))))
```

12.0.220 defun pileColumn

```
[tokPosn p415]
```

— defun pileColumn —

```
(defun |pileColumn| (arg)
  (cdr (|tokPosn| (caar arg))))
```

12.0.221 defun pileForests

```
[pileForest p340]
[npNull p335]
```

[pileForests p339]
 [pileCtree p342]

— **defun pileForests** —

```
(defun |pileForests| (h n s)
  (let (t1 h1 tmp1)
    (setq tmp1 (|pileForest| n s))
    (setq h1 (car tmp1))
    (setq t1 (cadr tmp1))
    (cond
      ((|npNull| h1) (list t n h s))
      (t (|pileForests| (|pileCtree| h h1) n t1)))))
```

12.0.222 defun pileForest

[pileTree p339]
 [pileForest1 p340]

— **defun pileForest** —

```
(defun |pileForest| (n s)
  (let (t1 h1 t2 h hh b tmp)
    (setq tmp (|pileTree| n s))
    (setq b (car tmp))
    (setq hh (cadr tmp))
    (setq h (caddr tmp))
    (setq t2 (caddr tmp))
    (cond
      (b
        (setq tmp (|pileForest1| hh t2))
        (setq h1 (car tmp))
        (setq t1 (cadr tmp))
        (list (cons h h1) t1))
      (t
        (list nil s)))))
```

12.0.223 defun pileForest1

[eqpileTree p341]
 [pileForest1 p340]

— defun pileForest1 —

```
(defun |pileForest1| (n s)
  (let (t1 h1 t2 h n1 b tmp)
    (setq tmp (|eqpileTree| n s))
    (setq b (car tmp))
    (setq n1 (cadr tmp))
    (setq h (caddr tmp))
    (setq t2 (cadddr tmp))
    (cond
      (b
       (setq tmp (|pileForest1| n t2))
       (setq h1 (car tmp))
       (setq t1 (cadr tmp))
       (list (cons h h1) t1))
      (t (list nil s)))))
```

12.0.224 defun eqpileTree

[npNull p335]
 [pileColumn p339]
 [pileForests p339]

— defun eqpileTree —

```
(defun |eqpileTree| (n s)
  (let (hh t1 h tmp)
    (cond
      ((|npNull| s) (list nil n nil s))
      (t
       (setq tmp (list (car s) (cdr s)))
       (setq h (car tmp))
       (setq t1 (cadr tmp))
       (setq hh (|pileColumn| (car h)))
       (cond
         ((equal hh n) (|pileForests| (car h) hh t1))
         (t (list nil n nil s)))))))
```

12.0.225 defun pileCtree

[dqAppend p346]
 [pileCforest p342]

— **defun pileCtree** —

```
(defun |pileCtree| (x y)
  (|dqAppend| x (|pileCforest| y)))
```

—————

12.0.226 defun pileCforest

Only enpiles forests with ≥ 2 trees [tokPart p415]
 [enPile p342]
 [separatePiles p343]

— **defun pileCforest** —

```
(defun |pileCforest| (x)
  (let (f)
    (cond
      ((null x) nil)
      ((null (cdr x)) (setq f (car x))
        (cond
          ((eq (|tokPart| (caar f)) 'if) (|enPile| f))
          (t f)))
      (t (|enPile| (|separatePiles| x))))))
```

—————

12.0.227 defun enPile

[dqConcat p345]
 [dqUnit p345]
 [tokConstruct p413]
 [firstTokPosn p343]
 [lastTokPosn p343]

— **defun enPile** —

```
(defun |enPile| (x)
  (|dqConcat|
```

```
(list
  (|dqUnit| (|tokConstruct| ' |key| 'settab (|firstTokPosn| x)))
  x
  (|dqUnit| (|tokConstruct| ' |key| 'backtab (|lastTokPosn| x))))))
```

12.0.228 defun firstTokPosn

[tokPosn p415]

— defun firstTokPosn —

```
(defun |firstTokPosn| (arg) (|tokPosn| (caar arg)))
```

12.0.229 defun lastTokPosn

[tokPosn p415]

— defun lastTokPosn —

```
(defun |lastTokPosn| (arg) (|tokPosn| (cadr arg)))
```

12.0.230 defun separatePiles

[dqUnit p345]
 [tokConstruct p413]
 [lastTokPosn p343]
 [dqConcat p345]
 [separatePiles p343]

— defun separatePiles —

```
(defun |separatePiles| (x)
  (let (semicolon a)
    (cond
      ((null x) nil)
      ((null (cdr x)) (car x))
```



```
(t
  (setq a (car x))
  (setq semicolon
    (|dqUnit| (|tokConstruct| 'key| 'backset (|lastTokPosn| a))))
  (|dqConcat| (list a semicolon (|separatePiles| (cdr x))))))
```

Chapter 13

Dequeue Functions

The dqUnit makes a unit dq i.e. a dq with one item, from the item

13.0.231 defun dqUnit

— defun dqUnit 0 —

```
(defun |dqUnit| (s)
  (let (a)
    (setq a (list s))
    (cons a a)))
```

—————

13.0.232 defun dqConcat

The dqConcat function concatenates a list of dq's, destroying all but the last [dqAppend p346]

[dqConcat p345]

— defun dqConcat —

```
(defun |dqConcat| (ld)
  (cond
    ((null ld) nil)
    ((null (cdr ld)) (car ld))
    (t (|dqAppend| (car ld) (|dqConcat| (cdr ld))))))
```

—————

13.0.233 defun dqAppend

The dqAppend function appends 2 dq's, destroying the first

— **defun dqAppend 0** —

```
(defun |dqAppend| (x y)
  (cond
    ((null x) y)
    ((null y) x)
    (t
     (rplacd (cdr x) (car y))
     (rplacd x (cdr y)) x)))
```

—————

13.0.234 defun dqToList

— **defun dqToList 0** —

```
(defun |dqToList| (s)
  (when s (car s)))
```

—————

Chapter 14

Message Handling

14.1 The Line Object

14.1.1 defun Line object creation

This is called in only one place, the `incLine1` function.

— `defun lnCreate 0` —

```
(defun |lnCreate| (extraBlanks string globalNum &rest optFileStuff)
  (let ((localNum (first optFileStuff))
        (filename (second optFileStuff)))
    (unless localNum (setq localNum 0))
    (list extraBlanks string globalNum localNum filename)))
```

—————

14.1.2 defun Line element 0; Extra blanks

— `defun lnExtraBlanks 0` —

```
(defun |lnExtraBlanks| (lineObject) (elt lineObject 0))
```

—————

14.1.3 defun Line element 1; String

— `defun lnString 0` —

```
(defun |lnString| (lineObject) (elt lineObject 1))
```

14.1.4 defun Line element 2; Globlal number

— defun lnGlobalNum 0 —

```
(defun |lnGlobalNum| (lineObject) (elt lineObject 2))
```

14.1.5 defun Line element 2; Set Global number

— defun lnSetGlobalNum 0 —

```
(defun |lnSetGlobalNum| (lineObject num)
  (setf (elt lineObject 2) num))
```

14.1.6 defun Line elemnt 3; Local number

— defun lnLocalNum 0 —

```
(defun |lnLocalNum| (lineObject) (elt lineObject 3))
```

14.1.7 defun Line element 4; Place of origin

— defun lnPlaceOfOrigin 0 —

```
(defun |lnPlaceOfOrigin| (lineObject) (elt lineObject 4))
```

14.1.8 defun Line element 4: Is it a filename?

[lnFileName? p349]

— defun lnImmediate? 0 —

```
(defun |lnImmediate?| (lineObject) (null (|lnFileName?| lineObject)))
```

—————

14.1.9 defun Line element 4: Is it a filename?

— defun lnFileName? 0 —

```
(defun |lnFileName?| (lineObject)
  (let (filename)
    (when (consp (setq filename (elt lineObject 4))) filename)))
```

—————

14.1.10 defun Line element 4; Get filename

[lnFileName? p349]

[ncBug p370]

— defun lnFileName —

```
(defun |lnFileName| (lineObject)
  (let (fN)
    (if (setq fN (|lnFileName?| lineObject))
        fN
        (|ncBug| "there is no file name in %1" (list lineObject)))))
```

—————

14.2 Messages**14.2.1 defun msgCreate**

```
msgObject tag -- category of msg
           -- attributes as a-list
```

```

'imPr => dont save for list processing
toWhere, screen or file
'norep => only display once in list
pos -- position with possible FROM/TO tag
key -- key for message database
argL -- arguments to be placed in the msg test
prefix -- things like "Error: "
text -- the actual text

```

```

[setMsgForcedAttrList p366]
[putDatabaseStuff p367]
[initImPr p369]
[initToWhere p370]

```

— defun msgCreate —

```

(defun |msgCreate| (tag posWTag key argL optPre &rest optAttr)
  (let (msg)
    (when (consp key) (setq tag '|old|))
    (setq msg (list tag posWTag key argL optPre nil))
    (when (car optAttr) (|setMsgForcedAttrList| msg (car optAttr)))
    (|putDatabaseStuff| msg)
    (|initImPr| msg)
    (|initToWhere| msg)
    msg))

```

—————

14.2.2 defun getMsgPosTagOb

— defun getMsgPosTagOb 0 —

```

(defun |getMsgPosTagOb| (msg) (elt msg 1))

```

—————

14.2.3 defun getMsgKey

— defun getMsgKey 0 —

```

(defun |getMsgKey| (msg) (elt msg 2))

```

—————

14.2.4 defun getMsgArgL

— defun getMsgArgL 0 —

```
(defun |getMsgArgL| (msg) (elt msg 3))
```

—————

14.2.5 defun getMsgPrefix

— defun getMsgPrefix 0 —

```
(defun |getMsgPrefix| (msg) (elt msg 4))
```

—————

14.2.6 defun setMsgPrefix

— defun setMsgPrefix 0 —

```
(defun |setMsgPrefix| (msg val) (setf (elt msg 4) val))
```

—————

14.2.7 defun getMsgText

— defun getMsgText 0 —

```
(defun |getMsgText| (msg) (elt msg 5))
```

—————

14.2.8 defun setMsgText

— defun setMsgText 0 —


```
(defun |setMsgText| (msg val)
  (setf (elt msg 5) val))
```

14.2.9 defun getMsgPrefix?

— defun getMsgPrefix? 0 —

```
(defun |getMsgPrefix?| (msg)
  (let ((pre (|getMsgPrefix| msg)))
    (unless (eq pre '|noPre|) pre)))
```

14.2.10 defun getMsgTag

The valid message tags are: line, old, error, warn, bug, unimple, remark, stat, say, debug
[ncTag p417]

— defun getMsgTag 0 —

```
(defun |getMsgTag| (msg) (|ncTag| msg))
```

14.2.11 defun getMsgTag?

```
[IFCAR p??]  
[getMsgTag p352]
```

— defun getMsgTag? 0 —

```
(defun |getMsgTag?| (|msg|)
  (ifcar (member (|getMsgTag| |msg|)
    (list '|line| '|old| '|error| '|warn| '|bug|
          '|unimple| '|remark| '|stat| '|say| '|debug|))))
```

14.2.12 defun line?

[getMsgTag p352]

— **defun line?** —

```
(defun |line?| (msg) (eq (|getMsgTag| msg) '|line|))
```

—————

14.2.13 defun leader?

[getMsgTag p352]

— **defun leader?** —

```
(defun |leader?| (msg) (eq (|getMsgTag| msg) '|leader|))
```

—————

14.2.14 defun toScreen?

[getMsgToWhere p365]

— **defun toScreen?** —

```
(defun |toScreen?| (msg) (not (eq (|getMsgToWhere| msg) '|fileOnly|)))
```

—————

14.2.15 defun ncSoftError

Messages for the USERS of the compiler. The program being compiled has a minor error.

Give a message and continue processing. [desiredMsg p354]

[processKeyedError p355]

[msgCreate p349]

[\$newcompErrorCount p28]

— **defun ncSoftError** —

```
(defun |ncSoftError| (pos erMsgKey erArgL &rest optAttr)
  (declare (special |$newcompErrorCount|)))
```

```
(setq |$newcompErrorCount| (+ |$newcompErrorCount| 1))
(when (|desiredMsg| erMsgKey)
  (|processKeyedError|
   (|msgCreate| '|error| pos erMsgKey erArgL
    "Error" optAttr))))
```

14.2.16 defun ncHardError

The program being compiled is seriously incorrect. Give message and throw to a recovery point. [desiredMsg p354]

```
[processKeyedError p355]
[msgCreate p349]
[ncError p70]
[$newcompErrorCount p28]
```

— defun ncHardError —

```
(defun |ncHardError| (pos erMsgKey erArgL &rest optAttr)
  (let (erMsg)
    (declare (special |$newcompErrorCount|))
    (setq |$newcompErrorCount| (+ |$newcompErrorCount| 1))
    (if (|desiredMsg| erMsgKey)
      (setq erMsg
        (|processKeyedError|
         (|msgCreate| '|error| pos erMsgKey erArgL "Error" optAttr)))
      (|ncError|))))
```

14.2.17 defun desiredMsg

— defun desiredMsg 0 —

```
(defun |desiredMsg| (erMsgKey &rest optCatFlag)
  (cond
    ((null (null optCatFlag)) (car optCatFlag))
    (t t)))
```

14.2.18 defun processKeyedError

```
[getMsgTag? p352]
[getMsgKey p350]
[getMsgPrefix? p352]
[sayBrightly p??]
[CallerName p??]
[msgImPr? p360]
[msgOutputter p355]
[$ncMsgList p27]
```

— **defun processKeyedError** —

```
(defun |processKeyedError| (msg)
  (prog (pre erMsg)
    (declare (special |$ncMsgList|))
    (cond
      ((eq (|getMsgTag?| msg) '|old|)
        (setq erMsg (|getMsgKey| msg))
        (cond
          ((setq pre (|getMsgPrefix?| msg))
            (setq erMsg (cons '|%b| (cons pre (cons '|%d| erMsg)))))
          (|sayBrightly| (cons "old msg from " (cons (|CallerName| 4) erMsg)))
          ((|msgImPr?| msg) (|msgOutputter| msg))
          (t (setq |$ncMsgList| (cons msg |$ncMsgList|)))))))
```

—————

14.2.19 defun msgOutputter

```
[getStFromMsg p356]
[leader? p353]
[line? p353]
[toScreen? p353]
[flowSegmentedMsg p??]
[sayBrightly p??]
[toFile? p365]
[alreadyOpened? p365]
[$linelength p751]
```

— **defun msgOutputter** —

```
(defun |msgOutputter| (msg)
  (let (alreadyOpened shouldFlow st)
    (declare (special $linelength))
    (setq st (|getStFromMsg| msg))
```

```

(setq shouldFlow (null (or (|leader?| msg) (|line?| msg))))
(when (|toScreen?| msg)
  (when shouldFlow (setq st (|flowSegmentedMsg| st $linelength 0)))
  (|sayBrightly| st))
(when (|toFile?| msg)
  (when shouldFlow (setq st (|flowSegmentedMsg| st (- $linelength 6) 0)))
  (setq alreadyOpened (|alreadyOpened?| msg)))))

```

14.2.20 defun listOutputter

[msgOutputter p355]

— defun listOutputter —

```

(defun |listOutputter| (outputList)
  (dolist (msg outputList)
    (|msgOutputter| msg)))

```

14.2.21 defun getStFromMsg

[getPreStL p357]
 [getMsgPrefix? p352]
 [getMsgTag p352]
 [getMsgText p351]
 [getPosStL p358]
 [getMsgKey? p364]
 [pname p1001]
 [getMsgLitSym p364]
 [tabbing p364]

— defun getStFromMsg —

```

(defun |getStFromMsg| (msg)
  (let (st msgKey posStL preStL)
    (setq preStL (|getPreStL| (|getMsgPrefix?| msg)))
    (cond
      ((eq (|getMsgTag| msg) '|line|)
       (cons ""
        (cons "%x1" (append preStL (cons (|getMsgText| msg) nil))))))
    (t

```

```
(setq posStL (|getPosStL| msg))
(setq st
  (cons posStL
    (cons (|getMsgLitSym| msg)
      (cons ""
        (append preStL
          (cons (|tabbing| msg)
            (|getMsgText| msg))))))))))
```

14.2.22 defvar \$preLength

— initvars —

```
(defvar |$preLength| 11)
```

14.2.23 defun getPreStL

```
[size p1001]
[$preLength p357]
```

— defun getPreStL 0 —

```
(defun |getPreStL| (optPre)
  (let (spses extraPlaces)
    (declare (special |$preLength|))
    (cond
      ((null optPre) (list " "))
      (t
        (setq spses
          (cond
            ((< 0 (setq extraPlaces (- (- |$preLength| (size optPre)) 3)))
              (make-string extraPlaces)
            (t "")))
          (list '|%b| optPre spses ":" '|%d|))))))
```

14.2.24 defun getPosStL

```
[showMsgPos? p359]
[getMsgPos p361]
[msgImPr? p360]
[decideHowMuch p361]
[listDecideHowMuch p363]
[ppos p359]
[remLine p364]
[remFile p359]
[$lastPos p??]
```

— defun getPosStL —

```
(defun |getPosStL| (msg)
  (let (printedOrigin printedLineNum printedFileName fullPrintedPos howMuch
        msgPos)
    (declare (special |$lastPos|))
    (cond
      ((null (|showMsgPos?| msg)) "")
      (t
       (setq msgPos (|getMsgPos| msg))
       (setq howMuch
              (if (|msgImPr?| msg)
                  (|decideHowMuch| msgPos |$lastPos|)
                  (|listDecideHowMuch| msgPos |$lastPos|)))
       (setq |$lastPos| msgPos)
       (setq fullPrintedPos (|ppos| msgPos))
       (setq printedFileName
              (cons "%x2" (cons "[" (append (|remLine| fullPrintedPos) (cons "]" nil))))))
       (setq printedLineNum
              (cons "%x2" (cons "[" (append (|remFile| fullPrintedPos) (cons "]" nil))))))
       (setq printedOrigin
              (cons "%x2" (cons "[" (append fullPrintedPos (cons "]" nil))))))
       (cond
         ((eq howMuch 'org)
          (cons "" (append printedOrigin (cons '|%1| nil))))
         ((eq howMuch 'line)
          (cons "" (append printedLineNum (cons '|%1| nil))))
         ((eq howMuch 'file)
          (cons "" (append printedFileName (cons '|%1| nil))))
         ((eq howMuch 'all)
          (cons ""
               (append printedFileName
                      (cons '|%1|
                           (cons ""
                                (append printedLineNum
                                       (cons '|%1| nil))))))))
         (t ""))))))
```

14.2.25 defun ppos

```
[pfNoPosition? p414]
[pfImmediate? p??]
[pfCharPosn p237]
[pfLinePosn p237]
[porigin p90]
[pfileName p238]
```

— defun ppos —

```
(defun |ppos| (p)
  (let (org lpos cpos)
    (cond
      ((|pfNoPosition?| p) (list "no position"))
      ((|pfImmediate?| p) (list "console"))
      (t
       (setq cpos (|pfCharPosn| p))
       (setq lpos (|pfLinePosn| p))
       (setq org (|porigin| (|pfFileName| p)))
       (list org " " "line" " " lpos))))))
```

14.2.26 defun remFile

```
[IFCDR p??]
[IFCAR p??]
```

— defun remFile —

```
(defun |remFile| (positionList) (ifcdr (ifcdr positionList)))
```

14.2.27 defun showMsgPos?

```
[msgImPr? p360]
[leader? p353]
```



```

[$erMsgToss p??]

— defun showMsgPos? 0 —

(defun |showMsgPos?| (msg)
  (declare (special |$erMsgToss|))
  (or |$erMsgToss| (and (null (|msgImPr?| msg)) (null (|leader?| msg))))))

```

14.2.28 defvar \$imPrGuys

```

— initvars —

(defvar |$imPrGuys| (list '|imPr|))

```

14.2.29 defun msgImPr?

```

[getMsgCatAttr p360]

— defun msgImPr? —

(defun |msgImPr?| (msg)
  (eq (|getMsgCatAttr| msg '|$imPrGuys|) '|imPr|))

```

14.2.30 defun getMsgCatAttr

```

[ifcdr p??]
[qassq p??]
[ncAlist p417]

— defun getMsgCatAttr —

(defun |getMsgCatAttr| (msg cat)
  (ifcdr (qassq cat (|ncAlist| msg))))

```

14.2.31 defun getMsgPos

```
[getMsgFTTag? p361]
[getMsgPosTagOb p350]
```

— defun getMsgPos —

```
(defun |getMsgPos| (msg)
  (if (|getMsgFTTag?| msg)
      (cadr (|getMsgPosTagOb| msg))
      (|getMsgPosTagOb| msg)))
```

—————

14.2.32 defun getMsgFTTag?

```
[ifcar p??]
[getMsgPosTagOb p350]
```

— defun getMsgFTTag? —

```
(defun |getMsgFTTag?| (msg)
  (ifcar (member (ifcar (|getMsgPosTagOb| msg)) (list 'from 'to 'fromto))))
```

—————

14.2.33 defun decideHowMuch

When printing a msg, we wish not to show pos information that was shown for a previous msg with identical pos info. org prints out the word noposition or console [poNopos? p362]

```
[poPosImmediate? p362]
[poFileName p362]
[poLinePosn p363]
```

— defun decideHowMuch —

```
(defun |decideHowMuch| (pos oldPos)
  (cond
    ((or (and (|poNopos?| pos) (|poNopos?| oldPos))
         (and (|poPosImmediate?| pos) (|poPosImmediate?| oldPos)))
      'none)
    ((or (|poNopos?| pos) (|poPosImmediate?| pos)) 'org)
    ((or (|poNopos?| oldPos) (|poPosImmediate?| oldPos)) 'all)
    ((not (equal (|poFileName| oldPos) (|poFileName| pos))) 'all))
```

```
((not (equal (|poLinePosn| oldPos) (|poLinePosn| pos))) 'line)
(t 'none)))
```

14.2.34 defun poNopos?

— defun poNopos? 0 —

```
(defun |poNopos?| (posn)
  (equal posn (list 'lnposition))))
```

14.2.35 defun poPosImmediate?

```
[poNopos? p362]
[lnImmediate? p349]
[poGetLineObject p363]
```

— defun poPosImmediate? —

```
(defun |poPosImmediate?| (txp)
  (unless (|poNopos?| txp) (|lnImmediate?| (|poGetLineObject| txp))))
```

14.2.36 defun poFileName

```
[lnFileName p349]
[poGetLineObject p363]
```

— defun poFileName —

```
(defun |poFileName| (posn)
  (if posn
    (|lnFileName| (|poGetLineObject| posn))
    (caar posn)))
```

14.2.37 defun poGetLineObject

— defun poGetLineObject 0 —

```
(defun |poGetLineObject| (posn)
  (car posn))
```

—————

14.2.38 defun poLinePosn

```
[lnLocalNum p348]
[poGetLineObject p363]
```

— defun poLinePosn —

```
(defun |poLinePosn| (posn)
  (if posn
    (|lnLocalNum| (|poGetLineObject| posn))
    (cdar posn)))
```

—————

14.2.39 defun listDecideHowMuch

```
[poNopos? p362]
[poPosImmediate? p362]
[poGlobalLinePosn p73]
```

— defun listDecideHowMuch —

```
(defun |listDecideHowMuch| (pos oldPos)
  (cond
    ((or (and (|poNopos?| pos) (|poNopos?| oldPos))
         (and (|poPosImmediate?| pos) (|poPosImmediate?| oldPos)))
      'none)
    ((|poNopos?| pos) 'org)
    ((|poNopos?| oldPos) 'none)
    ((< (|poGlobalLinePosn| pos) (|poGlobalLinePosn| oldPos))
      (if (|poPosImmediate?| pos) 'org 'line))
    (t 'none)))
```

—————

14.2.40 defun remLine

— defun remLine 0 —

```
(defun |remLine| (positionList) (list (ifcar positionList)))
```

—————

14.2.41 defun getMsgKey?

[identp p1003]

— defun getMsgKey? 0 —

```
(defun |getMsgKey?| (msg)
  (let ((val (|getMsgKey| msg)))
    (when (identp val) val)))
```

—————

14.2.42 defun getMsgLitSym

[getMsgKey? p364]

— defun getMsgLitSym —

```
(defun |getMsgLitSym| (msg)
  (if (|getMsgKey?| msg) " " "*"))
```

—————

14.2.43 defun tabbing

[getMsgPrefix? p352]

[\$preLength p357]

— defun tabbing —

```
(defun |tabbing| (msg)
  (let (chPos)
    (declare (special |$preLength|)))
```

```
(setq chPos 2)
(when (|getMsgPrefix?| msg) (setq chPos (- (+ chPos |$preLength|) 1)))
(cons '|%t| chPos)))
```

14.2.44 defvar \$toWhereGuys

— initvars —

```
(defvar |$toWhereGuys| (list '|fileOnly| '|screenOnly|))
```

14.2.45 defun getMsgToWhere

```
[getMsgCatAttr p360]
```

— defun getMsgToWhere —

```
(defun |getMsgToWhere| (msg) (|getMsgCatAttr| msg '|$toWhereGuys|))
```

14.2.46 defun toFile?

```
[getMsgToWhere p365]
|$fn p??]
```

— defun toFile? —

```
(defun |toFile?| (msg)
  (declare (special |$fn|))
  (and (consp |$fn|) (not (eq (|getMsgToWhere| msg) '|screenOnly|))))
```

14.2.47 defun alreadyOpened?

```
[msgImPr? p360]
```

— defun alreadyOpened? —

```
(defun |alreadyOpened?| (msg) (null (|msgImPr?| msg)))
```

14.2.48 defun setMsgForcedAttrList

```
[setMsgForcedAttr p366]  
[whichCat p367]
```

— defun setMsgForcedAttrList —

```
(defun |setMsgForcedAttrList| (msg attrlist)  
  (dolist (attr attrlist)  
    (|setMsgForcedAttr| msg (|whichCat| attr) attr)))
```

14.2.49 defun setMsgForcedAttr

```
[setMsgCatlessAttr p367]  
[ncPutQ p418]
```

— defun setMsgForcedAttr —

```
(defun |setMsgForcedAttr| (msg cat attr)  
  (if (eq cat '|catless|)  
      (|setMsgCatlessAttr| msg attr)  
      (|ncPutQ| msg cat attr)))
```

14.2.50 defvar \$attrCats

— initvars —

```
(defvar |$attrCats| (list '|$imPrGuys| '|$toWhereGuys| '|$repGuys|))
```

14.2.51 defun whichCat

[ListMember? p??]
 [\$attrCats p366]

— defun whichCat —

```
(defun |whichCat| (attr)
  (let ((found '|catless|) done)
    (declare (special |$attrCats|))
    (loop for cat in |$attrCats| do
      (when (|ListMember?| attr (eval cat))
        (setq found cat)
        (setq done t))
      until done)
    found))
```

14.2.52 defun setMsgCatlessAttr

TPDHERE: Changed from —catless— to '—catless— [ncPutQ p418]
 [ifcdr p??]
 [qassq p??]
 [ncAlist p417]

— defun setMsgCatlessAttr —

```
(defun |setMsgCatlessAttr| (msg attr)
  (|ncPutQ| msg '|catless|
    (cons attr (ifcdr (qassq |catless| (|ncAlist| msg))))))
```

14.2.53 defun putDatabaseStuff

TPDHERE: The variable al is undefined [getMsgInfoFromKey p368]
 [setMsgUnforcedAttrList p368]
 [setMsgText p351]

— defun putDatabaseStuff —

```
(defun |putDatabaseStuff| (msg)
  (let (attributes text tmp)
```



```

(setq tmp (|getMsgInfoFromKey| msg))
(setq text (car tmp))
(setq attributes (cadr tmp))
(when attributes (|setMsgUnforcedAttrList| msg al))
(|setMsgText| msg text)))

```

14.2.54 defun getMsgInfoFromKey

```

[getMsgKey? p364]
[getErFromDbL p??]
[getMsgKey p350]
[segmentKeyedMsg p332]
[removeAttributes p??]
[substituteSegmentedMsg p??]
[getMsgArgL p351]
[$msgDatabaseName p328]

```

— defun getMsgInfoFromKey —

```

(defun |getMsgInfoFromKey| (msg)
  (let (|$msgDatabaseName| attributes tmp msgText dbl msgKey)
    (declare (special |$msgDatabaseName|))
    (setq |$msgDatabaseName| nil)
    (setq msgText
      (cond
        ((setq msgKey (|getMsgKey?| msg))
         (|fetchKeyedMsg| msgKey nil))
        (t (|getMsgKey| msg))))
    (setq msgText (|segmentKeyedMsg| msgText))
    (setq tmp (|removeAttributes| msgText))
    (setq msgText (car tmp))
    (setq attributes (cadr tmp))
    (setq msgText (|substituteSegmentedMsg| msgText (|getMsgArgL| msg)))
    (list msgText attributes)))

```

14.2.55 defun setMsgUnforcedAttrList

```

[setMsgUnforcedAttr p369]
[whichCat p367]

```

— defun setMsgUnforcedAttrList —

```
(defun |setMsgUnforcedAttrList| (msg attrlist)
  (dolist (attr attrlist)
    (|setMsgUnforcedAttr| msg (|whichCat| attr) attr)))
```

14.2.56 defun setMsgUnforcedAttr

```
[setMsgCatlessAttr p367]
[qassq p??]
[ncAlist p417]
[ncPutQ p418]
```

— defun setMsgUnforcedAttr —

```
(defun |setMsgUnforcedAttr| (msg cat attr)
  (cond
    ((eq cat '|catless|) (|setMsgCatlessAttr| msg attr))
    ((null (qassq cat (|ncAlist| msg))) (|ncPutQ| msg cat attr))))
```

14.2.57 defvar \$imPrTagGuys

— initvars —

```
(defvar |$imPrTagGuys| (list '|unimple| '|bug| '|debug| '|say| '|warn|))
```

14.2.58 defun initImPr

```
[getMsgTag p352]
[setMsgUnforcedAttr p369]
[$imPrTagGuys p369]
[$erMsgToss p??]
```

— defun initImPr —

```
(defun |initImPr| (msg)
  (declare (special |$imPrTagGuys| |$erMsgToss|)))
```

```
(when (or |$erMsgToss| (member (|getMsgTag| msg) |$imPrTagGuys|))
      (|setMsgUnforcedAttr| msg '|$imPrGuys| '|imPr|)))
```

14.2.59 defun initToWhere

```
[getMsgCatAttr p360]
[setMsgUnforcedAttr p369]
```

— defun initToWhere —

```
(defun |initToWhere| (msg)
  (if (member '|trace| (|getMsgCatAttr| msg '|catless|))
      (|setMsgUnforcedAttr| msg '|$toWhereGuys| '|screenOnly|)))
```

14.2.60 defun ncBug

Bug in the compiler: something which shouldn't have happened did. [processKeyedError p355]
 [msgCreate p349]
 [enable-backtrace p??]
 [ncAbort p??]
 [\$nopus p28]
 [\$newcompErrorCount p28]

— defun ncBug —

```
(defun |ncBug| (erMsgKey erArgL &rest optAttr)
  (let (erMsg)
    (declare (special |$nopus| |$newcompErrorCount|))
    (setq |$newcompErrorCount| (+ |$newcompErrorCount| 1))
    (setq erMsg
      (|processKeyedError|
        (|msgCreate| '|bug| |$nopus| erMsgKey erArgL "Bug!" optAttr)))
    (enable-backtrace nil)
    (break)
    (|ncAbort|)))
```

14.2.61 defun processMsgList

```
[erMsgSort p371]
[makeMsgFromLine p373]
[poGlobalLinePosn p73]
[getMsgPos p361]
[queueUpErrors p374]
[listOutputter p356]
[$noRepList p??]
[$outputList p??]
```

— defun processMsgList —

```
(defun |processMsgList| (erMsgList lineList)
  (let (|$noRepList| |$outputList| st globalNumOfLine msgLine)
    (declare (special |$noRepList| |$outputList|))
    (setq |$outputList| nil)
    (setq |$noRepList| nil)
    (setq erMsgList (|erMsgSort| erMsgList))
    (dolist (line lineList)
      (setq msgLine (|makeMsgFromLine| line))
      (setq |$outputList| (cons msgLine |$outputList|))
      (setq globalNumOfLine (|poGlobalLinePosn| (|getMsgPos| msgLine)))
      (setq erMsgList (|queueUpErrors| globalNumOfLine erMsgList)))
    (setq |$outputList| (append erMsgList |$outputList|))
    (setq st "-----SOURCE-TEXT-&-ERRORS-----")
    (|listOutputter| (reverse |$outputList|))))
```

—————

14.2.62 defun erMsgSort

```
[erMsgSep p372]
[listSort p??]
```

— defun erMsgSort —

```
(defun |erMsgSort| (erMsgList)
  (let (msgWOPos msgWPos tmp)
    (setq tmp (|erMsgSep| erMsgList))
    (setq msgWPos (car tmp))
    (setq msgWOPos (cadr tmp))
    (setq msgWPos (|listSort| #'|erMsgCompare| msgWPos))
    (setq msgWOPos (reverse msgWOPos))
    (append msgWPos msgWOPos)))
```

14.2.63 defun erMsgCompare

[compareposns p372]
[getMsgPos p361]

— defun erMsgCompare —

```
(defun |erMsgCompare| (ob1 ob2)
  (|compareposns| (|getMsgPos| ob2) (|getMsgPos| ob1)))
```

14.2.64 defun compareposns

[poGlobalLinePosn p73]
[poCharPosn p379]

— defun compareposns —

```
(defun |compareposns| (a b)
  (let (c d)
    (setq c (|poGlobalLinePosn| a))
    (setq d (|poGlobalLinePosn| b))
    (if (equal c d)
        (not (< (|poCharPosn| a) (|poCharPosn| b)))
        (not (< c d)))))
```

14.2.65 defun erMsgSep

[poNopos? p362]
[getMsgPos p361]

— defun erMsgSep —

```
(defun |erMsgSep| (erMsgList)
  (let (msgWOPos msgWPos)
    (dolist (msg erMsgList)
      (if (|poNopos?| (|getMsgPos| msg))
          (setq msgWOPos (cons msg msgWOPos))
```

```
(setq msgWPos (cons msg msgWPos)))
(list msgWPos msgWOPos))
```

14.2.66 defun makeMsgFromLine

```
[getLinePos p374]
[getLineText p374]
[poGlobalLinePosn p73]
[poLinePosn p363]
[strconc p??]
[rep p373]
[char p??]
[size p1001]
[$preLength p357]
```

— defun makeMsgFromLine —

```
(defun |makeMsgFromLine| (line)
  (let (localNumOfLine stNum globalNumOfLine textOfLine posOfLine)
    (declare (special |$preLength|))
    (setq posOfLine (|getLinePos| line))
    (setq textOfLine (|getLineText| line))
    (setq globalNumOfLine (|poGlobalLinePosn| posOfLine))
    (setq stNum (princ-to-string (|poLinePosn| posOfLine)))
    (setq localNumOfLine
      (strconc (|rep| #\space (- |$preLength| 7 (size stNum))) stNum))
    (list '|line| posOfLine nil nil (strconc "Line" localNumOfLine) textOfLine)))
```

14.2.67 defun rep

TPDHERE: This function should be replaced by fillerspaces

— defun rep 0 —

```
(defun |rep| (c n)
  (if (< 0 n)
    (make-string n :initial-element (character c))
    ""))
```

14.2.68 defun getLinePos

```

— defun getLinePos 0 —

(defun |getLinePos| (line) (car line))

```

14.2.69 defun getLineText

```

— defun getLineText 0 —

(defun |getLineText| (line) (cdr line))

```

14.2.70 defun queueUpErrors

```

;queueUpErrors(globalNumOfLine,msgList)==
;   thisPosMsgs := []
;   notThisLineMsgs := []
;   for msg in msgList _
;     while thisPosIsLess(getMsgPos msg,globalNumOfLine) repeat
;       --these are msgs that refer to positions from earlier compilations
;       if not redundant (msg,notThisPosMsgs) then
;         notThisPosMsgs := [msg,:notThisPosMsgs]
;       msgList := rest msgList
;   for msg in msgList _
;     while thisPosIsEqual(getMsgPos msg,globalNumOfLine) repeat
;       if not redundant (msg,thisPosMsgs) then
;         thisPosMsgs := [msg,:thisPosMsgs]
;       msgList := rest msgList
;   if thisPosMsgs then
;     thisPosMsgs := processChPosesForOneLine thisPosMsgs
;     $outputList := NCONC(thisPosMsgs,$outputList)
;   if notThisPosMsgs then
;     $outputList := NCONC(notThisPosMsgs,$outputList)
;   msgList

```

```

[processChPosesForOneLine p378]
[$outputList p??]

```

```

— defun queueUpErrors —

```

```

(DEFUN |queueUpErrors| (|globalNumOfLine| |msgList|)
  (PROG (|notThisPosMsgs| |notThisLineMsgs| |thisPosMsgs|)
    (DECLARE (SPECIAL |$outputList|))
    (RETURN
      (PROGN
        (SETQ |thisPosMsgs| NIL)
        (SETQ |notThisLineMsgs| NIL)
        ((LAMBDA (|bfVar#7| |msg|)
          (LOOP
            (COND
              ((OR (ATOM |bfVar#7|)
                (PROGN (SETQ |msg| (CAR |bfVar#7|)) NIL)
                (NOT (|thisPosIsLess| (|getMsgPos| |msg|)
                  |globalNumOfLine|))))
              (RETURN NIL))
            'T
            (PROGN
              (COND
                ((NULL (|redundant| |msg| |notThisPosMsgs|))
                  (SETQ |notThisPosMsgs|
                    (CONS |msg| |notThisPosMsgs|))))
                (SETQ |msgList| (CDR |msgList|))))
              (SETQ |bfVar#7| (CDR |bfVar#7|))))
          |msgList| NIL)
        ((LAMBDA (|bfVar#8| |msg|)
          (LOOP
            (COND
              ((OR (ATOM |bfVar#8|)
                (PROGN (SETQ |msg| (CAR |bfVar#8|)) NIL)
                (NOT (|thisPosIsEqual| (|getMsgPos| |msg|)
                  |globalNumOfLine|))))
              (RETURN NIL))
            'T
            (PROGN
              (COND
                ((NULL (|redundant| |msg| |thisPosMsgs|))
                  (SETQ |thisPosMsgs| (CONS |msg| |thisPosMsgs|))))
                (SETQ |msgList| (CDR |msgList|))))
              (SETQ |bfVar#8| (CDR |bfVar#8|))))
          |msgList| NIL)
        (COND
          (|thisPosMsgs|
            (SETQ |thisPosMsgs|
              (|processChPosesForOneLine| |thisPosMsgs|))
            (SETQ |$outputList| (NCONC |thisPosMsgs| |$outputList|))))
          (COND
            (|notThisPosMsgs|
              (SETQ |$outputList|
                (NCONC |notThisPosMsgs| |$outputList|))))
            |msgList|))))

```

14.2.71 defun thisPosIsLess

[poNopos? p362]
[poGlobalLinePosn p73]

— defun thisPosIsLess —

```
(defun |thisPosIsLess| (pos num)
  (unless (|poNopos?| pos) (< (|poGlobalLinePosn| pos) num)))
```

14.2.72 defun thisPosIsEqual

[poNopos? p362]
[poGlobalLinePosn p73]

— defun thisPosIsEqual —

```
(defun |thisPosIsEqual| (pos num)
  (unless (|poNopos?| pos) (equal (|poGlobalLinePosn| pos) num)))
```

14.2.73 defun redundant

```
redundant(msg,thisPosMsgs) ==
  found := NIL
  if msgNoRep? msg then
    for item in $noRepList repeat
      sameMsg?(msg,item) => return (found := true)
    $noRepList := [msg,$noRepList]
  found or MEMBER(msg,thisPosMsgs)
```

[msgNoRep? p377]
[sameMsg? p378]
[\$noRepList p??]

— defun redundant —

```

(defun |redundant| (msg thisPosMsgs)
  (prog (found)
    (declare (special |$noRepList|))
    (return
      (progn
        (cond
          ((|msgNoRep?| msg)
            ((lambda (Var9 item)
              (loop
                (cond
                  ((or (atom Var9) (progn (setq item (car Var9)) nil))
                    (return nil))
                  (t
                     (cond
                       ((|sameMsg?| msg item) (return (setq found t))))))
                (setq Var9 (cdr Var9))))
              |$noRepList| nil)
            (setq |$noRepList| (list msg |$noRepList|))))
          (or found (member msg thisPosMsgs)))))))

```

14.2.74 defvar \$repGuys

— initvars —

```

(defvar |$repGuys| (list '|noRep| '|rep|))

```

14.2.75 defun msgNoRep?

[getMsgCatAttr p360]

— defun msgNoRep? —

```

(defun |msgNoRep?| (msg) (eq (|getMsgCatAttr| msg '|$repGuys|) '|noRep|))

```

14.2.76 defun sameMsg?

```
[getMsgKey p350]
[getMsgArgL p351]
```

— **defun sameMsg?** —

```
(defun |sameMsg?| (msg1 msg2)
  (and (equal (|getMsgKey| msg1) (|getMsgKey| msg2))
        (equal (|getMsgArgL| msg1) (|getMsgArgL| msg2))))
```

14.2.77 defun processChPosesForOneLine

```
[posPointers p380]
[getMsgFTTag? p361]
[putFTText p381]
[poCharPosn p379]
[getMsgPos p361]
[getMsgPrefix p351]
[setMsgPrefix p351]
[strconc p??]
[size p1001]
[makeLeaderMsg p379]
[$preLength p357]
```

— **defun processChPosesForOneLine** —

```
(defun |processChPosesForOneLine| (msgList)
  (let (leaderMsg oldPre posLetter chPosList)
    (declare (special |$preLength|))
    (setq chPosList (|posPointers| msgList))
    (dolist (msg msgList)
      (when (|getMsgFTTag?| msg) (|putFTText| msg chPosList))
      (setq posLetter (cdr (assoc (|poCharPosn| (|getMsgPos| msg)) chPosList)))
      (setq oldPre (|getMsgPrefix| msg))
      (|setMsgPrefix| msg
        (strconc oldPre
          (make-string (- |$preLength| 4 (size oldPre))) posLetter)))
      (setq leaderMsg (|makeLeaderMsg| chPosList))
      (nconc msgList (list leaderMsg))))
```

14.2.78 defun poCharPosn

— defun poCharPosn 0 —

```
(defun |poCharPosn| (posn)
  (cdr posn))
```

—————

14.2.79 defun makeLeaderMsg

```
makeLeaderMsg chPosList ==
  st := MAKE_-FULL_-CVEC ($preLength- 3)
  oldPos := -1
  for [posNum,:posLetter] in reverse chPosList repeat
    st := STRCONC(st, _
      rep(char ".", (posNum - oldPos - 1)),posLetter)
    oldPos := posNum
  ['leader,$nupos,'nokey,NIL,NIL,[st] ]
```

```
[$nupos p28]
[$preLength p357]
```

— defun makeLeaderMsg —

```
(defun |makeLeaderMsg| (chPosList)
  (let (posLetter posNum oldPos st)
    (declare (special |$nupos| |$preLength|))
    (setq st (make-string (- |$preLength| 3)))
    (setq oldPos -1)
    ((lambda (Var15 Var14)
      (loop
        (cond
          ((or (atom Var15) (progn (setq Var14 (car Var15)) nil))
            (return nil))
          (t
            (and (consp Var14)
              (progn
                (setq posNum (car Var14))
                (setq posLetter (cdr Var14))
                t)
              (progn
                (setq st
                  (strconc st (|rep| (|char| '|.|) (- posNum oldPos 1)) posLetter))
                (setq oldPos posNum))))))
      (setq Var15 (cdr Var15))))))
```

```
(reverse chPosList) nil)
(list '|leader| |$npos| '|nokey| nil nil (list st))))
```

14.2.80 defun posPointers

TPDHERE: getMsgFTTag is nonsense [poCharPosn p379]

```
[getMsgPos p361]
[IFCAR p??]
[getMsgPos2 p380]
[insertPos p381]
```

— defun posPointers —

```
(defun |posPointers| (msgList)
  (let (posLetterList pos ftPosList posList increment pointers)
    (setq pointers "ABCDEFGHIJKLMNOPQRS")
    (setq increment 0)
    (dolist (msg msgList)
      (setq pos (|poCharPosn| (|getMsgPos| msg)))
      (unless (equal pos (ifcar posList))
        (setq posList (cons pos posList)))
      ; this should probably read TPDHERE
      ; (when (eq (|getMsgPosTagOb| msg) 'fromto)
      (when (eq |getMsgFTTag| 'fromto)
        (setq ftPosList (cons (|poCharPosn| (|getMsgPos2| msg)) ftPosList))))
      (dolist (toPos ftPosList)
        (setq posList (|insertPos| toPos posList)))
      (dolist (pos posList)
        (setq posLetterList
          (cons (cons pos (elt pointers increment)) posLetterList))
        (setq increment (+ increment 1)))
      posLetterList))
```

14.2.81 defun getMsgPos2

```
[getMsgFTTag? p361]
[getMsgPosTagOb p350]
[ncBug p370]
```

— defun getMsgPos2 —

```
(defun |getMsgPos2| (msg)
  (if (|getMsgFTTag?| msg)
      (caddr (|getMsgPosTag0b| msg))
      (|ncBug| "not a from to" nil)))
```

14.2.82 defun insertPos

This function inserts a position in the proper place of a position list. This is used for the 2nd pos of a fromto [done p??]

— defun insertPos 0 —

```
(defun |insertPos| (newPos posList)
  (let (pos top bot done)
    (setq bot (cons 0 posList))
    (do () (done)
      (setq top (cons (car bot) top))
      (setq bot (cdr bot))
      (setq pos (car bot))
      (setq done
        (cond
          ((< pos newPos) nil)
          ((equal pos newPos) t)
          ((< newPos pos)
           (setq top (cons newPos top))
           t))))
    (cons (cdr (reverse top)) bot)))
```

14.2.83 defun putFTText

```
[getMsgFTTag? p361]
[poCharPosn p379]
[getMsgPos p361]
[setMsgText p351]
[getMsgText p351]
[getMsgPos2 p380]
```

— defun putFTText —

```
(defun |putFTText| (msg chPosList)
  (let (charMarker2 pos2 markingText charMarker pos tag)
```

```

(setq tag (|getMsgFTTag?| msg))
(setq pos (|poCharPosn| (|getMsgPos| msg)))
(setq charMarker (cdr (assoc pos chPosList)))
(cond
  ((eq tag 'from)
   (setq markingText (list "(from " charMarker " and on) "))
   (|setMsgText| msg (append markingText (|getMsgText| msg))))
  ((eq tag 'to)
   (setq markingText (list "(up to " charMarker ") "))
   (|setMsgText| msg (append markingText (|getMsgText| msg))))
  ((eq tag 'fromto)
   (setq pos2 (|poCharPosn| (|getMsgPos2| msg)))
   (setq charMarker2 (cdr (assoc pos2 chPosList)))
   (setq markingText (list "(from " charMarker " up to " charMarker2 ") "))
   (|setMsgText| msg (append markingText (|getMsgText| msg))))))

```

14.2.84 defun From

This is called from parameter list of nc message functions

— **defun From 0** —

```
(defun |From| (pos) (list 'from pos))
```

14.2.85 defun To

This is called from parameter list of nc message functions

— **defun To 0** —

```
(defun |To| (pos) (list 'to pos))
```

14.2.86 defun FromTo

This is called from parameter list of nc message functions

— **defun FromTo 0** —

```
(defun |FromTo| (pos1 pos2) (list 'fromto pos1 pos2))
```

Chapter 15

The Interpreter Syntax

15.1 syntax assignment

— assignment.help —

Immediate, Delayed, and Multiple Assignment

```
=====
Immediate Assignment
=====
```

A variable in Axiom refers to a value. A variable has a name beginning with an uppercase or lowercase alphabetic character, "%", or "!". Successive characters (if any) can be any of the above, digits, or "?". Case is distinguished. The following are all examples of valid, distinct variable names:

a	tooBig?	a1B2c3%!?
A	%j	numberOfPoints
beta6	%J	numberofpoints

The "!=" operator is the immediate assignment operator. Use it to associate a value with a variable. The syntax for immediate assignment for a single variable is:

```
variable := expression
```

The value returned by an immediate assignment is the value of expression.

```
a := 1
1
```

Type: PositiveInteger

The right-hand side of the expression is evaluated, yielding 1. The value is then assigned to a.

```
b := a
1
```

Type: PositiveInteger

The right-hand side of the expression is evaluated, yielding 1. This value is then assigned to b. Thus a and b both have the value 1 after the sequence of assignments.

```
a := 2
2
```

Type: PositiveInteger

What is the value of b if a is assigned the value 2?

```
b
1
```

Type: PositiveInteger

The value of b is left unchanged.

This is what we mean when we say this kind of assignment is immediate. The variable b has no dependency on a after the initial assignment. This is the usual notion of assignment in programming languages such as C, Pascal, and Fortran.

```
=====
Delayed Assignment
=====
```

Axiom provides delayed assignment with "==". This implements a delayed evaluation of the right-hand side and dependency checking. The syntax for delayed assignment is

```
variable == expression
```

The value returned by a delayed assignment is the unique value of Void.

```
a == 1
```

Type: Void

```
b == a
```

Type: Void

Using a and b as above, these are the corresponding delayed assignments.

```

a
  Compiling body of rule a to compute value of type PositiveInteger
  1
    Type: PositiveInteger

```

The right-hand side of each delayed assignment is left unevaluated until the variables on the left-hand sides are evaluated.

```

b
  Compiling body of rule b to compute value of type PositiveInteger
  1
    Type: PositiveInteger

```

This gives the same results as before. But if we change a to 2

```

a == 2
  Compiled code for a has been cleared.
  Compiled code for b has been cleared.
  1 old definition(s) deleted for function or rule a
    Type: Void

```

Then a evaluates to 2, as expected

```

a
  Compiling body of rule a to compute value of type PositiveInteger
  2
    Type: PositiveInteger

```

but the value of b reflects the change to a

```

b
  Compiling body of rule b to compute value of type PositiveInteger
  2
    Type: PositiveInteger

```

Multiple Immediate Assignments

It is possible to set several variables at the same time by using a tuple of variables and a tuple of expressions. A tuple is a collection of things separated by commas, often surrounded by parentheses. The syntax for multiple immediate assignment is

```
( var1, var2, ..., varN ) := ( expr1, expr2, ..., exprN )
```

The value returned by an immediate assignment is the value of exprN.

```
( x, y ) := ( 1, 2 )
2
```

Type: PositiveInteger

This sets `x` to 1 and `y` to 2. Multiple immediate assignments are parallel in the sense that the expressions on the right are all evaluated before any assignments on the left are made. However, the order of evaluation of these expressions is undefined.

```
( x, y ) := ( y, x )
1
```

Type: PositiveInteger

```
x
2
```

Type: PositiveInteger

The variable `x` now has the previous value of `y`.

```
y
1
```

Type: PositiveInteger

The variable `y` now has the previous value of `x`.

There is no syntactic form for multiple delayed assignments.

15.2 syntax blocks

— blocks.help —

```
=====
Blocks
=====
```

A block is a sequence of expressions evaluated in the order that they appear, except as modified by control expressions such as `leave`, `return`, `iterate`, and `if-then-else` constructions. The value of a block is the value of the expression last evaluated in the block.

To leave a block early, use `"=>"`. For example,

```
i < 0 => x
```

The expression before the `"=>"` must evaluate to true or false. The expression following the `"=>"` is the return value of the block.

A block can be constructed in two ways:

1. the expressions can be separated by semicolons and the resulting expression surrounded by parentheses, and
 2. the expressions can be written on succeeding lines with each line indented the same number of spaces (which must be greater than zero).
- A block entered in this form is called a pile

Only the first form is available if you are entering expressions directly to Axiom. Both forms are available in .input files. The syntax for a simple block of expressions entered interactively is

```
( expression1 ; expression2 ; ... ; expressionN )
```

The value returned by a block is the value of an "=>" expression, or expressionN if no "=>" is encountered.

In .input files, blocks can also be written in piles. The examples given here are assumed to come from .input files.

```
a :=
  i := gcd(234,672)
  i := 2*i**5 - i + 1
  1 / i

      1
-----
    23323

Type: Fraction Integer
```

In this example, we assign a rational number to a using a block consisting of three expressions. This block is written as a pile. Each expression in the pile has the same indentation, in this case two spaces to the right of the first line.

```
a := ( i := gcd(234,672); i := 2*i**5 - i + 1; 1 / i )

      1
-----
    23323

Type: Fraction Integer
```

Here is the same block written on one line. This is how you are required to enter it at the input prompt.

```
( a := 1; b := 2; c := 3; [a,b,c] )
[1,2,3]

Type: List PositiveInteger
```

AAxiom gives you two ways of writing a block and the preferred way in an .input file is to use a pile. Roughly speaking, a pile is a block whose constituent expressions are indented the same amount. You begin a pile by starting a new line for the first expression, indenting it to the right of the previous line. You then enter the second expression on a new line, vertically aligning it with the first line. And so on. If you need to enter an inner pile, further indent its lines to the right of the outer pile. Axiom knows where a pile ends. It ends when a subsequent line is indented to the left of the pile or the end of the file.

Also See:

- o)help if
- o)help repeat
- o)help while
- o)help for
- o)help suchthat
- o)help parallel
- o)help lists

1

15.3 system clef

— clef.help —

Entering printable keys generally inserts new text into the buffer (unless in overwrite mode, see below). Other special keys can be used to modify the text in the buffer. In the description of the keys below, `^n` means Control-n, or holding the CONTROL key down while pressing "n". Errors will ring the terminal bell.

- `^A/^E` : Move cursor to beginning/end of the line.
- `^F/^B` : Move cursor forward/backward one character.
- `^D` : Delete the character under the cursor.
- `^H, DEL` : Delete the character to the left of the cursor.
- `^K` : Kill from the cursor to the end of line.
- `^L` : Redraw current line.
- `^O` : Toggle overwrite/insert mode. Initially in insert mode. Text added in overwrite mode (including yanks) overwrite existing text, while insert mode does not overwrite.
- `^P/^N` : Move to previous/next item on history list.

¹ "if" (15.6 p 397) "repeat" (15.10 p 404) "while" (65.1.2 p 995) "for" (15.5 p 393) "suchthat" (15.11 p 408) "parallel" (15.9 p 401) "lists" (?? p ??)

`^R/^S` : Perform incremental reverse/forward search for string on the history list. Typing normal characters adds to the current search string and searches for a match. Typing `^R/^S` marks the start of a new search, and moves on to the next match. Typing `^H` or `DEL` deletes the last character from the search string, and searches from the starting location of the last search. Therefore, repeated `DEL`'s appear to unwind to the match nearest the point at which the last `^R` or `^S` was typed. If `DEL` is repeated until the search string is empty the search location begins from the start of the history list. Typing `ESC` or any other editing character accepts the current match and loads it into the buffer, terminating the search.
`^T` : Toggle the characters under and to the left of the cursor.
`^Y` : Yank previously killed text back at current location. Note that this will overwrite or insert, depending on the current mode.
`^U` : Show help (this text).
`TAB` : Perform command completion based on word to the left of the cursor. Words are deemed to contain only the alphanumeric and the `% ! ? _` characters.
`NL, CR` : returns current buffer to the program.

DOS and ANSI terminal arrow key sequences are recognized, and act like:

`up` : same as `^P`
`down` : same as `^N`
`left` : same as `^B`
`right` : same as `^F`

15.4 syntax collection

— collection.help —

=====
 Collection -- Creating Lists and Streams with Iterators
 =====

All of the loop expressions which do not use the `repeat leave` or `iterate` words can be used to create lists and streams. For example:

This creates a simple list of the integers from 1 to 10:

```
list := [i for i in 1..10]
      [1,2,3,4,5,6,7,8,9,10]
                        Type: List PositiveInteger
```


Create a stream of the integers greater than or equal to 1:

```
stream := [i for i in 1..]
[1,2,3,4,5,6,7,...]
Type: Stream PositiveInteger
```

This is a list of the prime numbers between 1 and 10, inclusive:

```
[i for i in 1..10 | prime? i]
[2,3,5,7]
Type: List PositiveInteger
```

This is a stream of the prime integers greater than or equal to 1:

```
[i for i in 1.. | prime? i]
[2,3,5,7,11,13,17,...]
Type: Stream PositiveInteger
```

This is a list of the integers between 1 and 10, inclusive, whose squares are less than 700:

```
[i for i in 1..10 while i*i < 700]
[1,2,3,4,5,6,7,8,9,10]
Type: List PositiveInteger
```

This is a stream of the integers greater than or equal to 1 whose squares are less than 700:

```
[i for i in 1.. while i*i < 700]
[1,2,3,4,5,6,7,...]
Type: Stream PositiveInteger
```

The general syntax of a collection is

```
[ collectExpression iterator1 iterator2 ... iteratorN ]
```

where each iterator is either a for or a while clause. The loop terminates immediately when the end test of any iterator succeeds or when a return expression is evaluated in collectExpression. The value returned by the collection is either a list or a stream of elements, one for each iteration of the collectExpression.

Be careful when you use while to create a stream. By default Axiom tries to compute and display the first ten elements of a stream. If the while condition is not satisfied quickly, Axiom can spend a long (potentially infinite) time trying to compute the elements. Use

```
)set streams calculate
```

to change the defaults to something else. This also affects the number of terms computed and displayed for power series. For the purposes of these examples we have use this system command to display fewer than ten terms.

15.5 syntax for

— for.help —

```
=====
for loops
=====
```

Axiom provide the `for` and `in` keywords in repeat loops, allowing you to integrate across all elements of a list, or to have a variable take on integral values from a lower bound to an upper bound. We shall refer to these modifying clauses of repeat loops as `for` clauses. These clauses can be present in addition to `while` clauses (See `)help while`). As with all other types of repeat loops, `leave` (see `)help leave`) can be used to prematurely terminate evaluation of the loop.

The syntax for a simple loop using `for` is

```
for iterator repeat loopbody
```

The iterator has several forms. Each form has an end test which is evaluated before `loopbody` is evaluated. A `for` loop terminates immediately when the end test succeeds (evaluates to true) or when a `leave` or `return` expression is evaluated in `loopbody`. The value returned by the loop is the unique value of `Void`.

```
=====
for i in n..m repeat
=====
```

If `for` is followed by a variable name, the `in` keyword and then an integer segment of the form `n..m`, the end test for this loop is the predicate `i > m`. The body of the loop is evaluated `m-n+1` times if this number is greater than 0. If this number is less than or equal to 0, the loop body is not evaluated at all.

The variable `i` has the value `n`, `n+1`, ..., `m` for successive iterations of the loop body. The loop variable is a local variable within the loop body. Its value is not available outside the loop body and its value and

type within the loop body completely mask any outer definition of a variable with the same name.

```
for i in 10..12 repeat output(i**3)
1000
1331
1728
```

Type: Void

The loop prints the values of 10^3 , 11^3 , and 12^3 .

```
a := [1,2,3]
[1,2,3]
```

Type: List PositiveInteger

```
for i in 1..#a repeat output(a.i)
1
2
3
```

Type: Void

Iterate across this list using "." to access the elements of a list and the # operation to count its elements.

This type of iteration is applicable to anything that uses ".". You can also use it with functions that use indices to extract elements.

```
m := matrix [ [1,2],[4,3],[9,0] ]
+-   +-
| 1  2 |
| 4  3 |
| 9  0 |
+-   +-
```

Type: Matrix Integer

Define m to be a matrix.

```
for i in 1..nrows(m) repeat output row(m.i)
[1,2]
[4,3]
[9,0]
```

Type: Void

Display the rows of m.

You can iterate with for-loops.

```
for i in 1..5 repeat
  if odd?(i) then iterate
  output(i)
```

```
2
4
```

Type: Void

Display the even integers in a segment.

```
=====
for i in n..m by s repeat
=====
```

By default, the difference between values taken on by a variable in loops such as

```
for i in n..m repeat ...
```

is 1. It is possible to supply another, possibly negative, step value by using the `by` keyword along with `for` and `in`. Like the upper and lower bounds, the step value following the `by` keyword must be an integer. Note that the loop

```
for i in 1..2 by 0 repeat output(i)
```

will not terminate by itself, as the step value does not change the index from its initial value of 1.

```
for i in 1..5 by 2 repeat output(i)
1
3
5
```

Type: Void

This expression displays the odd integers between two bounds.

```
for i in 5..1 by -2 repeat output(i)
5
3
1
```

Type: Void

Use this to display the numbers in reverse order.

```
=====
for i in n.. repeat
=====
```

If the value after the `".."` is omitted, the loop has no end test. A potentially infinite loop is thus created. The variable is given the successive values `n`, `n+1`, `n+2`, ... and the loop is terminated only if a `leave` or `return` expression is evaluated in the loop body. However, you may also add some other modifying clause on the `repeat`, for example,

a while clause, to stop the loop.

```
for i in 15.. while not prime?(i) repeat output(i)
15
16
```

Type: Void

This loop displays the integers greater than or equal to 15 and less than the first prime number greater than 15.

```
=====
for x in l repeat
=====
```

Another variant of the for loop has the form:

```
for x in list repeat loopbody
```

This form is used when you want to iterate directly over the elements of a list. In this form of the for loop, the variable *x* takes on the value of each successive element in *l*. The end test is most simply stated in English: "are there no more *x* in *l*?"

```
l := [0, -5, 3]
[0, -5, 3]
```

Type: List Integer

```
for x in l repeat output(x)
0
-5
3
```

Type: Void

This displays all of the elements of the list *l*, one per line.

Since the list constructing expression

```
expand [n..m]
```

creates the list

```
[n, n+1, ..., m]
```

you might be tempted to think that the loops

```
for i in n..m repeat output(i)
```

and

```
for x in expand [n..m] repeat output(x)
```

are equivalent. The second form first creates the expanded list (no matter how large it might be) and then does the iteration. The first form potentially runs in much less space, as the index variable `i` is simply incremented once per loop and the list is not actually created. Using the first form is much more efficient.

Of course, sometimes you really want to iterate across a specific list. This displays each of the factors of 2400000:

```
for f in factors(factor(2400000)) repeat output(f)
[factor= 2, exponent= 8]
[factor= 3, exponent= 1]
[factor= 5, exponent= 5]
Type: Void
```

15.6 syntax if

— if.help —

```
=====
If-then-else
=====
```

Like many other programming languages, Axiom uses the three keywords `if`, `then`, and `else` to form conditional expressions. The `else` part of the conditional is optional. The expression between the `if` and `then` keywords is a predicate: an expression that evaluates to or is convertible to either true or false, that is, a Boolean.

The syntax for conditional expressions is

```
if predicate then expression1 else expression2
```

where the "else expression2" part is optional. The value returned from a conditional expression is expression1 if the predicate evaluates to true and expression2 otherwise. If no else clause is given, the value is always the unique value of Void.

An if-then-else expression always returns a value. If the else clause is missing then the entire expression returns the unique value of Void. If both clauses are present, the type of the value returned by if is obtained by resolving the types of the values of the two clauses.

The predicate must evaluate to, or be convertible to, an object of type Boolean: true or false. By default, the equal sign "=" creates an equation.

```
x + 1 = y
x + 1 = y
```

Type: Equation Polynomial Integer

This is an equation, not a boolean condition. In particular, it is an object of type Equation Polynomial Integer.

However, for predicates in if expressions, Axiom places a default target type of Boolean on the predicate and equality testing is performed. Thus you need not qualify the "=" in any way. In other contexts you may need to tell Axiom that you want to test for equality rather than create an equation. In these cases, use "@" and a target type of Boolean.

The compound symbol meaning "not equal" in Axiom is "~=". This can be used directly without a package call or a target specification. The expression "a ~= b" is directly translated to "not(a = b)".

Many other functions have return values of type Boolean. These include <, <=, >, >=, ~=, and member?. By convention, operations with names ending in "?" return Boolean values.

The usual rules for files are suspended for conditional expressions. In .input files, the then and else keywords can begin in the same column as the corresponding if by may also appear to the right. Each of the following styles of writing if-then-else expressions is acceptable:

```
if i>0 then output("positive") else output("nonpositive")
```

```
if i>0 then output("positive")
  else output("nonpositive")
```

```
if i>0 then output("positive")
else output("nonpositive")
```

```
if i>0
then output("positive")
else output("nonpositive")
```

```
if i>0
  then output("positive")
  else output("nonpositive")
```

A block can follow the then or else keywords. In the following two assignments to a, the then and else clauses each are followed by two line piles. The value returned in each is the value of the second line.

```
a :=
```

```

if i > 0 then
  j := sin(i * pi())
  exp(j + 1/j)
else
  j := cos(i * 0.5 * pi())
  log(abs(j)**5 + i)

a :=
if i > 0
then
  j := sin(i * pi())
  exp(j + 1/j)
else
  j := cos(i * 0.5 * pi())
  log(abs(j)**5 + i)

```

These are both equivalent to the following:

```

a :=
if i > 0 then (j := sin(i * pi()); exp(j + 1/j))
else (j := cos(i * 0.5 * pi()); log(abs(j)**5 + i))

```

—

15.7 syntax iterate

— iterate.help —

```

=====
iterate in loops
=====

```

Axiom provides an `iterate` expression that skips over the remainder of a loop body and starts the next loop execution. We first initialize a counter.

```

i := 0
0
                                     Type: NonNegativeInteger

```

Display the even integers from 2 to 5:

```

repeat
  i := i + 1
  if i > 5 then leave

```



```

    if odd?(i) then iterate
    output(i)
2
4
                                Type: Void

```

15.8 syntax leave

— leave.help —

```

=====
leave in loops
=====

```

The leave keyword is often more useful in terminating a loop. A leave causes control to transfer to the expression immediately following the loop. As loops always return the unique value of Void, you cannot return a value with leave. That is, leave takes no argument.

```

f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then leave
    i := i + 1
  i
                                Type: Void

```

This example is a modification of the last example in the previous section. Instead of using return we'll use leave.

```

f()
7
                                Type: PositiveInteger

```

The loop terminates when factorial(i) gets big enough. The last line of the function evaluates to the corresponding "good" value of i and the function terminates, returning that value.

You can only use leave to terminate the evaluation of one loop. Lets consider a loop within a loop, that is, a loop with a nested loop. First, we initialize two counter variables.

```

(i,j) := (1,1)
1

```

Type: PositiveInteger

```
repeat
  repeat
    if (i + j) > 10 then leave
    j := j + 1
  if (i + j) > 10 then leave
  i := i + 1
```

Type: Void

Nested loops must have multiple leave expressions at the appropriate nesting level. How would you rewrite this so $(i + j) > 10$ is only evaluated once?

```
=====
leave vs => in loop bodies
=====
```

Compare the following two loops:

<pre>i := 1 repeat i := i + 1 i > 3 => i output(i)</pre>	<pre>i := 1 repeat i := i + 1 if i > 3 then leave output(i)</pre>
--	--

In the example on the left, the values 2 and 3 for i are displayed but then the " \Rightarrow " does not allow control to reach the call to output again. The loop will not terminate until you run out of space or interrupt the execution. The variable i will continue to be incremented because the " \Rightarrow " only means to leave the block, not the loop.

In the example on the right, upon reaching 4, the leave will be executed, and both the block and the loop will terminate. This is one of the reasons why both " \Rightarrow " and leave are provided. Using a while clause with the " \Rightarrow " lets you simulate the action of leave.

15.9 syntax parallel

— parallel.help —

```
=====
parallel iteration
=====
```

Sometimes you want to iterate across two lists in parallel, or perhaps you want to traverse a list while incrementing a variable.

The general syntax of a repeat loop is

```
iterator1, iterator2, ..., iteratorN repeat loopbody
```

where each iterator is either a for or a while clause. The loop terminates immediately when the end test of any iterator succeeds or when a leave or return expression is evaluated in loopbody. The value returned by the loop is the unique value of Void.

```
l := [1,3,5,7]
    [1,3,5,7]
                                Type: List PositiveInteger
```

```
m := [100,200]
    [100,200]
                                Type: List PositiveInteger
```

```
sum := 0
    0
                                Type: NonNegativeInteger
```

Here we write a loop to iterate across two lists, computing the sum of the pairwise product of the elements:

```
for x in l for y in m repeat
    sum := sum + x*y
                                Type: Void
```

The last two elements of l are not used in the calculation because m has two fewer elements than l.

```
sum
    700
                                Type: NonNegativeInteger
```

This is the "dot product".

Next we write a loop to compute the sum of the products of the loop elements with their positions in the loop.

```
l := [2,3,5,7,11,13,17,19,23,29,31,37]
    [2,3,5,7,11,13,17,19,23,29,31,37]
                                Type: List PositiveInteger
```

```
sum := 0
    0
```

Type: NonNegativeInteger

```
for i in 0.. for x in l repeat sum := i * x
Type: Void
```

Here looping stops when the list `l` is exhausted, even though the `for i in 0..` specifies no terminating condition.

```
sum
407
```

Type: NonNegativeInteger

When `"|"` is used to qualify any of the `for` clauses in a parallel iteration, the variables in the predicates can be from an outer scope or from a `for` clause in or to the left of the modified clause.

This is correct:

```
for i in 1..10 repeat
  for j in 200..300 | odd? (i+j) repeat
    output [i,j]
```

But this is not correct. The variable `j` has not been defined outside the inner loop:

```
for i in 1..01 | odd? (i+j) repeat -- wrong, j not defined
  for j in 200..300 repeat
    output [i,j]
```

It is possible to mix several of repeat modifying clauses on a loop:

```
for i in 1..10
  for j in 151..160 | odd? j
    while i + j < 160 repeat
      output [i,j]
[1,151]
[3,153]
Type: Void
```

Here are useful rules for composing loop expressions:

1. while predicates can only refer to variables that are global (or in an outer scope) or that are defined in `for` clauses to the left of the predicate.
2. A "such that" predicate (something following `"|"`) must directly follow a `for` clause and can only refer to variables that are global (or in an outer scope) or defined in the modified `for` clause or any `for` clause to the left.

15.10 syntax repeat

— repeat.help —

Repeat Loops

A loop is an expression that contains another expression, called the loop body, which is to be evaluated zero or more times. All loops contain the repeat keyword and return the unique value of Void. Loops can contain inner loops to any depth.

The most basic loop is of the form

```
repeat loopbody
```

Unless loopbody contains a leave or return expression, the loop repeats forever. The value returned by the loop is the unique value of Void.

Axiom tries to determine completely the type of every object in a loop and then to translate the loop body to Lisp or even to machine code. This translation is called compilation.

If Axiom decides that it cannot compile the loop, it issues a message stating the problem and then the following message:

```
We will attempt to step through and interpret the code
```

It is still possible that Axiom can evaluate the loop but in interpret-code mode.

Return in Loops

A return expression is used to exit a function with a particular value. In particular, if a return is in a loop within the function, the loop is terminated whenever the return is evaluated.

```
f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then return i
  i := i + 1
```

Type: Void

f()

Type: Void

When factorial(i) is big enough, control passes from inside the loop all the way outside the function, returning the value of i (so we think). What went wrong? Isn't it obvious that this function should return an integer? Well, Axiom makes no attempt to analyze the structure of a loop to determine if it always returns a value because, in general, this is impossible. So Axiom has this simple rule: the type of the function is determined by the type of its body, in this case a block. The normal value of a block is the value of its last expression, in this case, a loop. And the value of every loop is the unique value of Void. So the return type of f is Void.

There are two ways to fix this. The best way is for you to tell Axiom what the return type of f is. You do this by giving f a declaration

```
f:() -> Integer
```

prior to calling for its value. This tells Axiom "trust me -- an integer is returned". Another way is to add a dummy expression as follows.

```
f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then return i
    i := i + 1
  0
Type: Void
```

Note that the dummy expression will never be evaluated but it is the last expression in the function and will determine the return type.

```
f()
7
Type: PositiveInteger
```

```
=====
leave in loops
=====
```

The leave keyword is often more useful in terminating a loop. A leave causes control to transfer to the expression immediately following the loop. As loops always return the unique value of Void, you cannot return a value with leave. That is, leave takes no argument.

```
f() ==
  i := 1
```

```

repeat
  if factorial(i) > 1000 then leave
  i := i + 1
i

```

Type: Void

This example is a modification of the last example in the previous section. Instead of using return we'll use leave.

```

f()
7

```

Type: PositiveInteger

The loop terminates when factorial(i) gets big enough. The last line of the function evaluates to the corresponding "good" value of i and the function terminates, returning that value.

You can only use leave to terminate the evaluation of one loop. Lets consider a loop within a loop, that is, a loop with a nested loop. First, we initialize two counter variables.

```

(i,j) := (1,1)
1

```

Type: PositiveInteger

```

repeat
  repeat
    if (i + j) > 10 then leave
    j := j + 1
  if (i + j) > 10 then leave
  i := i + 1

```

Type: Void

Nested loops must have multiple leave expressions at the appropriate nesting level. How would you rewrite this so (i + j) > 10 is only evaluated once?

```

=====
leave vs => in loop bodies
=====

```

Compare the following two loops:

<pre> i := 1 repeat i := i + 1 i > 3 => i output(i) </pre>	<pre> i := 1 repeat i := i + 1 if i > 3 then leave output(i) </pre>
--	--

In the example on the left, the values 2 and 3 for i are displayed but

then the "`=>`" does not allow control to reach the call to output again. The loop will not terminate until you run out of space or interrupt the execution. The variable `i` will continue to be incremented because the "`=>`" only means to leave the block, not the loop.

In the example on the right, upon reaching 4, the leave will be executed, and both the block and the loop will terminate. This is one of the reasons why both "`=>`" and `leave` are provided. Using a while clause with the "`=>`" lets you simulate the action of `leave`.

```
=====
iterate in loops
=====
```

Axiom provides an `iterate` expression that skips over the remainder of a loop body and starts the next loop execution. We first initialize a counter.

```
i := 0
0
                                     Type: NonNegativeInteger
```

Display the even integers from 2 to 5:

```
repeat
  i := i + 1
  if i > 5 then leave
  if odd?(i) then iterate
  output(i)
2
4
                                     Type: Void
```

Also See:

- o)help blocks
- o)help if
- o)help while
- o)help for
- o)help suchthat
- o)help parallel
- o)help lists

² "blocks" (15.2 p 388) "if" (15.6 p 397) "while" (65.1.2 p 995) "for" (15.5 p 393) "suchthat" (15.11 p 408) "parallel" (15.9 p 401) "lists" (?? p ??)

15.11 syntax suchthat

— suchthat.help —

```
=====
Such that predicates
=====
```

A for loop can be followed by a "|" and then a predicate. The predicate qualifies the use of the values from the iterator that follows the for. Think of the vertical bar "|" as the phrase "such that".

```
for n in 0..4 | odd? n repeat output n
1
3
```

Type: Void

This loop expression prints out the integers n in the given segment such that n is odd.

A for loop can also be written

```
for iterator | predicate repeat loopbody
```

which is equivalent to:

```
for iterator repeat if predicate then loopbody else iterate
```

The predicate need not refer only to the variable in the for clause. Any variable in an outer scope can be part of the predicate.

```
for i in 1..50 repeat
  for j in 1..50 | factorial(i+j) < 25 repeat
    output [i,j]
[1,1]
[1,2]
[1,3]
[2,1]
[2,2]
[3,1]
```

Type: Void

—————

15.12 syntax syntax

— syntax.help —

The Axiom Interactive Language has the following features documented here.

More information is available by typing

```
)help feature
```

where feature is one of:

```
assignment -- Immediate and delayed assignments
blocks      -- Blocks of expressions
collection  -- creating lists with iterators
for          -- for loops
if           -- If-then-else statements
iterate     -- using iterate in loops
leave       -- using leave in loops
parallel    -- parallel iterations
repeat      -- repeat loops
suchthat    -- suchthat predicates
while       -- while loops
```

—

15.13 syntax while

— while.help —

```
=====
while loops
=====
```

The repeat in a loop can be modified by adding one or more while clauses. Each clause contains a predicate immediately following the while keyword. The predicate is tested before the evaluation of the body of the loop. The loop body is evaluated whenever the predicate in a while clause is true.

The syntax for a simple loop using while is

```
while predicate repeat loopbody
```

The predicate is evaluated before loopbody is evaluated. A while loop terminates immediately when predicate evaluates to false or when a leave or return expression is evaluated. See `)help repeat` for more information on leave and return.

Here is a simple example of using while in a loop. We first initialize the counter.

```
i := 1
1
                                     Type: PositiveInteger

while i < 1 repeat
  output "hello"
  i := i + 1
                                     Type: Void
```

The steps involved in computing this example are

- (1) set i to 1
- (2) test the condition `i < 1` and determine that it is not true
- (3) do not evaluate the loop body and therefore do not display "hello"

```
(x, y) := (1, 1)
1
                                     Type: PositiveInteger
```

If you have multiple predicates to be tested use the logical and operation to separate them. Axiom evaluates these predicates from left to right.

```
while x < 4 and y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2
[1,1]
[2,3]
[3,5]
                                     Type: Void
```

A leave expression can be included in a loop body to terminate a loop even if the predicate in any while clauses are not false.

```
(x, y) := (1, 1)
1
                                     Type: PositiveInteger
```

```
while x < 4 and y < 10 repeat
  if x + y > 7 then leave
```

```
output [x,y]
x := x + 1
y := y + 2
[1,1]
[2,3]
```

Type: Void

—————▶

Chapter 16

Abstract Syntax Trees (ptrees)

Abstract Syntax Trees

These functions create and examine abstract syntax trees. These are called pform, for short.

!! This file also contains constructors for concrete syntax, although
!! they should be somewhere else.

THE PFORM DATA STRUCTURE

Leaves: [hd, tok, pos]
Trees: [hd, tree, tree, ...]
hd is either an id or (id . alist)

16.0.1 defun Construct a leaf token

The tokConstruct function is a constructor and selectors for leaf tokens. A leaf token looks like [head, token, position] where head is either an id or (id . alist) [ifcar p??]

[pfNoPosition? p414]

[ncPutQ p418]

— defun tokConstruct —

```
(defun |tokConstruct| (head token &rest position)
  (let (result)
    (setq result (cons head token))
    (cond
      ((ifcar position)
       (cond
         ((|pfNoPosition?| (car position)) result)
         (t (|ncPutQ| result '|posn| (car position)) result)))
```

```
(t result))))
```

16.0.2 defun Return a part of a node

```
[ifcar p??]
```

— defun pfAbSynOp —

```
(defun |pfAbSynOp| (form)
  (let (hd)
    (setq hd (car form))
    (or (ifcar hd) hd)))
```

16.0.3 defun Compare a part of a node

```
[eqcar p??]
```

— defun pfAbSynOp? —

```
(defun |pfAbSynOp?| (form op)
  (let (hd)
    (setq hd (car form))
    (or (eq hd op) (eqcar hd op))))
```

16.0.4 defun pfNoPosition?

```
[poNoPosition? p415]
```

— defun pfNoPosition? —

```
(defun |pfNoPosition?| (pos)
  (|poNoPosition?| pos))
```

16.0.5 defun poNoPosition?

[eqcar p??]

— defun poNoPosition? 0 —

```
(defun |poNoPosition?| (pos)
  (eqcar pos '|noPosition|))
```

—————

16.0.6 defun tokType

[ncTag p417]

— defun tokType —

```
(defun |tokType| (x) (|ncTag| x))
```

—————

16.0.7 defun tokPart

— defun tokPart 0 —

```
(defun |tokPart| (x) (cdr x))
```

—————

16.0.8 defun tokPosn

[qassq p??]

[ncAlist p417]

[pfNoPosition p416]

— defun tokPosn —

```
(defun |tokPosn| (x)
  (let (a)
    (setq a (qassq '|posn| (|ncAlist| x)))
    (cond
```



```
(a (cdr a))
(t (|pfNoPosition|))))
```

16.0.9 defun pfNoPosition

[poNoPosition p416]

— defun pfNoPosition —

```
(defun |pfNoPosition| () (|poNoPosition|))
```

16.0.10 defun poNoPosition

[\$nopus p28]

— defun poNoPosition 0 —

```
(defun |poNoPosition| ()
  (declare (special |$nopus|))
  |$nopus|)
```

Chapter 17

Attributed Structures

For objects which are pairs where the CAR field is either just a tag (an identifier) or a pair which is the tag and an association list.

17.0.11 defun ncTag

Pick off the tag [ncBug p370]

[qcar p??]

[identp p1003]

— defun ncTag —

```
(defun |ncTag| (x)
  (cond
    ((null (consp x)) (|ncBug| 's2cb0031 nil))
    (t
     (setq x (qcar x))
     (cond
      ((identp x) x)
      ((null (consp x)) (|ncBug| 's2cb0031 nil))
      (t (qcar x))))))
```

—————

17.0.12 defun ncAlist

Pick off the property list [ncBug p370]

[qcar p??]

[identp p1003]

[qcdr p??]

— defun ncAlist —

```
(defun |ncAlist| (x)
  (cond
    ((null (consp x)) (|ncBug| 's2cb0031 nil))
    (t
     (setq x (qcar x))
     (cond
       ((identp x) nil)
       ((null (consp x)) (|ncBug| 's2cb0031 nil))
       (t (qcdr x))))))
```

17.0.13 defun ncEltQ

Get the entry for key k on x's association list [qassq p??]
 [ncAlist p417]
 [ncBug p370]

— defun ncEltQ —

```
(defun |ncEltQ| (x k)
  (let (r)
    (setq r (qassq k (|ncAlist| x)))
    (cond
      ((null r) (|ncBug| 's2cb0007 (list k)))
      (t (cdr r)))))
```

17.0.14 defun ncPutQ

```
;;-- Put (k . v) on the association list of x and return v
;;-- case1: ncPutQ(x,k,v) where k is a key (an identifier), v a value
;;--       put the pair (k . v) on the association list of x and return v
;;-- case2: ncPutQ(x,k,v) where k is a list of keys, v a list of values
;;--       equivalent to [ncPutQ(x,key,val) for key in k for val in v]
ncPutQ(x,k,v) ==
;   LISTP k =>
;     for key in k for val in v repeat ncPutQ(x,key,val)
;     v
;   r := QASSQ(k,ncAlist x)
;   if NULL r then
```

```

;      r := CONS( CONS(k,v), ncAlist x)
;      RPLACA(x,CONS(ncTag x,r))
;      else
;      RPLACD(r,v)
;      v

```

```

[qassq p??]
[ncAlist p417]
[ncTag p417]

```

— defun ncPutQ —

```

(defun |ncPutQ| (x k v)
  (let (r)
    (cond
      ((listp k)
       ((lambda (Var1 key Var2 val)
          (loop
            (cond
              ((or (atom Var1)
                   (progn (setq key (car Var1)) nil)
                          (atom Var2)
                          (progn (setq val (car Var2)) nil))
               (return nil))
            (t
             (|ncPutQ| x key val)))
          (setq Var1 (cdr Var1))
          (setq Var2 (cdr Var2))))
        k nil v nil)
      v)
    (t
     (setq r (qassq k (|ncAlist| x)))
     (cond
       ((null r)
        (setq r (cons (cons k v) (|ncAlist| x)))
        (rplaca x (cons (|ncTag| x) r)))
       (t
        (rplacd r v)))
     v))))

```

—

Chapter 18

System Command Handling

The system commands are the top-level commands available in Axiom that can all be invoked by prefixing the symbol with a closed-paren. Thus, to see they copyright you type:

```
)copyright
```

New commands need to be added to this table. The command invoked will be the first entry of the pair and the “user level” of the command will be the second entry.

See:

- The “abbreviations” (19.2.1 p 463) command
- The “boot” (5.1.8 p 25) command
- The “browse” (?? p ??) command
- The “cd” (?? p ??) command
- The “clear” (23.3.1 p 479) command
- The “close” (24.2.2 p 488) command
- The “compile” (?? p ??) command
- The “copyright” (26.2.1 p 500) command
- The “credits” (27.3.1 p 503) command
- The “display” (29.2.1 p 513) command
- The “edit” (30.2.1 p 522) command
- The “fin” (31.1.1 p 526) command

- The “frame” (32.5.16 p 543) command
- The “help” (33.2.1 p 550) command
- The “history” (34.4.7 p 560) command
- The “lisp” (?? p ??) command
- The “library” (63.1.34 p 971) command
- The “load” (38.1.1 p 607) command
- The “ltrace” (39.1.1 p 610) command
- The “pquit” (40.2.1 p 612) command
- The “quit” (41.2.1 p 616) command
- The “read” (42.1.1 p 620) command
- The “savesystem” (43.1.1 p 624) command
- The “set” (44.37.1 p 785) command
- The “show” (45.1.1 p 792) command
- The “spool” (?? p ??) command
- The “summary” (47.1.1 p 806) command
- The “synonym” (48.1.1 p 808) command
- The “system” (?? p ??) command
- The “trace” (50.1.7 p 821) command
- The “trademark” (26.2.2 p 501) command
- The “undo” (51.3.6 p 886) command
- The “what” (52.1.2 p 903) command
- The “with” (53.1.1 p 911) command
- The “workfiles” (54.1.1 p 913) command
- The “zsystemdevelopment” (55.1.1 p 917) command

18.1 Variables Used

18.1.1 defvar \$systemCommands

— initvars —

```
(defvar |$systemCommands| nil)
```

—————

— postvars —

```
(eval-when (eval load)
  (setq |$systemCommands|
    '(
```

(abbreviations	. compiler)
(boot	. development)
(browse	. development)
(cd	. interpreter)
(clear	. interpreter)
(close	. interpreter)
(compiler	. compiler)
(copyright	. interpreter)
(credits	. interpreter)
(describe	. interpreter)
(display	. interpreter)
(edit	. interpreter)
(fin	. development)
(frame	. interpreter)
(help	. interpreter)
(history	. interpreter)
(lisp	. development)
(library	. interpreter)
(load	. interpreter)
(ltrace	. interpreter)
(pquit	. interpreter)
(quit	. interpreter)
(read	. interpreter)
(savesystem	. interpreter)
(set	. interpreter)
(show	. interpreter)
(spool	. interpreter)
(summary	. interpreter)
(synonym	. interpreter)
(system	. interpreter)
(trace	. interpreter)


```

(|trademark|                . |interpreter|)
(|undo|                      . |interpreter|)
(|what|                      . |interpreter|)
(|with|                      . |interpreter|)
(|workfiles|                . |development|)
(|zsystemdevelopment|       . |interpreter|)
)))

```

18.1.2 defvar \$syscommands

This table is used to look up a symbol to see if it might be a command.

— **initvars** —

```
(defvar $syscommands nil)
```

— **postvars** —

```
(eval-when (eval load)
  (setq $syscommands (mapcar #'car |$systemCommands|)))
```

18.1.3 defvar \$noParseCommands

This is a list of the commands which have their arguments passed verbatim. Certain functions, such as the lisp function need to be able to handle all kinds of input that will not be acceptable to the interpreter.

— **initvars** —

```
(defvar |$noParseCommands| nil)
```

— **postvars** —

```
(eval-when (eval load)
  (setq |$noParseCommands|
```

```
'(|boot| |copyright| |credits| |fin| |lisp| |pquit| |quit|  
  |synonym| |system| |trademark| )))
```

18.2 Functions

18.2.1 defun handleNoParseCommands

The system commands given by the global variable `$noParseCommands` require essentially no preprocessing/parsing of their arguments. Here we dispatch the functions which implement these commands.

There are four standard commands which receive arguments

- boot
- lisp
- synonym
- system

There are six standard commands which do not receive arguments –

- quit
- fin
- pquit
- credits
- copyright
- trademark

As these commands do not necessarily exhaust those mentioned in `$noParseCommands`, we provide a generic dispatch based on two conventions: commands which do not require an argument name themselves, those which do have their names prefixed by “np”. This makes it possible to dynamically define new system commands provided you handle the argument parsing.

18.2.2 defun Handle a top level command

```
[concat p1003]
[expand-tabs p??]
[processSynonyms p33]
[substring p??]
[getFirstWord p449]
[unAbbreviateKeyword p449]
[member p1004]
[handleNoParseCommands p425]
[splitIntoOptionBlocks p427]
[handleTokenSizeSystemCommands p427]
[handleParsedSystemCommands p448]
[$tokenCommands p455]
[$noParseCommands p424]
[line p??]
```

— defun doSystemCommand —

```
(defun |doSystemCommand| (string)
  (let (line tok unab optionList)
    (declare (special line |$tokenCommands| |$noParseCommands|))
    (setq string (concat "") (expand-tabs string)))
    (setq line string)
    (|processSynonyms|)
    (setq string line)
    (setq string (substring string 1 nil))
    (cond
      ((string= string "") nil)
      (t
       (setq tok (|getFirstWord| string))
       (cond
         (tok
          (setq unab (|unAbbreviateKeyword| tok))
          (cond
            ((|member| unab |$noParseCommands|)
             (|handleNoParseCommands| unab string))
            (t
             (setq optionList (|splitIntoOptionBlocks| string))
             (cond
               ((|member| unab |$tokenCommands|)
                (|handleTokenSizeSystemCommands| unab optionList))
               (t
                (|handleParsedSystemCommands| unab optionList)
                nil))))))
          (t nil))))))
```

—————

18.2.3 defun Split block into option block

[stripSpaces p451]

— defun splitIntoOptionBlocks —

```
(defun |splitIntoOptionBlocks| (str)
  (let (inString block (blockStart 0) (parenCount 0) blockList)
    (dotimes (i (1- (|#| str)))
      (cond
        ((char= (elt str i) #" " ) (setq inString (null inString)))
        (t
         (when (and (char= (elt str i) #\"()\" ) (null inString))
           (incf parenCount))
         (when (and (char= (elt str i) #\"\\\" ) (null inString))
           (decf parenCount))
         (when
          (and (char= (elt str i) #\"\\\" )
               (null inString)
               (= parenCount -1))
            (setq block (|stripSpaces| (subseq str blockStart i)))
            (setq blockList (cons block blockList))
            (setq blockStart (1+ i))
            (setq parenCount 0))))))
    (setq blockList (cons (|stripSpaces| (subseq str blockStart)) blockList))
    (nreverse blockList)))
```

—

18.2.4 defun Tokenize a system command

[dumbTokenize p447]

[tokTran p447]

[systemCommand p428]

— defun handleTokenizeSystemCommands —

```
(defun |handleTokenizeSystemCommands| (unabr optionList)
  (declare (ignore unabr))
  (let (parcmd)
    (setq optionList (mapcar #'(lambda (x) (|dumbTokenize| x)) optionList))
    (setq parcmd
      (mapcar #'(lambda (opt) (mapcar #'(lambda (tok) (|tokTran| tok)) opt))
              optionList))
    (when parcmd (|systemCommand| parcmd))))
```

—

18.2.5 defun Handle system commands

You can type “)?” and see trivial help information. You can type “)? compile” and see compiler related information [selectOptionLC p459]

```
[helpSpad2Cmd p550]
[selectOption p459]
[commandsForUserLevel p428]
[$options p??]
[$e p??]
[$systemCommands p423]
[$syscommands p424]
[$CategoryFrame p??]
```

— defun systemCommand —

```
(defun |systemCommand| (cmd)
  (let (|$options| |$e| op argl options fun)
    (declare (special |$options| |$e| |$systemCommands| $syscommands
                      |$CategoryFrame|))
    (setq op (caar cmd))
    (setq argl (cdar cmd))
    (setq options (cdr cmd))
    (setq |$options| options)
    (setq |$e| |$CategoryFrame|)
    (setq fun (|selectOptionLC| op $syscommands '|commandError|))
    (if (and argl (eq (elt argl 0) '?)) (nequal fun '|synonym|))
        (|helpSpad2Cmd| (cons fun nil))
    (progn
      (setq fun
        (|selectOption| fun (|commandsForUserLevel| |$systemCommands|)
                          '|commandUserLevelError|))
      (funcall fun argl))))
```

—————

18.2.6 defun Select commands matching this user level

The `$UserLevel` variable contains one of three values: `compiler`, `development`, or `interpreter`. This variable is used to select a subset of commands from the list stored in `$systemCommands`, representing all of the commands that are valid for this level. [satisfiesUserLevel p431]

— defun commandsForUserLevel —

```
(defun |commandsForUserLevel| (arg)
  (let (c)
    (dolist (pair arg)
```

```
(when (|satisfiesUserLevel| (cdr pair))
      (setq c (cons (car pair) c)))
(nreverse c))
```

18.2.7 defun No command begins with this string

[commandErrorMessage p429]

— **defun commandError** —

```
(defun |commandError| (x u)
  (|commandErrorMessage| '|command| x u))
```

18.2.8 defun No option begins with this string

[commandErrorMessage p429]

— **defun optionError** —

```
(defun |optionError| (x u)
  (|commandErrorMessage| '|option| x u))
```

18.2.9 defvar \$oldline

— **initvars** —

```
(defvar $oldline nil "used to output command lines")
```

18.2.10 defun No command/option begins with this string

[commandAmbiguityError p432]
[sayKeyedMsg p331]

```
[terminateSystemCommand p432]
[$oldline p429]
[line p??]
```

— **defun commandErrorMessage** —

```
(defun |commandErrorMessage| (kind x u)
  (declare (special $oldline line))
  (setq $oldline line)
  (if u
    (|commandAmbiguityError| kind x u)
    (progn
      (|sayKeyedMsg| 'S2IZ0008 (list kind x))
      (|terminateSystemCommand|))))
```

—————

18.2.11 defun Option not available at this user level

```
[userLevelErrorMessage p430]
```

— **defun optionUserLevelError** —

```
(defun |optionUserLevelError| (x u)
  (|userLevelErrorMessage| ' |option| x u))
```

—————

18.2.12 defun Command not available at this user level

```
[userLevelErrorMessage p430]
```

— **defun commandUserLevelError** —

```
(defun |commandUserLevelError| (x u)
  (|userLevelErrorMessage| ' |command| x u))
```

—————

18.2.13 defun Command not available error message

```
[commandAmbiguityError p432]
[sayKeyedMsg p331]
```

[terminateSystemCommand p432]
 [\$UserLevel p784]

— **defun userLevelErrorMessage** —

```
(defun |userLevelErrorMessage| (kind x u)
  (declare (special |$UserLevel|))
  (if u
    (|commandAmbiguityError| kind x u)
    (progn
      (|sayKeyedMsg| 'S2IZ0007 (list |$UserLevel| kind))
      (|terminateSystemCommand|))))
```

—————

18.2.14 defun satisfiesUserLevel

[UserLevel p784]

— **defun satisfiesUserLevel 0** —

```
(defun |satisfiesUserLevel| (x)
  (declare (special |$UserLevel|))
  (cond
    ((eq x '|interpreter|) t)
    ((eq |$UserLevel| '|interpreter|) nil)
    ((eq x '|compiler|) t)
    ((eq |$UserLevel| '|compiler|) nil)
    (t t)))
```

—————

18.2.15 defun hasOption

[stringPrefix? p??]
 [pname p1001]

— **defun hasOption** —

```
(defun |hasOption| (al opt)
  (let ((optPname (pname opt)) found)
    (loop for pair in al do
      (when (|stringPrefix?| (pname (car pair)) optPname) (setq found pair))
      until found)
    found))
```

18.2.16 defun terminateSystemCommand

[tersyscommand p432]

— defun terminateSystemCommand —

```
(defun |terminateSystemCommand| nil (tersyscommand))
```

18.2.17 defun Terminate a system command

[spadThrow p??]

— defun tersyscommand —

```
(defun tersyscommand ()
  (fresh-line)
  (setq chr 'endoflinechr)
  (setq tok 'end_unit)
  (|spadThrow|))
```

18.2.18 defun commandAmbiguityError

[sayKeyedMsg p331]

[sayMSG p333]

[bright p??]

[terminateSystemCommand p432]

— defun commandAmbiguityError —

```
(defun |commandAmbiguityError| (kind x u)
  (|sayKeyedMsg| 's2iz0009 (list kind x))
  (dolist (a u) (|sayMSG| (cons " " (|bright| a))))
  (|terminateSystemCommand|))
```

18.2.19 defun getParserMacroNames

The `$pfMacros` is a list of all of the user-defined macros. [`$pfMacros` p100]

— **defun getParserMacroNames 0** —

```
(defun |getParserMacroNames| ()
  (declare (special |$pfMacros|))
  (remove-duplicates (mapcar #'car |$pfMacros|)))
```

18.2.20 defun clearParserMacro

Note that if a macro is defined twice this will clear the last instance. Thus:

```
a ==> 3
a ==> 4
)d macros
a ==> 4
)clear prop a
)d macros
a ==> 3
)clear prop a
)d macros
nil
```

```
[ifcdr p??]
[assoc p??]
[remalist p??]
[$pfMacros p100]
```

— **defun clearParserMacro** —

```
(defun |clearParserMacro| (macro)
  (declare (special |$pfMacros|))
  (when (ifcdr (|assoc| macro |$pfMacros|))
    (setq |$pfMacros| (remalist |$pfMacros| macro))))
```

18.2.21 defun displayMacro

```
[isInterpMacro p??]
[sayBrightly p??]
```

```
[bright p??]
[strconc p??]
[object2String p??]
[mathprint p??]
[$op p??]
```

— **defun displayMacro** —

```
(defun |displayMacro| (name)
  (let (|$op| m body args)
    (declare (special |$op|))
    (setq m (|isInterpMacro| name))
    (cond
      ((null m)
        (|sayBrightly|
          (cons " " (append (|bright| name)
                           (cons "is not an interpreter macro." nil))))))
      (t
        (setq |$op| (strconc "macro " (|object2String| name)))
        (setq args (car m))
        (setq body (cdr m))
        (setq args
          (cond
            ((null args) nil)
            ((null (cdr args)) (car args))
            (t (cons '|Tuple| args)))))
        (|mathprint| (cons 'map (cons (cons args body) nil)))))))
```

—————

18.2.22 defun displayWorkspaceNames

```
[getInterpMacroNames p??]
[getParserMacroNames p433]
[sayMessage p??]
[msort p??]
[getWorkspaceNames p435]
[sayAsManyPerLineAsPossible p??]
[sayBrightly p??]
[setdifference p??]
```

— **defun displayWorkspaceNames** —

```
(defun |displayWorkspaceNames| ()
  (let (pmacs names imacs)
    (setq imacs (|getInterpMacroNames|))
```

```
(setq pmacs (|getParserMacroNames|))
(|sayMessage| "Names of User-Defined Objects in the Workspace:")
(setq names (msort (append (|getWorkspaceNames|) pmacs)))
(if names
  (|sayAsManyPerLineAsPossible| (mapcar #'|object2String| names))
  (|sayBrightly| " * None *"))
(setq imacs (setdifference imacs pmacs))
(when imacs
  (|sayMessage| "Names of System-Defined Objects in the Workspace:")
  (|sayAsManyPerLineAsPossible| (mapcar #'|object2String| imacs))))
```

18.2.23 defun getWorkspaceNames

```
;getWorkspaceNames() ==
; NMSORT [n for [n,..] in CAAR $InteractiveFrame |
;   (n ^= "--macros--" and n^= "--flags--")]
```

```
[nequal p??]
[seq p??]
[nmsort p??]
[exit p??]
[$InteractiveFrame p??]
```

— defun getWorkspaceNames —

```
(defun |getWorkspaceNames| ()
  (PROG (n)
    (declare (special |$InteractiveFrame|))
    (RETURN
      (SEQ (NMSORT (PROG (G166322)
        (setq G166322 NIL)
        (RETURN
          (DO ((G166329 (CAAR |$InteractiveFrame|)
            (CDR G166329))
              (G166313 NIL))
            ((OR (ATOM G166329)
              (PROGN
                (SETQ G166313 (CAR G166329))
                NIL)
              (PROGN
                (PROGN
                  (setq n (CAR G166313))
                  G166313)
                NIL))
            (NREVERSEO G166322))
```

```
(SEQ (EXIT (COND
  ((AND (NEQUAL n '--macros--)
    (NEQUAL n '--flags--))
    (SETQ G166322
      (CONS n G166322))))))))))
```

18.2.24 defun fixObjectForPrinting

The \$msgdbPrims variable is set to:

```
(|%b| |%d| |%l| |%i| |%u| %U |%n| |%x| |%ce| |%rj|
 "%U" "%b" "%d" "%l" "%i" "%u" "%U" "%n" "%x" "%ce" "%rj")
```

```
[object2Identifier p??]
[member p1004]
[strconc p??]
[pname p1001]
[$msgdbPrims p329]
```

— defun fixObjectForPrinting —

```
(defun |fixObjectForPrinting| (v)
  (let (vp)
    (declare (special |$msgdbPrims|))
    (setq vp (|object2Identifier| v))
    (cond
      ((eq vp '%) "\\%")
      ((|member| vp |$msgdbPrims|) (strconc "\\" (pname vp)))
      (t v))))
```

18.2.25 defun displayProperties,sayFunctionDeps

```
;displayProperties(option,l) ==
; $dependentAlist : local := nil
; $dependeeAlist : local := nil
; [opt,:vl]:= (l or ['properties])
; imacs := getInterpMacroNames()
; pmacs := getParserMacroNames()
; macros := REMDUP append(imacs, pmacs)
; if vl is ['all] or null vl then
;   vl := MSORT append(getWorkspaceNames(),macros)
```

```

; if $frameMessages then sayKeyedMsg("S2IZ0065",[$interpreterFrameName])
; null vl =>
;   null $frameMessages => sayKeyedMsg("S2IZ0066",NIL)
;   sayKeyedMsg("S2IZ0067",[$interpreterFrameName])
; interpFunctionDepAlists()
; for v in vl repeat
;   isInternalMapName(v) => 'iterate
;   pl := getIProplist(v)
;   option = 'flags =>      getAndSay(v,"flags")
;   option = 'value =>      displayValue(v,getI(v,'value),nil)
;   option = 'condition => displayCondition(v,getI(v,"condition"),nil)
;   option = 'mode =>       displayMode(v,getI(v,'mode),nil)
;   option = 'type =>       displayType(v,getI(v,'value),nil)
;   option = 'properties =>
;     v = "--flags--" => nil
;     pl is [ ['cacheInfo,:.],:.] => nil
;     v1 := fixObjectForPrinting(v)
;     sayMSG ["Properties of",:bright prefix2String v1,':"]
;     null pl =>
;       v in pmacs =>
;         sayMSG '"    This is a user-defined macro.'"
;         displayParserMacro v
;       isInterpMacro v =>
;         sayMSG '"    This is a system-defined macro.'"
;         displayMacro v
;       sayMSG '"    none"
;   propsSeen:= nil
;   for [prop,:val] in pl | ^MEMQ(prop,propsSeen) and val repeat
;     prop in '(alias generatedCode IS_-GENSYM mapBody localVar) =>
;       nil
;     prop = 'condition =>
;       displayCondition(prop,val,true)
;     prop = 'recursive =>
;       sayMSG '"    This is recursive.'"
;     prop = 'isInterpreterFunction =>
;       sayMSG '"    This is an interpreter function.'"
;     sayFunctionDeps v where
;       sayFunctionDeps x ==
;       if dependents := GETALIST($dependentAlist,x) then
;         null rest dependents =>
;           sayMSG ['"    The following function or rule ",
;             '"depends on this:":bright first dependents]
;           sayMSG
;             '"    The following functions or rules depend on this:"
;           msg := ["%b",'"    "]
;           for y in dependents repeat msg := ['" ",y,:msg]
;           sayMSG [:nreverse msg,"%d"]
;       if dependees := GETALIST($dependeeAlist,x) then
;         null rest dependees =>
;           sayMSG ['"    This depends on the following function ",

```

```

;          "or rule:",:bright first dependees]
;      sayMSG
;      "    This depends on the following functions or rules:"
;      msg := ["%b", "    "]
;      for y in dependees repeat msg := [" " ,y,:msg]
;      sayMSG [:nreverse msg,"%d"]
;      prop = 'isInterpreterRule =>
;      sayMSG "    This is an interpreter rule."
;      sayFunctionDeps v
;      prop = 'localModemap =>
;      displayModemap(v,val,true)
;      prop = 'mode =>
;      displayMode(prop,val,true)
;      prop = 'value =>
;      val => displayValue(v,val,true)
;      sayMSG [" " ,prop,"": " ,val]
;      propsSeen:= [prop,:propsSeen]
;      sayKeyedMsg("S2IZ0068",[option])
;      terminateSystemCommand()

```

```

[seq p??]
[getalist p??]
[exit p??]
[sayMSG p333]
[bright p??]
[$dependeeAlist p??]
[$dependentAlist p??]

```

— defun displayProperties,sayFunctionDeps —

```

(defun |displayProperties,sayFunctionDeps| (x)
  (prog (dependents dependees msg)
    (declare (special |$dependeeAlist| |$dependentAlist|))
    (return
      (seq
        (if (setq dependents (getalist |$dependentAlist| x))
          (seq
            (if (null (cdr dependents))
              (exit
                (|sayMSG| (cons "    The following function or rule "
                              (cons "depends on this:" (|bright| (car dependents))))))
              (|sayMSG| "    The following functions or rules depend on this:")
              (setq msg (cons '|%b| (cons "    " nil)))
              (do ((G166397 dependents (cdr G166397)) (y nil))
                ((or (atom G166397) (progn (setq y (car G166397)) nil)) nil)
                (seq (exit (setq msg (cons " " (cons y msg))))))
              (exit (|sayMSG| (append (nreverse msg) (cons '|%d| nil))))))
          nil)
        (exit

```

```

(if (setq dependees (getalist |$dependeeAlist| x))
  (seq
    (if (null (cdr dependees))
      (exit
        (|sayMSG| (cons "    This depends on the following function "
          (cons "or rule:" (|bright| (car dependees)))))))
      (|sayMSG| "    This depends on the following functions or rules:")
      (setq msg (cons '|%b| (cons "    " nil)))
      (do ((G166406 dependees (cdr G166406)) (y nil))
        ((or (atom G166406) (progn (setq y (car G166406)) nil)) nil)
        (seq (exit (setq msg (cons " " (cons y msg))))))
      (exit (|sayMSG| (append (nreverse msg) (cons '|%d| nil)))))
    nil))))))

```

18.2.26 defun displayValue

```

[sayMSG p333]
[fixObjectForPrinting p436]
[pname p1001]
[objValUnwrap p??]
[objMode p??]
[displayRule p??]
[strconc p??]
[prefix2String p??]
[objMode p??]
[getdatabase p967]
[concat p1003]
[form2String p??]
[mathprint p??]
[outputFormat p??]
[objMode p??]
[$op p??]
[$EmptyMode p??]

```

— defun displayValue —

```

(defun |displayValue| (|$op| u omitVariableNameIfTrue)
  (declare (special |$op|))
  (let (expr op rhs label labmode)
    (declare (special |$EmptyMode|))
    (if (null u)
      (|sayMSG|
        (list '|    Value of | (|fixObjectForPrinting| (pname |$op|)) ": (none)"))
      (progn

```



```

(setq expr (|objValUnwrap| u))
(if (or (and (pairp expr) (progn (setq op (qcar expr)) t) (eq op 'map))
      (equal (|objMode| u) |$EmptyMode|))
    (|displayRule| |$op| expr)
    (progn
      (cond
        (omitVariableNameIfTrue
         (setq rhs ": ")
         (setq label "Value (has type ")
         (t
          (setq rhs ": ")
          (setq label (strconc "Value of " (pname |$op|) ": "))))
        (setq labmode (|prefix2String| (|objMode| u)))
        (when (atom labmode) (setq labmode (list labmode)))
        (if (eq (getdatabase expr 'constructorkind) '|domain|)
            (|sayMSG| (|concat| " " label labmode rhs (|form2String| expr)))
            (|mathprint|
             (cons 'concat
                  (cons label
                       (append labmode
                              (cons rhs
                                   (cons (|outputFormat| expr (|objMode| u)) nil)))))))
        nil))))))

```

18.2.27 defun displayType

```

[sayMSG p333]
[fixObjectForPrinting p436]
[pname p1001]
[prefix2String p??]
[objMode p??]
[concat p1003]
[$op p??]

```

— defun displayType —

```

(defun |displayType| (|$op| u omitVariableNameIfTrue)
  (declare (special |$op|) (ignore omitVariableNameIfTrue))
  (let (type)
    (if (null u)
        (|sayMSG|
         (list " Type of value of " (|fixObjectForPrinting| (pname |$op|))
              ": (none)"))
        (progn
         (setq type (|prefix2String| (|objMode| u)))

```

```
(when (atom type) (setq type (list type)))
(|sayMSG|
 (|concat|
  (cons "    Type of value of "
   (cons (|fixObjectForPrinting| (pname |$op|))
    (cons ": " type))))))
nil)))))
```

18.2.28 defun getAndSay

```
[getI p??]
[sayMSG p333]
```

— defun getAndSay —

```
(defun |getAndSay| (v prop)
  (let (val)
    (if (setq val (|getI| v prop))
      (|sayMSG| (cons '| | (cons val (cons '|%l| nil)))))
      (|sayMSG| (cons '| | none| (cons '|%l| nil))))))
```

18.2.29 defun displayProperties

```
[getInterpMacroNames p??]
[getParserMacroNames p433]
[remdup p??]
[pairp p??]
[qcdr p??]
[qcar p??]
[msort p??]
[getWorkspaceNames p435]
[sayKeyedMsg p331]
[interpFunctionDepAlists p445]
[isInternalMapName p??]
[getIProplist p??]
[getAndSay p441]
[displayValue p439]
[getI p??]
[displayCondition p445]
[displayMode p446]
```

```

[displayType p440]
[fixObjectForPrinting p436]
[sayMSG p333]
[bright p??]
[prefix2String p??]
[member p1004]
[displayParserMacro p444]
[isInterpMacro p??]
[displayMacro p433]
[displayProperties,sayFunctionDeps p436]
[displayModemap p446]
[exit p??]
[seq p??]
[terminateSystemCommand p432]
[$dependentAlist p??]
[$dependeeAlist p??]
[$frameMessages p718]
[$interpreterFrameName p??]

```

— defun displayProperties —

```

(defun |displayProperties| (option al)
  (let (|$dependentAlist| |$dependeeAlist| tmp1 opt imacs pmacs macros vl pl
        tmp2 vone prop val propsSeen)
    (declare (special |$dependentAlist| |$dependeeAlist| |$frameMessages|
                      |$interpreterFrameName|))
    (setq |$dependentAlist| nil)
    (setq |$dependeeAlist| nil)
    (setq tmp1 (or al (cons '|properties| nil)))
    (setq opt (car tmp1))
    (setq vl (cdr tmp1))
    (setq imacs (|getInterpMacroNames|))
    (setq pmacs (|getParserMacroNames|))
    (setq macros (remdup (append imacs pmacs)))
    (when (or
            (and (pairp vl) (eq (qcdr vl) nil) (eq (qcar vl) '|all|))
            (null vl))
      (setq vl (msort (append (|getWorkspaceNames|) macros))))
    (when |$frameMessages|
      (|sayKeyedMsg| 'S2IZ0065 (cons |$interpreterFrameName| nil)))
    (cond
      ((null vl)
       (if (null |$frameMessages|
           (|sayKeyedMsg| 'S2IZ0066 nil))
           (|sayKeyedMsg| 'S2IZ0067 (cons |$interpreterFrameName| nil)))
       (t
        (|interpFunctionDepAlists|)
        (do ((G166440 vl (cdr G166440)) (v nil))

```

```

((or (atom G166440) (progn (setq v (car G166440)) nil)) nil)
(seq (exit
      (cond
        ((|isInternalMapName| v) '|iterate|)
        (t
         (setq pl (|getIProplist| v))
         (cond
           ((eq option '|flags|)
            (|getAndSay| v '|flags|))
           ((eq option '|value|)
            (|displayValue| v (|getI| v '|value|) nil))
           ((eq option '|condition|)
            (|displayCondition| v (|getI| v '|condition|) nil))
           ((eq option '|model|)
            (|displayModel| v (|getI| v '|model|) nil))
           ((eq option '|type|)
            (|displayType| v (|getI| v '|value|) nil))
           ((eq option '|properties|)
            (cond
              ((eq v '|--flags--|)
               nil)
              ((and (pairp pl)
                     (progn
                      (setq tmp2 (qcar pl))
                      (and (pairp tmp2) (eq (qcar tmp2) '|cacheInfo|))))
               nil)
              (t
               (setq vone (|fixObjectForPrinting| v))
               (|sayMSG|
                (cons "Properties of"
                      (append (|bright| (|prefix2String| vone)) (cons ":" nil))))))
            (cond
              ((null pl)
               (cond
                 ((|member| v pmacs)
                  (|sayMSG| " This is a user-defined macro.")
                  (|displayParserMacro| v))
                 ((|isInterpMacro| v)
                  (|sayMSG| " This is a system-defined macro.")
                  (|displayMacro| v))
                 (t
                  (|sayMSG| " none")))))
              (t
               (setq propsSeen nil)
               (do ((G166451 pl (cdr G166451)) (G166425 nil))
                   ((or (atom G166451)
                        (progn (setq G166425 (car G166451)) nil)
                        (progn
                         (progn
                          (setq prop (car G166425))

```

```

        (setq val (cdr G166425))
        G166425)
    nil))
  nil)
(seq (exit
(cond
  ((and (null (member prop propsSeen)) val)
    (cond
      ((|member| prop
        '(|alias| |generatedCode| IS-GENSYM
          |mapBody| |localVars|))
        nil)
      ((eq prop '|condition|)
        (|displayCondition| prop val t))
      ((eq prop '|recursive|)
        (|sayMSG| "    This is recursive."))
      ((eq prop '|isInterpreterFunction|)
        (|sayMSG| "    This is an interpreter function.")
        (|displayProperties,sayFunctionDeps| v))
      ((eq prop '|isInterpreterRule|)
        (|sayMSG| "    This is an interpreter rule.")
        (|displayProperties,sayFunctionDeps| v))
      ((eq prop '|localModemap|)
        (|displayModemap| v val t))
      ((eq prop '|model|)
        (|displayModel| prop val t))
      (t
        (when (eq prop '|value|)
          (exit
            (when val
              (exit (|displayValue| v val t))))))
        (|sayMSG| (list "    " prop ": " val))
        (setq propsSeen (cons prop propsSeen))))))))))
(t
  (|sayKeyedMsg| 'S2IZ0068 (cons option nil))))))
(|terminateSystemCommand|))))

```

18.2.30 defun displayParserMacro

```

[pfPrintSrcLines p??]
[$pfMacros p100]

```

— defun displayParserMacro —

```

(defun |displayParserMacro| (m)

```

```
(let ((m (assq m |$pfMacros|)))
(declare (special |$pfMacros|))
  (when m (|pfPrintSrcLines| (caddr m)))))
```

18.2.31 defun displayCondition

```
[bright p??]
[sayBrightly p??]
[concat p1003]
[pred2English p??]
```

— defun displayCondition —

```
(defun |displayCondition| (v condition giveVariableIfNil)
  (let (varPart condPart)
    (when giveVariableIfNil (setq varPart (cons ' | of| (|bright| v))))
    (setq condPart (or condition '|true|))
    (|sayBrightly|
      (|concat| '| condition| varPart '|: | (|pred2English| condPart)))))
```

18.2.32 defun interpFunctionDepAlists

```
[putalist p??]
[getalist p??]
[getFlag p??]
[$e p??]
[$dependeeAlist p??]
[$dependentAlist p??]
[$InteractiveFrame p??]
```

— defun interpFunctionDepAlists —

```
(defun |interpFunctionDepAlists| ()
  (let (|$e|)
    (declare (special |$e| |$dependeeAlist| |$dependentAlist|
                      |$InteractiveFrame|))
    (setq |$e| |$InteractiveFrame|)
    (setq |$dependentAlist| (cons (cons nil nil) nil))
    (setq |$dependeeAlist| (cons (cons nil nil) nil))
    (mapcar #'(lambda (dep)
```

```

(let (dependee dependent)
  (setq dependee (first dep))
  (setq dependent (second dep))
  (setq |$dependentAlist|
    (putalist |$dependentAlist| dependee
      (cons dependent (getalist |$dependentAlist| dependee))))
  (spadlet |$dependeeAlist|
    (putalist |$dependeeAlist| dependent
      (cons dependee (getalist |$dependeeAlist| dependent))))))
(|getFlag| '$dependencies|)))

```

18.2.33 defun displayModemap

```

[bright p??]
[sayBrightly p??]
[concat p1003]
[formatSignature p??]

```

— defun displayModemap —

```

(defun |displayModemap| (v val giveVariableIfNil)
  (labels (
    (g (v mm giveVariableIfNil)
      (let (local signature fn varPart prefix)
        (setq local (caar mm))
        (setq signature (cdar mm))
        (setq fn (cadr mm))
        (unless (eq local '|interpOnly|)
          (spadlet varPart (unless giveVariableIfNil (cons " of" (|bright| v))))
          (spadlet prefix
            (cons '| Compiled function type| (append varPart (cons '|: | nil)))))
          (|sayBrightly| (|concat| prefix (|formatSignature| signature))))))
    (mapcar #'(lambda (x) (g v x giveVariableIfNil)) val)))

```

18.2.34 defun displayMode

```

[bright p??]
[fixObjectForPrinting p436]
[sayBrightly p??]
[concat p1003]

```

[prefix2String p??]

— defun displayMode —

```
(defun |displayMode| (v mode giveVariableIfNil)
  (let (varPart)
    (when mode
      (unless giveVariableIfNil
        (setq varPart (cons '| of| (|bright| (|fixObjectForPrinting| v))))))
    (|sayBrightly|
      (|concat| '| Declared type or mode| varPart '|: |
        (|prefix2String| mode))))))
```

—————

18.2.35 defun Split into tokens delimited by spaces

[stripSpaces p451]

— defun dumbTokenize —

```
(defun |dumbTokenize| (str)
  (let (inString token (tokenStart 0) previousSpace tokenList)
    (dotimes (i (1- (|#| str)))
      (cond
        ((char= (elt str i) #"") ; don't split strings
          (setq inString (null inString))
          (setq previousSpace nil))
        ((and (char= (elt str i) #"space") (null inString))
          (unless previousSpace
            (setq token (|stripSpaces| (subseq str tokenStart i)))
            (setq tokenList (cons token tokenList))
            (setq tokenStart (1+ i))
            (setq previousSpace t)))
          (t
            (setq previousSpace nil))))
    (setq tokenList (cons (|stripSpaces| (subseq str tokenStart)) tokenList))
    (nreverse tokenList)))
```

—————

18.2.36 defun Convert string tokens to their proper type

[isIntegerString p448]

— defun tokTran —


```
(defun |tokTran| (tok)
  (let (tmp)
    (if (stringp tok)
      (cond
        ((eql (|#| tok) 0) nil)
        ((setq tmp (|isIntegerString| tok)) tmp)
        ((char= (elt tok 0) #" " ) (subseq tok 1 (1- (|#| tok))))
        (t (intern tok)))
      tok)))
```

18.2.37 defun Is the argument string an integer?

— defun isIntegerString 0 —

```
(defun |isIntegerString| (tok)
  (multiple-value-bind (int len) (parse-integer tok :junk-allowed t)
    (when (and int (= len (length tok))) int)))
```

18.2.38 defun Handle parsed system commands

[dumbTokenize p447]
 [parseSystemCmd p449]
 [tokTran p447]
 [systemCommand p428]

— defun handleParsedSystemCommands —

```
(defun |handleParsedSystemCommands| (unabr optionList)
  (declare (ignore unabr))
  (let (restOptionList parcmd trail)
    (setq restOptionList (mapcar #'|dumbTokenize| (cdr optionList)))
    (setq parcmd (|parseSystemCmd| (car optionList)))
    (setq trail
      (mapcar #'(lambda (opt)
                    (mapcar #'(lambda (tok) (|tokTran| tok)) opt)) restOptionList))
    (|systemCommand| (cons parcmd trail))))
```

18.2.39 defun Parse a system command

[tokTran p447]
 [stripSpaces p451]
 [parseFromString p48]
 [dumbTokenize p447]

— defun parseSystemCmd —

```
(defun |parseSystemCmd| (opt)
  (let (spaceIndex)
    (if (setq spaceIndex (search " " opt))
      (list
        (|tokTran| (|stripSpaces| (subseq opt 0 spaceIndex)))
        (|parseFromString| (|stripSpaces| (subseq opt spaceIndex))))
      (mapcar #'|tokTran| (|dumbTokenize| opt)))))
```

18.2.40 defun Get first word in a string

[subseq p??]
 [stringSpaces p??]

— defun getFirstWord —

```
(defun |getFirstWord| (string)
  (let (spaceIndex)
    (setq spaceIndex (search " " string))
    (if spaceIndex
      (|stripSpaces| (subseq string 0 spaceIndex))
      string)))
```

18.2.41 defun Unabbreviate keywords in commands

[selectOptionLC p459]
 [selectOption p459]
 [commandsForUserLevel p428]
 [\$systemCommands p423]
 [\$currentLine p??]
 [\$syscommands p424]
 [line p??]

— **defun unAbbreviateKeyword** —

```
(defun |unAbbreviateKeyword| (x)
  (let (xp)
    (declare (special |$systemCommands| |$currentLine| $syscommands line))
    (setq xp (|selectOptionLC| x $syscommands '|commandErrorIfAmbiguous|))
    (cond
      ((null xp)
        (setq xp '|system|)
        (setq line (concat ")system " (substring line 1 (1- (|#| line)))))
      (spadlet |$currentLine| line)))
    (|selectOption| xp (|commandsForUserLevel| |$systemCommands|)
      '|commandUserLevelError|)))
```

18.2.42 **defun** The command is ambiguous error

```
[commandAmbiguityError p432]
[$oldline p429]
[line p??]
```

— **defun commandErrorIfAmbiguous** —

```
(defun |commandErrorIfAmbiguous| (x u)
  (declare (special $oldline line))
  (when u
    (setq $oldline line)
    (|commandAmbiguityError| '|command| x u)))
```

```
[stripSpaces p451]
[nplisp p452]
[stripLisp p451]
[sayKeyedMsg p331]
[npboot p452]
[npsystem p452]
[npsynonym p453]
[member p1004]
[concat p1003]
```

— **defun handleNoParseCommands** —

```
(defun |handleNoParseCommands| (unab string)
```

```

(let (spaceindex funname)
  (setq string (|stripSpaces| string))
  (setq spaceindex (search " " string))
  (cond
    ((eq unab '|lisp|)
     (if spaceindex
      (|nplisp| (|stripLisp| string))
      (|sayKeyedMsg| 's2iv0005 nil)))
    ((eq unab '|boot|)
     (if spaceindex
      (|npboot| (subseq string (1+ spaceindex)))
      (|sayKeyedMsg| 's2iv0005 nil)))
    ((eq unab '|system|)
     (if spaceindex
      (|npssystem| unab string)
      (|sayKeyedMsg| 's2iv0005 nil)))
    ((eq unab '|synonym|)
     (if spaceindex
      (|npsynonym| unab (subseq string (1+ spaceindex)))
      (|npsynonym| unab "")))
    ((null spaceindex)
     (funcall unab))
    ((|member| unab '(|quit| |fin| |pquit| |credits| |copyright| |trademark|))
     (|sayKeyedMsg| 's2iv0005 nil))
    (t
     (setq funname (intern (concat "np" (string unab))))
     (funcall funname (subseq string (1+ spaceindex))))))

```

18.2.43 defun Remove the spaces surrounding a string

TPDHERE: This should probably be a macro or eliminated

— defun stripSpaces 0 —

```

(defun |stripSpaces| (str)
  (string-trim '(\space) str))

```

18.2.44 defun Remove the lisp command prefix

— defun stripLisp 0 —

```

(defun |stripLisp| (str)

```

```
(if (string= (subseq str 0 4) "lisp")
    (subseq str 4)
    str))
```

18.2.45 defun Handle the)lisp command

[\$ans p??]

— defun nplisp 0 —

```
(defun |nplisp| (str)
  (declare (special |$ans|))
  (setq |$ans| (eval (read-from-string str)))
  (format t "~&Value = ~S~%" |$ans|))
```

18.2.46 defun The)boot command is no longer supported

TPDHERE: Remove all boot references from top level

— defun npboot 0 —

```
(defun |npboot| (str)
  (declare (ignore str))
  (format t "The )boot command is no longer supported~%"))
```

18.2.47 defun Handle the)system command

Note that unAbbreviateKeyword returns the word “system” for unknown words so we have to search for this case. This complication may never arrive in practice. [sayKeyedMsg p331]

— defun npsystem —

```
(defun |npsystem| (unab str)
  (let (spaceIndex sysPart)
    (setq spaceIndex (search " " str))
    (cond
      ((null spaceIndex) (|sayKeyedMsg| 'S2IZ0080 (list str)))
      (t
```

```
(setq sysPart (subseq str 0 spaceIndex))
(if (search sysPart (string unab))
    (obey (subseq str (1+ spaceIndex)))
    (|sayKeyedMsg| 'S2IZ0080 (list sysPart))))))
```

18.2.48 defun Handle the)synonym command

[npProcessSynonym p453]

— defun npsynonym —

```
(defun |npsynonym| (unab str)
  (declare (ignore unab))
  (|npProcessSynonym| str))
```

18.2.49 defun Handle the synonym system command

[printSynonyms p454]
 [processSynonymLine p811]
 [putalist p??]
 [terminateSystemCommand p432]
 [\$CommandSynonymAlist p458]

— defun npProcessSynonym —

```
(defun |npProcessSynonym| (str)
  (let (pair)
    (declare (special |$CommandSynonymAlist|))
    (if (= (length str) 0)
        (|printSynonyms| nil)
        (progn
          (setq pair (|processSynonymLine| str))
          (if |$CommandSynonymAlist|
              (putalist |$CommandSynonymAlist| (car pair) (cdr pair)))
              (setq |$CommandSynonymAlist| (cons pair nil))))
    (|terminateSystemCommand|)))
```

18.2.50 defun printSynonyms

[centerAndHighlight p??]
 [specialChar p936]
 [filterListOfStringsWithFn p907]
 [synonymsForUserLevel p809]
 [printLabelledList p454]
 [\$CommandSynonymAlist p458]
 [\$linelength p751]

— defun printSynonyms —

```
(defun |printSynonyms| (patterns)
  (prog (ls t1)
    (declare (special |$CommandSynonymAlist| $linelength)
      (|centerAndHighlight| ' |System Command Synonyms|
        $linelength (|specialChar| ' |hbar|)))
    (setq ls
      (|filterListOfStringsWithFn| patterns
        (do ((t2 (|synonymsForUserLevel| |$CommandSynonymAlist|) (cdr t2)))
          ((atom t2) (nreverse0 t1))
          (push (cons (princ-to-string (caar t2)) (cdar t2)) t1))
        (|function| car)))
      (|printLabelledList| ls "user" "synonyms" ") " patterns)))
```

18.2.51 defun Print a list of each matching synonym

The prefix goes before each element on each side of the list, eg, ") " [sayMessage p??]

[blankList p??]
 [substring p??]
 [entryWidth p??]
 [sayBrightly p??]
 [concat p1003]
 [fillerSpaces p20]

— defun printLabelledList —

```
(defun |printLabelledList| (ls label1 label2 prefix patterns)
  (let (comm syn wid)
    (if (null ls)
      (if (null patterns)
        (|sayMessage| (list " No " label1 "-defined " label2 " in effect."))
        (|sayMessage|
          ' (" No " ,label1 "-defined " ,label2 " satisfying patterns:"
```

```

    |%l| "      " |%b| ,@(append (|blankList| patterns) (list '|%d|'))))
(progn
  (when patterns
    (|sayMessage|
      '(',label1 "-defined " ,label2 " satisfying patterns:" |%l| "      "
        |%b| ,@(append (|blankList| patterns) (list '|%d|'))))
    (do ((t1 ls (cdr t1)))
      ((atom t1) nil)
      (setq syn (caar t1))
      (setq comm (cdar t1))
      (when (string= (substring syn 0 1) "|")
        (setq syn (substring syn 1 nil)))
      (when (string= syn "%i") (setq syn "%i "))
      (setq wid (max (- 30 (|entryWidth| syn)) 1))
      (|sayBrightly|
        (|concat| '|%b| prefix syn '|%d| (|fillerSpaces| wid ".")
          " " prefix comm)))
      (|sayBrightly| ""))))

```

18.2.52 defvar \$tokenCommands

This is a list of the commands that expect the interpreter to parse their arguments. Thus the history command expects that Axiom will have tokenized and validated the input before calling the history function.

— **initvars** —

```
(defvar |$tokenCommands| nil)
```

— **postvars** —

```

(eval-when (eval load)
  (setq |$tokenCommands|
    '( |abbreviations|
      |cd|
      |clear|
      |close|
      |compiler|
      |depends|
      |display|
      |describe|
      |edit|

```



```

|frame|
|frame|
|help|
|history|
|input|
|library|
|load|
|ltrace|
|read|
|savesystem|
|set|
|spool|
|undo|
|what|
|with|
|workfiles|
|zsystemdevelopment|
)))

```

18.2.53 defvar \$InitialCommandSynonymAlist

Axiom can create “synonyms” for commands. We create an initial table of synonyms which are in common use.

— initvars —

```
(defvar |$InitialCommandSynonymAlist| nil)
```

18.2.54 defun Print the current version information

```

[*yearweek* p??]
[*build-version* p??]

```

— defun axiomVersion 0 —

```

(defun axiomVersion ()
  (declare (special *build-version* *yearweek*))
  (concatenate 'string "Axiom " *build-version* " built on " *yearweek*))

```

— postvars —

```
(eval-when (eval load)
  (setq |$InitialCommandSynonymAlist|
    '(
      (|?|          . "what commands")
      (|ap|         . "what things")
      (|apr|        . "what things")
      (|apropos|    . "what things")
      (|cache|      . "set functions cache")
      (|cl|         . "clear")
      (|cls|        . "zsystemdevelopment )cls")
      (|cms|        . "system")
      (|col|        . "compiler")
      (|d|          . "display")
      (|depl|       . "display dependents")
      (|dependents| . "display dependents")
      (|e|          . "edit")
      (|expose|     . "set expose add constructor")
      (|fc|         . "zsystemdevelopment )c")
      (|fd|         . "zsystemdevelopment )d")
      (|fdt|        . "zsystemdevelopment )dt")
      (|fct|        . "zsystemdevelopment )ct")
      (|fctl|       . "zsystemdevelopment )ctl")
      (|fel|        . "zsystemdevelopment )e")
      (|fec|        . "zsystemdevelopment )ec")
      (|fect|       . "zsystemdevelopment )ect")
      (|fns|        . "exec spadfn")
      (|fortran|    . "set output fortran")
      (|h|          . "help")
      (|hd|         . "system hypertex &")
      (|kclam|      . "boot clearClams ( )")
      (|killcaches| . "boot clearConstructorAndLispLibCaches ( )")
      (|patch|      . "zsystemdevelopment )patch")
      (|pause|      . "zsystemdevelopment )pause")
      (|prompt|     . "set message prompt")
      (|recurrence| . "set functions recurrence")
      (|restore|    . "history )restore")
      (|save|       . "history )save")
      (|startGraphics| . "system $AXIOM/lib/viewman &")
      (|startNAGLink| . "system $AXIOM/lib/nagman &")
      (|stopGraphics| . "lisp (|sockSendSignal| 2 15)")
      (|stopNAGLink| . "lisp (|sockSendSignal| 8 15)")
      (|time|       . "set message time")
      (|type|       . "set message type")
      (|unexpose|   . "set expose drop constructor")
      (|up|         . "zsystemdevelopment )update")
      (|version|    . "lisp (axiomVersion)")
      (|w|          . "what")
    )
  )
```

```

      (|wc|      . "what categories")
      (|wd|      . "what domains")
      (|who|     . "lisp (pprint credits)")
      (|wp|      . "what packages")
      (|ws|      . "what synonyms")
    )))

```

18.2.55 `defvar $CommandSynonymAlist`

The actual list of synonyms is initialized to be the same as the above initial list of synonyms. The user synonyms that are added during a session are pushed onto this list for later lookup.

— **initvars** —

```
(defvar |$CommandSynonymAlist| nil)
```

— **postvars** —

```

(eval-when (eval load)
  (setq |$CommandSynonymAlist| (copy-alist |$InitialCommandSynonymAlist|)))

```

18.2.56 `defun ncloopCommand`

The `$systemCommandFunction` is set in `SpadInterpretStream` to point to the function `InterpExecuteSpadSystemCommand`. The system commands are handled by the function kept in the “hook” variable `$systemCommandFunction` which has the default function `InterpExecuteSpadSystemCommand`. Thus, when a system command is entered this function is called.

The only exception is the `)include` function which inserts the contents of a file inline in the input stream. This is useful for processing `)read` of input files. [ncloopPrefix? p459]

[ncloopInclude1 p599]

[`$systemCommandFunction` p??]

[`$systemCommandFunction` p??]

— **defun ncloopCommand** —

```

(defun |ncloopCommand| (line n)
  (let (a)

```

```
(declare (special |$systemCommandFunction|))
(if (setq a (|ncloopPrefix?| ")include" line))
    (|ncloopInclude1| a n)
    (progn
      (funcall |$systemCommandFunction| line)
      n))))
```

18.2.57 defun ncloopPrefix?

If we find the prefix string in the whole string starting at position zero we return the remainder of the string without the leading prefix.

— **defun ncloopPrefix? 0** —

```
(defun |ncloopPrefix?| (prefix whole)
  (when (eql (search prefix whole) 0)
    (subseq whole (length prefix))))
```

18.2.58 defun selectOptionLC

```
[selectOption p459]
[downcase p??]
[object2Identifier p??]
```

— **defun selectOptionLC** —

```
(defun |selectOptionLC| (x 1 errorFunction)
  (|selectOption| (downcase (|object2Identifier| x)) 1 errorFunction))
```

18.2.59 defun selectOption

```
[member p1004]
[identp p1003]
[stringPrefix? p??]
[pname p1001]
[pairp p??]
[qcdr p??]
```

[qcar p??]

— defun selectOption —

```
(defun |selectOption| (x l errorfunction)
  (let (u y)
    (cond
      ((|member| x l) x)
      ((null (identp x))
       (cond
         (errorfunction (funcall errorfunction x u))
         (t nil)))
      (t
       (setq u
              (let (t0)
                (do ((t1 l (cdr t1)) (y nil))
                    ((or (atom t1) (progn (setq y (car t1)) nil)) (nreverse0 t0))
                  (if (|stringPrefix?| (pname x) (pname y))
                      (setq t0 (cons y t0)))))))
       (cond
         ((and (pairp u) (eq (qcdr u) nil) (progn (setq y (qcar u)) t)) y)
         (errorfunction (funcall errorfunction x u))
         (t nil))))))
```

Chapter 19

)abbreviations help page Command

19.1 abbreviations help page man page

— abbreviations.help —

```
=====
A.2. )abbreviation
=====
```

User Level Required: compiler

Command Syntax:

-)abbreviation query [nameOrAbbrev]
-)abbreviation category abbrev fullname [quiet]
-)abbreviation domain abbrev fullname [quiet]
-)abbreviation package abbrev fullname [quiet]
-)abbreviation remove nameOrAbbrev

Command Description:

This command is used to query, set and remove abbreviations for category, domain and package constructors. Every constructor must have a unique abbreviation. This abbreviation is part of the name of the subdirectory under which the components of the compiled constructor are stored. Furthermore, by issuing this command you let the system know what file to load automatically if you use a new constructor. Abbreviations must start with a letter and then be followed by up to seven letters or digits. Any letters appearing in the abbreviation must be in uppercase.

When used with the query argument, this command may be used to list the name associated with a particular abbreviation or the abbreviation for a constructor. If no abbreviation or name is given, the names and corresponding abbreviations for all constructors are listed.

The following shows the abbreviation for the constructor List:

```
)abbreviation query List
```

The following shows the constructor name corresponding to the abbreviation NNI:

```
)abbreviation query NNI
```

The following lists all constructor names and their abbreviations.

```
)abbreviation query
```

To add an abbreviation for a constructor, use this command with category, domain or package. The following add abbreviations to the system for a category, domain and package, respectively:

```
)abbreviation domain SET Set
)abbreviation category COMPCAT ComplexCategory
)abbreviation package LIST2MAP ListToMap
```

If the)quiet option is used, no output is displayed from this command. You would normally only define an abbreviation in a library source file. If this command is issued for a constructor that has already been loaded, the constructor will be reloaded next time it is referenced. In particular, you can use this command to force the automatic reloading of constructors.

To remove an abbreviation, the remove argument is used. This is usually only used to correct a previous command that set an abbreviation for a constructor name. If, in fact, the abbreviation does exist, you are prompted for confirmation of the removal request. Either of the following commands will remove the abbreviation VECTOR2 and the constructor name VectorFunctions2 from the system:

```
)abbreviation remove VECTOR2
)abbreviation remove VectorFunctions2
```

Also See:

- o)compile

19.2 Functions

19.2.1 defun abbreviations

[abbreviationsSpad2Cmd p463]

— defun abbreviations —

```
(defun |abbreviations| (l)
  (|abbreviationsSpad2Cmd| l))
```

—————

19.2.2 defun abbreviationsSpad2Cmd

[listConstructorAbbreviations p464]
 [abbreviation? p??]
 [abbQuery p514]
 [deldatabase p967]
 [size p1001]
 [sayKeyedMsg p331]
 [mkUserConstructorAbbreviation p??]
 [setdatabase p966]
 [seq p??]
 [exit p??]
 [opOf p??]
 [helpSpad2Cmd p550]
 [selectOptionLC p459]
 [pairp p??]
 [qcar p??]
 [qcdr p??]
 [\$options p??]

— defun abbreviationsSpad2Cmd —

```
(defun |abbreviationsSpad2Cmd| (arg)
  (let (abopts quiet opt key type constructor t2 a b al)
    (declare (special |$options|))
    (if (null arg)
      (|helpSpad2Cmd| '(|abbreviations|))
      (progn
        (setq abopts '(|query| |domain| |category| |package| |remove|))
        (setq quiet nil)
        (do ((t0 |$options| (cdr t0)) (t1 nil))
            ((or (atom t0)
```



```

      (progn (setq t1 (car t0)) nil)
      (progn (progn (setq opt (car t1)) t1) nil))
    nil)
  (setq opt (|selectOptionLC| opt '(|quiet|) '|optionError|))
  (when (eq opt '|quiet|) (setq quiet t)))
(when
  (and (pairp arg)
    (progn
      (setq opt (qcar arg))
      (setq al (qcdr arg))
      t))
  (setq key (|opOf| (car al)))
  (setq type (|selectOptionLC| opt abopts '|optionError|))
  (cond
    ((eq type '|query|)
      (cond
        ((null al) (|listConstructorAbbreviations|))
        ((setq constructor (|abbreviation?| key))
          (|abbQuery| constructor))
        (t (|abbQuery| key))))
    ((eq type '|remove|)
      (deldatabase key 'abbreviation))
    ((oddp (size al))
      (|sayKeyedMsg| 's2iz0002 (list type)))
    (t
      (do () (nil nil)
        (seq
          (exit
            (cond
              ((null al) (return '|fromLoop|))
              (t
                (setq t2 al)
                (setq a (car t2))
                (setq b (cadr t2))
                (setq al (cddr t2))
                (|mkUserConstructorAbbreviation| b a type)
                (setdatabase b 'abbreviation a)
                (setdatabase b 'constructorkind type))))))
      (unless quiet
        (|sayKeyedMsg| 's2iz0001 (list a type (|opOf| b))))))))))

```

19.2.3 defun listConstructorAbbreviations

```

[upcase p??]
[queryUserKeyedMsg p??]
[string2id-n p??]

```

[whatSpad2Cmd p904]
 [sayKeyedMsg p331]

— **defun listConstructorAbbreviations** —

```
(defun |listConstructorAbbreviations| ()
  (let (x)
    (setq x (upcase (|queryUserKeyedMsg| 's2iz0056 nil)))
    (if (member (string2id-n x 1) '(Y YES))
      (progn
        (|whatSpad2Cmd| '(|categories|))
        (|whatSpad2Cmd| '(|domains|))
        (|whatSpad2Cmd| '(|packages|)))
      (|sayKeyedMsg| 's2iz0057 nil))))
```

Chapter 20

)boot help page Command

20.1 boot help page man page

— boot.help —

```
=====
A.3. )boot
=====
```

User Level Required: development

Command Syntax:

-)boot bootExpression

Command Description:

This command is used by AXIOM system developers to execute expressions written in the BOOT language. For example,

```
)boot times3(x) == 3*x
```

creates and compiles the Lisp function ‘‘times3’’ obtained by translating the BOOT code.

Also See:

- o)fin
- o)lisp
- o)set
- o)system

¹

20.2 Functions

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

¹ “fin” (31.1.1 p 526) “lisp” (?? p ??) “set” (44.37.1 p 785) “system” (?? p ??)

Chapter 21

)browse help page Command

21.1 browse help page man page

— browse.help —

User Level Required: development

Command Syntax:

```
)browse
```

Command Description:

This command is used by Axiom system users to start the Axiom top level loop listening for browser connections.

21.2 Overview

The Axiom book on the help browser is a complete rewrite of the hyperdoc mechanism. There are several components that were needed to make this function. Most of the web browser components are described in bookvol11.pamphlet. This portion describes some of the design issues needed to support the interface.

The axServer command takes a port (defaulting to 8085) and a program to handle the browser interaction (defaulting to multiServ). The axServer function opens the port, constructs the

stream, and passes the stream to multiServ. The multiServ loop processes one interaction at a time.

So the basic process is that the Axiom “)browse” command opens a socket and listens for http requests. Based on the type of request (either ‘GET’ or ‘POST’) and the content of the request, which is one of:

- command - algebra request/response
- lispcall - a lisp s-expression to be evaluated
- showcall - an Axiom)show command

the multiServ function will call a handler function to evaluate the command line and construct a response. GET requests result in a new browser page. POST requests result in an inline result.

Most responses contain the fields:

- stepnum - this is the Axiom step number
- command - this is the original command from the browser
- algebra - this is the Axiom 2D algebra output
- mathml - this is the MathML version of the Axiom algebra
- type - this is the type of the Axiom result

21.3 Browsers, MathML, and Fonts

This work has the Firefox browser as its target. Firefox has built-in support for MathML, javascript, and XMLHttpRequest handling. More details are available in bookvol11.pamphlet but the very basic machinery for communication with the browser involves a dance between the browser and the multiServ function (see the axserver.spad.pamphlet).

In particular, a simple request is embedded in a web page as:

```
<ul>
  <li>
    <input type="submit" id="p3" class="subbut"
      onclick="makeRequest('p3');"
      value="sin(x)" />
    <div id="ansp3"><div></div></div>
  </li>
</ul>
```

which says that this is an html “input” field of type “submit”. The CSS display class is “subbut” which is of a different color than the surrounding text to make it obvious that you can click on this field. Clickable fields that have no response text are of class “noresult”.

The javascript call to “makeRequest” gives the “id” of this input field, which must be unique in the page, as an argument. In this case, the argument is ‘p3’. The “value” field holds the display text which will be passed back to Axiom as a command.

When the result arrives the “showanswer” function will select out the mathml field of the response, construct the “id” of the html div to hold the response by concatenating the string “ans” (answer) to the “id” of the request resulting, in this case, as “ansp3”. The “showanswer” function will find this div and replace it with a div containing the mathml result.

The “makeRequest” function is:

```
function makeRequest(arg) {
  http_request = new XMLHttpRequest();
  var command = cmdline(arg);
  //alert(command);
  http_request.open('POST', '127.0.0.1:8085', true);
  http_request.onreadystatechange = handleResponse;
  http_request.setRequestHeader('Content-Type', 'text/plain');
  http_request.send("command="+command);
  return(false);
}
```

It contains a request to open a local server connection to Axiom, sets “handleResponse” as the function to call on reply, sets up the type of request, fills in the command field, and sends off the http request.

When a response is received, the “handleResponse” function checks for the correct reply state, strips out the important text, and calls “showanswer”.

```
function handleResponse() {
  if (http_request.readyState == 4) {
    if (http_request.status == 200) {
      showanswer(http_request.responseText, 'mathAns');
    } else
    {
      alert('There was a problem with the request.' + http_request.statusText);
    }
  }
}
```

See bookvol11.pamphlet for further details.

21.4 The axServer/multiServ loop

The basic call to start an Axiom browser listener is:

```
)set message autoload off
)set output mathml on
axServer(8085,multiServ)$AXSERV
```


This call sets the port, opens a socket, attaches it to a stream, and then calls “multiServ” with that stream. The “multiServ” function loops serving web responses to that port.

21.5 The)browse command

In order to make the whole process cleaner the function “)browse” handles the details. This code creates the command-line function for)browse

The browse function does the internal equivalent of the following 3 command line statments:

```
)set message autoload off
)set output mathml on
axServer(8085,multiServ)$AXSERV
```

which causes Axiom to start serving web pages on port 8085

For those unfamiliar with calling algebra from lisp there are a few points to mention.

The loadLib needs to be called to load the algebra code into the image. Normally this is automatic but we are not using the interpreter so we need to do this “by hand”.

Each algebra file contains a ”constructor function” which builds the domain, which is a vector, and then caches the vector so that every call to the contructor returns an EQ vector, that is, the same vector. In this case, we call the constructor |AxiomServer|

The axServer function was mangled internally to |AXSERV;axServer;IMV;2|. The multiServ function was mangled to |AXSERV;multiServ;SeV;3| Note well that if you change axserver.spad these names might change which will generate the error message along the lines of:

```
System error:
The function $\vert$AXSERV;axServer;IMV;2$\vert$ is undefined.
```

To fix this you need to look at int/algebra/AXSERV.nrlib/code.lsp and find the new mangled function name. A better solution would be to dynamically look up the surface names in the domain vector.

Each Axiom function expects the domain vector as the last argument. This is not obvious from the call as the interpreter supplies it. We must do that “by hand”.

We don’t call the multiServ function. We pass it as a parameter to the axServer function. When it does get called by the SPADCALL macro it needs to be a lisp pair whose car is the function and whose cdr is the domain vector. We construct that pair here as the second argument to axServer. The third, hidden, argument to axServer is the domain vector which we supply “by hand”.

The socket can be supplied on the command line but defaults to 8085. Axiom supplies the arguments as a list.

21.6 Variables Used

21.7 Functions

```
[set p785]
[loadLib p??]
[AxiomServer p??]
[AXSERV;axServer;IMV;2 p??]
```

— defun browse —

```
(defun |browse| (socket)
  (let (axserv browser)
    (if socket
      (setq socket (car socket))
      (setq socket 8085))
    (|set| '(|mes| |auto| |off|))
    (|set| '(|out| |mathml| |on|))
    (|loadLib| '|AxiomServer|)
    (setq axsolv (|AxiomServer|))
    (setq browser
      (|AXSERV;axServer;IMV;2| socket
        (cons #'|AXSERV;multiServ;SeV;3| axsolv) axsolv))))
```

Now we have to bolt it into Axiom. This involves two lookups.

We create the lisp pair

```
(|browse| . |development|)
```

and cons it into the \$systemCommands command table. This allows the command to be executed in development mode. This lookup decides if this command is allowed. It also has the side-effect of putting the command into the \$SYSCOMMANDS variable which is used to determine if the token is a command.

21.8 The server support code

Chapter 22

)cd help page Command

22.1 cd help page man page

— cd.help —

```
=====
A.4. )cd
=====
```

User Level Required: interpreter

Command Syntax:

-)cd directory

Command Description:

This command sets the AXIOM working current directory. The current directory is used for looking for input files (for)read), AXIOM library source files (for)compile), saved history environment files (for)history)restore), compiled AXIOM library files (for)library), and files to edit (for)edit). It is also used for writing spool files (via)spool), writing history input files (via)history)write) and history environment files (via)history)save),and compiled AXIOM library files (via)compile).

If issued with no argument, this command sets the AXIOM current directory to your home directory. If an argument is used, it must be a valid directory name. Except for the ‘)’ at the beginning of the command, this has the same syntax as the operating system cd command.

Also See:

o)compile

- o `)edit`
- o `)history`
- o `)library`
- o `)read`
- o `)spool`

1

22.2 Variables Used

22.3 Functions

¹ “`edit`” (30.2.1 p 522) “`history`” (34.4.7 p 560) “`library`” (63.1.34 p 971) “`read`” (42.1.1 p 620) “`spool`” (?? p ??)

Chapter 23

)clear help page Command

23.1 clear help page man page

— clear.help —

```
=====
A.6. )clear
=====
```

User Level Required: interpreter

Command Syntax:

```
- )clear all
- )clear completely
- )clear properties all
- )clear properties obj1 [obj2 ...]
- )clear value      all
- )clear value      obj1 [obj2 ...]
- )clear mode       all
- )clear mode       obj1 [obj2 ...]
```

Command Description:

This command is used to remove function and variable declarations, definitions and values from the workspace. To empty the entire workspace and reset the step counter to 1, issue

```
)clear all
```

To remove everything in the workspace but not reset the step counter, issue

```
)clear properties all
```

To remove everything about the object `x`, issue

```
)clear properties x
```

To remove everything about the objects `x`, `y` and `f`, issue

```
)clear properties x y f
```

The word `properties` may be abbreviated to the single letter `'p'`.

```
)clear p all
```

```
)clear p x
```

```
)clear p x y f
```

All definitions of functions and values of variables may be removed by either

```
)clear value all
```

```
)clear v all
```

This retains whatever declarations the objects had. To remove definitions and values for the specific objects `x`, `y` and `f`, issue

```
)clear value x y f
```

```
)clear v x y f
```

To remove the declarations of everything while leaving the definitions and values, issue

```
)clear mode all
```

```
)clear m all
```

To remove declarations for the specific objects `x`, `y` and `f`, issue

```
)clear mode x y f
```

```
)clear m x y f
```

The `)display names` and `)display properties` commands may be used to see what is currently in the workspace.

The command

```
)clear completely
```

does everything that `)clear all` does, and also clears the internal system function and constructor caches.

Also See:

- o `)display`

- o)history
- o)undo

1

23.2 Variables Used

23.2.1 defvar \$clearOptions

— initvars —

```
(defvar |$clearOptions| '(|modes| |operations| |properties| |types| |values|))
```

23.3 Functions

23.3.1 defun clear

[clearSpad2Cmd p480]

— defun clear —

```
(defun |clear| (1)
  (|clearSpad2Cmd| 1))
```

23.3.2 defvar \$clearExcept

— initvars —

```
(defvar |$clearExcept| nil)
```

¹ “display” (29.2.1 p 513) “history” (34.4.7 p 560) “undo” (51.3.6 p 886)

23.3.3 defun clearSpad2Cmd

TPDHERE: Note that this function also seems to parse out)except)completely and)scaches which don't seem to be documented. [selectOptionLC p459]

```
[sayKeyedMsg p331]
[clearCmdAll p483]
[clearCmdCompletely p482]
[clearCmdSortedCaches p481]
[clearCmdExcept p484]
[clearCmdParts p484]
[updateCurrentInterpreterFrame p537]
[$clearExcept p479]
[$options p??]
[$clearOptions p479]
```

— defun clearSpad2Cmd —

```
(defun |clearSpad2Cmd| (l)
  (let (|$clearExcept| opt optlist arg)
    (declare (special |$clearExcept| |$options| |$clearOptions|))
    (cond
      (|$options|
        (setq |$clearExcept|
          (prog (t0)
            (setq t0 t)
            (return
              (do ((t1 nil (null t0))
                  (t2 |$options| (cdr t2))
                  (t3 nil))
                ((or t1
                     (atom t2)
                     (progn (setq t3 (car t2)) nil)
                     (progn (progn (setq opt (car t3)) t3) nil))
                 t0)
              (setq t0
                (and t0
                  (eq
                    (|selectOptionLC| opt '(|except|) '|optionError|)
                    '|except|))))))))))
      (cond
        ((null l)
          (setq optlist
            (prog (t4)
              (setq t4 nil)
              (return
                (do ((t5 |$clearOptions| (cdr t5)) (x nil))
                  ((or (atom t5) (progn (setq x (car t5)) nil)) t4)
                  (setq t4 (append t4 '(|%1| " " ,x))))))
              (|sayKeyedMsg| 's2iz0010 (list optlist)))
```

```
(t
  (setq arg
    (|selectOptionLC| (car l) '(|all| |completely| |scaches|) nil))
  (cond
    ((eq arg '|all|)      (|clearCmdAll|))
    ((eq arg '|completely|) (|clearCmdCompletely|))
    ((eq arg '|scaches|)   (|clearCmdSortedCaches|))
    (|$clearExcept|       (|clearCmdExcept| l))
    (t
      (|clearCmdParts| l)
      (|updateCurrentInterpreterFrame|))))))
```

23.3.4 defun clearCmdSortedCaches

```
[compiledLookupCheck p??]
[spadcall p??]
[$lookupDefaults p??]
[$Void p??]
[$ConstructorCache p??]
```

— defun clearCmdSortedCaches —

```
(defun |clearCmdSortedCaches| ()
  (let (|$lookupDefaults| domain pair)
    (declare (special |$lookupDefaults| |$Void| |$ConstructorCache|))
    (do ((t0 (hget |$ConstructorCache| '|SortedCache|) (cdr t0))
        (t1 nil))
      ((or (atom t0)
          (progn
            (setq t1 (car t0))
            (setq domain (cddr t1))
            nil))
        nil)
      (setq pair (|compiledLookupCheck| '|clearCache| (list |$Void|) domain))
      (spadcall pair))))
```

23.3.5 defvar \$functionTable

— initvars —

```
(defvar |$functionTable| nil)
```

23.3.6 defun clearCmdCompletely

```
[clearCmdAll p483]
[sayKeyedMsg p331]
[clearClams p??]
[clearConstructorCaches p??]
[reclaim p39]
[$localExposureData p670]
[$xdatabase p??]
[$CatOfCatDatabase p??]
[$DomOfCatDatabase p??]
[$JoinOfCatDatabase p??]
[$JoinOfDomDatabase p??]
[$attributeDb p??]
[$functionTable p481]
[$existingFiles p??]
[$localExposureDataDefault p670]
```

— defun clearCmdCompletely —

```
(defun |clearCmdCompletely| ()
  (declare (special |$localExposureData| |$xdatabase| |$CatOfCatDatabase|
    |$DomOfCatDatabase| |$JoinOfCatDatabase| |$JoinOfDomDatabase|
    |$attributeDb| |$functionTable| |$existingFiles|
    |$localExposureDataDefault|))
  (|clearCmdAll|)
  (setq |$localExposureData| (copy-seq |$localExposureDataDefault|))
  (setq |$xdatabase| nil)
  (setq |$CatOfCatDatabase| nil)
  (setq |$DomOfCatDatabase| nil)
  (setq |$JoinOfCatDatabase| nil)
  (setq |$JoinOfDomDatabase| nil)
  (setq |$attributeDb| nil)
  (setq |$functionTable| nil)
  (|sayKeyedMsg| 's2iz0013 nil)
  (|clearClams|)
  (|clearConstructorCaches|)
  (setq |$existingFiles| (make-hash-table :test #'equal))
  (|sayKeyedMsg| 's2iz0014 nil)
  (reclaim)
  (|sayKeyedMsg| 's2iz0015 nil))
```

23.3.7 defun clearCmdAll

[clearCmdSortedCaches p481]
 [untraceMapSubNames p847]
 [resetInCoreHist p566]
 [deleteFile p999]
 [histFileName p558]
 [updateCurrentInterpreterFrame p537]
 [clearMacroTable p484]
 [sayKeyedMsg p331]
 [\$frameRecord p885]
 [\$previousBindings p885]
 [\$variableNumberAlist p??]
 [\$InteractiveFrame p??]
 [\$useInternalHistoryTable p557]
 [\$internalHistoryTable p??]
 [\$frameMessages p718]
 [\$interpreterFrameName p??]
 [\$currentLine p??]

— defun clearCmdAll —

```
(defun |clearCmdAll| ()
  (declare (special |$frameRecord| |$previousBindings| |$variableNumberAlist|
    |$InteractiveFrame| |$useInternalHistoryTable| |$internalHistoryTable|
    |$frameMessages| |$interpreterFrameName| |$currentLine|))
  (|clearCmdSortedCaches|)
  (setq |$frameRecord| nil)
  (setq |$previousBindings| nil)
  (setq |$variableNumberAlist| nil)
  (|untraceMapSubNames| /tracenames)
  (setq |$InteractiveFrame| (list (list nil)))
  (|resetInCoreHist|)
  (when |$useInternalHistoryTable|
    (setq |$internalHistoryTable| nil)
    (|deleteFile| (|histFileName|)))
  (setq |$IOindex| 1)
  (|updateCurrentInterpreterFrame|)
  (setq |$currentLine| "clear all")
  (|clearMacroTable|)
  (when |$frameMessages|
    (|sayKeyedMsg| 's2iz0011 (list |$interpreterFrameName|))
    (|sayKeyedMsg| 's2iz0012 nil)))
```

23.3.8 defun clearMacroTable

[*\$pfMacros* *p100*]

— **defun clearMacroTable 0** —

```
(defun |clearMacroTable| ()
  (declare (special |$pfMacros|))
  (setq |$pfMacros| nil))
```

23.3.9 defun clearCmdExcept

Clear all the options except the argument. [*stringPrefix?* *p??*]

[*object2String* *p??*]

[*clearCmdParts* *p484*]

[*\$clearOptions* *p479*]

— **defun clearCmdExcept** —

```
(defun |clearCmdExcept| (arg)
  (let ((opt (car arg)) (vl (cdr arg)))
    (declare (special |$clearOptions|))
    (dolist (option |$clearOptions|)
      (unless (|stringPrefix?| (|object2String| opt)) (|object2String| option))
      (|clearCmdParts| (cons option vl))))))
```

23.3.10 defun clearCmdParts

[*selectOptionLC* *p459*]

[*pname* *p1001*]

[*types* *p??*]

[*modes* *p??*]

[*values* *p??*]

[*boot-equal* *p??*]

[*assocleft* *p??*]

[*remdup* *p??*]

[*assoc* *p??*]

```

[isMap p??]
[get p??]
[pairp p??]
[exit p??]
[untraceMapSubNames p847]
[seq p??]
[recordOldValue p570]
[recordNewValue p569]
[deleteAssoc p??]
[sayKeyedMsg p331]
[getParserMacroNames p433]
[getInterpMacroNames p??]
[clearDependencies p??]
[member p1004]
[clearParserMacro p433]
[sayMessage p??]
[fixObjectForPrinting p436]
[$e p??]
[$InteractiveFrame p??]
[$clearOptions p479]

```

— defun clearCmdParts —

```

(defun |clearCmdParts| (arg)
  (let (|$e| (opt (car arg)) option pmacs imacs (vl (cdr arg)) p1 lm prop p2)
    (declare (special |$e| |$InteractiveFrame| |$clearOptions|))
    (setq option (|selectOptionLC| opt |$clearOptions| '|optionError|))
    (setq option (intern (pname option)))
    (setq option
      (case option
        (|types| '|mode|)
        (|modes| '|mode|)
        (|values| '|value|)
        (t option)))
    (if (null vl)
      (|sayKeyedMsg| 's2iz0055 nil)
      (progn
        (setq pmacs (|getParserMacroNames|))
        (setq imacs (|getInterpMacroNames|))
        (cond
          ((boot-equal vl '(|all|))
           (setq vl (assocleft (caar |$InteractiveFrame|)))
           (setq vl (remdup (append vl pmacs)))))
        (setq |$e| |$InteractiveFrame|)
        (do ((t0 vl (cdr t0)) (x nil))
          ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
          (|clearDependencies| x t)
          (when (and (eq option '|properties|) (|member| x pmacs))

```

```

(|clearParserMacro| x))
(when (and (eq option '|properties|)
           (|member| x imacs)
           (null (|member| x pmacs))))
(|sayMessage| (cons
  "  You cannot clear the definition of the system-defined macro "
  (cons (|fixObjectForPrinting| x)
        (cons (intern "." "BOOT") nil)))))
(cond
  ((setq p1 (|assoc| x (caar |$InteractiveFrame|)))
   (cond
     ((eq option '|properties|)
      (cond
        ((|isMap| x)
         (seq
          (cond
            ((setq lm
              (|get| x '|localModemap| |$InteractiveFrame|))
             (cond
               ((pairp lm)
                (exit (|untraceMapSubNames| (cons (cadar lm) nil))))))
            (t nil))))))
        (dolist (p2 (cdr p1))
         (setq prop (car p2))
         (|recordOldValue| x prop (cdr p2))
         (|recordNewValue| x prop nil))
         (setf (caar |$InteractiveFrame|)
               (|deleteAssoc| x (caar |$InteractiveFrame|))))
        ((setq p2 (|assoc| option (cdr p1)))
         (|recordOldValue| x option (cdr p2))
         (|recordNewValue| x option nil)
         (rplacd p2 nil))))))
  nil)))

```

Chapter 24

)close help page Command

24.1 close help page man page

— close.help —

```
=====
A.5. )close
=====
```

User Level Required: interpreter

Command Syntax:

-)close
-)close)quietly

Command Description:

This command is used to close down interpreter client processes. Such processes are started by HyperDoc to run AXIOM examples when you click on their text. When you have finished examining or modifying the example and you do not want the extra window around anymore, issue

)close

to the AXIOM prompt in the window.

If you try to close down the last remaining interpreter client process, AXIOM will offer to close down the entire AXIOM session and return you to the operating system by displaying something like

This is the last AXIOM session. Do you want to kill AXIOM?

Type "y" (followed by the Return key) if this is what you had in mind. Type "n" (followed by the Return key) to cancel the command.

You can use the)quietly option to force AXIOM to close down the interpreter client process without closing down the entire AXIOM session.

Also See:

- o)quit
- o)pquit

1

24.2 Functions

24.2.1 defun queryClients

Returns the number of active scratchpad clients [sockSendInt p??]
 [sockGetInt p??]
 [\$SessionManager p??]
 [\$QueryClients p??]

— defun queryClients —

```
(defun |queryClients| ()
  (declare (special |$SessionManager| |$QueryClients|))
  (|sockSendInt| |$SessionManager| |$QueryClients|)
  (|sockGetInt| |$SessionManager|))
```

24.2.2 defun close

[throwKeyedMsg p??]
 [sockSendInt p??]
 [closeInterpreterFrame p540]
 [selectOptionLC p459]
 [upcase p??]
 [queryUserKeyedMsg p??]
 [string2id-n p??]

¹ "quit" (41.2.1 p 616) "pquit" (40.2.1 p 612)

```
[queryClients p488]
[$SpadServer p12]
[$SessionManager p??]
[$CloseClient p??]
[$currentFrameNum p43]
[$options p??]
```

— **defun close** —

```
(defun |close| (args)
  (declare (ignore args))
  (let (numClients opt fullopt quiet x)
    (declare (special |$SpadServer| |$SessionManager| |$CloseClient|
      |$currentFrameNum| |$options|))
    (if (null |$SpadServer|)
      (|throwKeyedMsg| 's2iz0071 nil))
    (progn
      (setq numClients (|queryClients|))
      (cond
        ((> numClients 1)
         (|sockSendInt| |$SessionManager| |$CloseClient|)
         (|sockSendInt| |$SessionManager| |$currentFrameNum|)
         (|closeInterpreterFrame| nil))
        (t
         (do ((t0 |$options| (cdr t0)) (t1 nil))
             ((or (atom t0)
                  (progn (setq t1 (car t0)) nil)
                  (progn (progn (setq opt (car t1)) t1) nil))
              nil)
          (setq fullopt (|selectOptionLC| opt '(|quiet|) '|optionError|))
          (unless quiet (setq quiet (eq fullopt '|quiet|))))
         (cond
          (quiet
           (|sockSendInt| |$SessionManager| |$CloseClient|)
           (|sockSendInt| |$SessionManager| |$currentFrameNum|)
           (|closeInterpreterFrame| nil))
          (t
           (setq x (upcase (|queryUserKeyedMsg| 's2iz0072 nil)))
           (when (member (string2id-n x 1) '(yes y)) (bye))))))))))
```


Chapter 25

)compile help page Command

25.1 compile help page man page

— compile.help —

```
=====
A.7. )compile
=====
```

User Level Required: compiler

Command Syntax:

-)compile
-)compile fileName
-)compile fileName.spad
-)compile directory/fileName.spad
-)compile fileName)quiet
-)compile fileName)noquiet
-)compile fileName)break
-)compile fileName)nobreak
-)compile fileName)library
-)compile fileName)nolibrary
-)compile fileName)vartrace
-)compile fileName)constructor nameOrAbbrev

Command Description:

You use this command to invoke the AXIOM library compiler. This compiles files with file extension .spad with the AXIOM system compiler. The command first looks in the standard system directories for files with extension .spad.

Should you not want the `)library` command automatically invoked, call `)compile` with the `)nolibrary` option. For example,

```
)compile mycode )nolibrary
```

By default, the `)library` system command exposes all domains and categories it processes. This means that the AXIOM interpreter will consider those domains and categories when it is trying to resolve a reference to a function. Sometimes domains and categories should not be exposed. For example, a domain may just be used privately by another domain and may not be meant for top-level use. The `)library` command should still be used, though, so that the code will be loaded on demand. In this case, you should use the `)nolibrary` option on `)compile` and the `)noexpose` option in the `)library` command. For example,

```
)compile mycode.spad )nolibrary
)library mycode )noexpose
```

Once you have established your own collection of compiled code, you may find it handy to use the `)dir` option on the `)library` command. This causes `)library` to process all compiled code in the specified directory. For example,

```
)library )dir /u/jones/as/quantum
```

You must give an explicit directory after `)dir`, even if you want all compiled code in the current working directory processed.

```
)library )dir .
```

You can compile category, domain, and package constructors contained in files with file extension `.spad`. You can compile individual constructors or every constructor in a file.

The full filename is remembered between invocations of this command and `)edit` commands. The sequence of commands

```
)compile matrix.spad
)edit
)compile
```

will call the compiler, edit, and then call the compiler again on the file `matrix.spad`. If you do not specify a directory, the working current directory (see description of command `)cd`) is searched for the file. If the file is not found, the standard system directories are searched.

If you do not give any options, all constructors within a file are compiled. Each constructor should have an `)abbreviation` command in the file in which it is defined. We suggest that you place the `)abbreviation` commands at the top of the file in the order in which the constructors are defined. The list of

`commands` serves as a table of contents for the file.

The `)library` option causes directories containing the compiled code for each constructor to be created in the working current directory. The name of such a directory consists of the constructor abbreviation and the `.NRLIB` file extension. For example, the directory containing the compiled code for the `MATRIX` constructor is called `MATRIX.NRLIB`. The `)nolibrary` option says that such files should not be created.

The `)vartrace` option causes the compiler to generate extra code for the constructor to support conditional tracing of variable assignments. (see description of command `)trace`). Without this option, this code is suppressed and one cannot use the `)vars` option for the `trace` command.

The `)constructor` option is used to specify a particular constructor to compile. All other constructors in the file are ignored. The constructor name or abbreviation follows `)constructor`. Thus either

```
)compile matrix.spad )constructor RectangularMatrix
```

or

```
)compile matrix.spad )constructor RMATRIX
```

compiles the `RectangularMatrix` constructor defined in `matrix.spad`.

The `)break` and `)nobreak` options determine what the compiler does when it encounters an error. `)break` is the default and it indicates that processing should stop at the first error. The value of the `)set break` variable then controls what happens.

Also See:

- o `)abbreviation`
- o `)edit`
- o `)library`

1

25.2 Functions

25.2.1 `defvar $/editfile`

— `initvars` —

¹ “abbreviation” (?? p ??) “edit” (30.2.1 p 522) “library” (63.1.34 p 971)

```
(defvar /editfile nil)
```

Chapter 26

)copyright help page Command

26.1 copyright help page man page

— copyright.help —

The term Axiom, in the field of computer algebra software, along with AXIOM and associated images are common-law trademarks. While the software license allows copies, the trademarks may only be used when referring to this project.

Axiom is distributed under terms of the Modified BSD license. Axiom was released under this license as of September 3, 2002. Source code is freely available at:
<http://savannah.nongnu.org/projects/axiom>
Copyrights remain with the original copyright holders. Use of this material is by permission and/or license. Individual files contain reference to these applicable copyrights. The copyright and license statements are collected here for reference.

Portions Copyright (c) 2003- The Axiom Team

The Axiom Team is the collective name for the people who have contributed to this project. Where no other copyright statement is noted in a file this copyright will apply.

Portions Copyright (c) 1991-2002, The Numerical ALgorithms Group Ltd. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of The Numerical Algorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions Copyright (C) 1989-95 GROUPE BULL

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL GROUPE BULL BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of GROUPE BULL shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from GROUPE BULL.

Portions Copyright (C) 2002, Codemist Ltd. All rights reserved.
acn@codemist.co.uk

CCL Public License 1.0
=====

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- (3) Neither the name of Codemist nor the names of other contributors may be used to endorse or promote products derived from this software without specific prior written permission.
- (4) If you distribute a modified form or either source or binary code
 - (a) you must make the source form of these modification available to Codemist;
 - (b) you grant Codemist a royalty-free license to use, modify or redistribute your modifications without limitation;
 - (c) you represent that you are legally entitled to grant these rights and that you are not providing Codemist with any code that violates any law or breaches any contract.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions Copyright (C) 1995-1997 Eric Young (eay@mincom.oz.au)
All rights reserved.

This package is an SSL implementation written
by Eric Young (eay@mincom.oz.au).
The implementation was written so as to conform with Netscapes SSL.

This library is free for commercial and non-commercial use as long as the following conditions are aheared to. The following conditions

apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@mincom.oz.au).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed.

If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used.

This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
 "This product includes cryptographic software written by
 Eric Young (eay@mincom.oz.au)"
 The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related :-).
4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:
 "This product includes software written by Tim Hudson (tjh@mincom.oz.au)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG 'AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The licence and distribution terms for any publically available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.]

Portions Copyright (C) 1988 by Leslie Lamport.

Portions Copyright (c) 1998 Free Software Foundation, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, distribute with modifications, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE ABOVE COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name(s) of the above copyright holders shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization.

Portions Copyright 1989-2000 by Norman Ramsey. All rights reserved.

Noweb is protected by copyright. It is not public-domain software or shareware, and it is not protected by a 'copyleft' agreement like the one used by the Free Software Foundation.

Noweb is available free for any use in any field of endeavor. You may redistribute noweb in whole or in part provided you acknowledge its source and include this COPYRIGHT file. You may modify noweb and create derived works, provided you retain this copyright notice, but the result may not be called noweb without my written consent.

You may sell noweb if you wish. For example, you may sell a CD-ROM including noweb.

You may sell a derived work, provided that all source code for your derived work is available, at no additional charge, to anyone who buys your derived work in any form. You must give permission for said source code to be used and modified under the terms of this license. You must state clearly that your work uses or is based on noweb and that noweb is available free of charge. You must also request that bug reports on your work be reported to you.

Portions Copyright (c) 1987 The RAND Corporation. All rights reserved.

Portions Copyright 1988-1995 by Stichting Mathematisch Centrum, Amsterdam, The Netherlands.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Portions Copyright (c) Renaud Rioboo and the University Paris 6.

Portions Copyright (c) 2003-2010 Jocelyn Guidry

Portions Copyright (c) 2001-2010 Timothy Daly

26.2 Functions

26.2.1 defun copyright

```
[obey p??]
[concat p1003]
[getenvIRON p31]
```

— defun copyright —

```
(defun |copyright| ()
  (obey (concat "cat " (getenvIRON "AXIOM") "/doc/spadhelp/copyright.help")))
```

26.2.2 defun trademark

— defun trademark 0 —

```
(defun |trademark| ()
  (format t "The term Axiom, in the field of computer algebra software, ~%")
  (format t "along with AXIOM and associated images are common-law ~%")
  (format t "trademarks. While the software license allows copies, the ~%")
  (format t "trademarks may only be used when referring to this project ~%"))
```

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

Chapter 27

)credits help page Command

27.1 credits help page man page

27.2 Variables Used

27.3 Functions

27.3.1 defun credits

[credits p503]

— defun credits 0 —

```
(defun |credits| ()  
  (declare (special credits))  
  (mapcar #'(lambda (x) (princ x) (terpri)) credits))
```

—————

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

Chapter 28

)describe help page Command

28.1 describe help page man page

— describe.help —

```
=====
)describe
=====
```

User Level Required: interpreter

Command Syntax:

-)describe categoryName
-)describe domainName
-)describe packageName

Command Description:

This command is used to display the comments for the operation, category, domain or package. The comments are part of the algebra source code.

The commands

```
)describe <categoryName> [internal]
)describe <domainName> [internal]
)describe <packageName> [internal]
```

will show a properly formatted version of the "Description:" keyword from the comments in the algebra source for the category, domain, or package requested.

If 'internal' is requested, then the internal format of the domain or package is described. Categories do not have an internal representation.

28.1.1 defvar \$describeOptions

The current value of \$describeOptions is

— initvars —

```
(defvar $describeOptions '(|category| |domain| |package|))
```

28.2 Functions

28.2.1 defun Print comment strings from algebra libraries

This trivial function satisfies the standard pattern of making a user command match the name of the function which implements the command. That command immediatly invokes a “Spad2Cmd” version. [describepad2cmd p??]

— defun describe —

```
(defun |describe| (l)
  (describeSpad2Cmd l))
```

28.2.2 defun describeSpad2Cmd

The describe command prints cleaned-up comment strings from the algebra libraries. It can print strings associated with a category, domain, package, or by operation.

This implements command line options of the form:

```
)describe categoryName [internal]
)describe domainName   [internal]
)describe packageName   [internal]
```

The describeInternal function will either call the “dc” function to describe the internal representation of the argument or it will print a cleaned up version of the text for the

"Description" keyword in the Category, Domain, or Package source code. [selectOptionLC p459]

[flatten p509]
 [cleanline p507]
 [getdatabase p967]
 [sayMessage p??]
 [\$e p??]
 [\$EmptyEnvironment p??]
 [\$describeOptions p506]

— defun describeSpad2Cmd —

```
(defun describeSpad2Cmd (l)
  (labels (
    (fullname (arg)
      "Convert abbreviations to the full constructor name"
      (let ((abb (getdatabase arg 'abbreviation)))
        (if abb arg (getdatabase arg 'constructor))))
    (describeInternal (cdp internal?)
      (if internal?
        (progn
          (unless (eq (getdatabase cdp 'constructorkind) '|category|) (|dc| cdp))
          (showdatabase cdp))
        (mapcar #'(lambda (x) (if (stringp x) (cleanline x)))
          (flatten (car (getdatabase (fullname cdp) 'documentation)))))))
    (let ((|e| |$EmptyEnvironment|) (opt (second l)))
      (declare (special |e| |$EmptyEnvironment| $describeOptions))
      (if (and (pairp l) (not (eq opt '?)))
        (describeInternal (first l) (second l))
        (|sayMessage|
         (append
          '(" )describe keyword arguments are")
          (mapcar #'(lambda (x) (format nil "~%      ~a" x)) $describeOptions)
          (format nil "~% or abbreviations thereof"))))))))
```

—————

28.2.3 defun cleanline

— defun cleanline —

```
(defun cleanline (line)
  (labels (
    (replaceInLine (thing other line)
      (do ((mark (search thing line) (search thing line)))
          ((null mark) line)
```

```

(setq line
  (concatenate 'string (subseq line 0 mark) other
    (subseq line (+ mark (length thing))))))

(removeFromLine (thing line) (replaceInLine thing "" line))

(removeKeyword (str line)
  (do ((mark (search str line) (search str line)))
    ((null mark) line)
    (let (left point mid right)
      (setq left (subseq line 0 mark))
      (setq point (search "]" line :start2 mark))
      (setq mid (subseq line (+ mark (length str)) point))
      (setq right (subseq line (+ point 1)))
      (setq line (concatenate 'string left mid right)))))

(addSpaces (str line)
  (do ((mark (search str line) (search str line)) (cnt))
    ((null mark) line)
    (let (left point mid right)
      (setq left (subseq line 0 mark))
      (setq point (search "]" line :start2 mark))
      (setq mid (subseq line (+ mark (length str)) point))
      (if (setq cnt (parse-integer mid :junk-allowed t))
        (setq mid (make-string cnt :initial-element #\ ))
        (setq mid ""))
      (setq right (subseq line (+ point 1)))
      (setq line (concatenate 'string left mid right)))))

(splitAtNewline (line)
  (do ((mark (search "%%" line) (search "%%" line)) (lines))
    ((null mark)
     (push " " lines)
     (push line lines)
     (nreverse lines))
    (push (subseq line 0 mark) lines)
    (setq line (subseq line (+ mark 2)))))

(wrapOneLine (line margin result)
  (if (null line)
    (nreverse result)
    (if (< (length line) margin)
      (wrapOneLine nil margin (append (list line) result))
      (let (oneline spill aspace)
        (setq aspace (position #\space (subseq line 0 margin) :from-end t))
        (setq oneline (string-trim '(\space) (subseq line 0 aspace)))
        (setq spill (string-trim '(\space) (subseq line aspace)))
        (wrapOneLine spill margin (append (list oneline) result))))))

(reflowParagraph (line))

```

```

(let (lst1)
  (setq lst1 (splitAtNewLine line))
  (dolist (x lst1)
    (mapcar #'(lambda(y) (format t "~a~%" y))
      (wrapOneLine x 70 nil))))))

(setq line (removeFromLine "{}" line))
(setq line (replaceInLine "\\blankline" "~%~%" line))
(setq line (replaceInLine "\\br" "~%" line))
(setq line (removeFromLine "\\\" line))
(dolist (str '("spad{" "spadtype{" "spadop{" "spadfun{" "spadatt{"
  "axiom{" "axiomType{" "spadignore{" "axiomFun{"
  "centerline{" "inputbitmap{" "axiomOp{" "spadgloss{"))
  (setq line (removeKeyword str line)))
(setq line (replaceInLine "{e.g.}" "e.g." line))
(dolist (str '("tab{" "indented{" ))
  (setq line (addSpaces str line)))
(reflowParagraph line))

```

28.2.4 defun flatten

— defun flatten 0 —

```

(defun flatten (x)
  (labels (
    (rec (x acc)
      (cond
        ((null x) acc)
        ((atom x) (cons x acc))
        (t (rec (car x) (rec (cdr x) acc))))))
    (rec x nil)))

```

Chapter 29

)display help page Command

29.1 display help page man page

— display.help —

```
=====
A.8. )display
=====
```

User Level Required: interpreter

Command Syntax:

-)display all
-)display properties
-)display properties all
-)display properties [obj1 [obj2 ...]]
-)display value all
-)display value [obj1 [obj2 ...]]
-)display mode all
-)display mode [obj1 [obj2 ...]]
-)display names
-)display operations opName

Command Description:

This command is used to display the contents of the workspace and signatures of functions with a given name. (A signature gives the argument and return types of a function.)

The command

`)display names`

lists the names of all user-defined objects in the workspace. This is useful if you do not wish to see everything about the objects and need only be reminded of their names.

The commands

```
)display all
)display properties
)display properties all
```

all do the same thing: show the values and types and declared modes of all variables in the workspace. If you have defined functions, their signatures and definitions will also be displayed.

To show all information about a particular variable or user functions, for example, something named `d`, issue

```
)display properties d
```

To just show the value (and the type) of `d`, issue

```
)display value d
```

To just show the declared mode of `d`, issue

```
)display mode d
```

All modemap for a given operation may be displayed by using `)display operations`. A modemap is a collection of information about a particular reference to an operation. This includes the types of the arguments and the return value, the location of the implementation and any conditions on the types. The modemap may contain patterns. The following displays the modemaps for the operation `FromcomplexComplexCategory`:

```
)d op complex
```

Also See:

- o `)clear`
- o `)history`
- o `)set`
- o `)show`
- o `)what`

29.1.1 defvar \$displayOptions

The current value of \$displayOptions is

— **initvars** —

```
(defvar |$displayOptions|
  '(|abbreviations| |all| |macros| |modes| |names| |operations|
    |properties| |types| |values|))
```

—————

29.2 Functions

29.2.1 defun display

This trivial function satisfies the standard pattern of making a user command match the name of the function which implements the command. That command immediatly invokes a “Spad2Cmd” version. [displaySpad2cmd p??]

— **defun display** —

```
(defun |display| (l)
  (displaySpad2Cmd l))
```

—————

29.2.2 displaySpad2Cmd

We process the options to the command and call the appropriate display function. There are really only 4 display functions. All of the other options are just subcases.

There is a slight mismatch between the \$displayOptions list of symbols and the options this command accepts so we have a cond branch to clean up the option variable. This allows for the options to be plural.

If we fall all the way thru we use the \$displayOptions list to construct a list of strings for the sayMessage function and tell the user what options are available. [abbQuery p514]

```
[opOf p??]
[listConstructorAbbreviations p464]
[displayOperations p515]
[displayMacros p516]
[displayWorkspaceNames p434]
```

```
[displayProperties p441]
[PAIRP p??]
[selectOptionLC p459]
[sayMessage p??]
[$e p??]
[$EmptyEnvironment p??]
[$displayOptions p513]
```

— **defun displaySpad2Cmd** —

```
(defun displaySpad2Cmd (l)
  (let ((|$e| |$EmptyEnvironment|) (opt (car l)) (vl (cdr l)) option)
    (declare (special |$e| |$EmptyEnvironment| |$displayOptions|))
    (if (and (pairp l) (not (eq opt '?)))
        (progn
          (setq option (|selectOptionLC| opt |$displayOptions| '|optionError|))
          (cond
            ((eq option '|all|)
              (setq l (list '|properties|))
              (setq option '|properties|))
            ((or (eq option '|modes|) (eq option '|types|))
              (setq l (cons '|type| vl))
              (setq option '|type|))
            ((eq option '|values|)
              (setq l (cons '|value| vl))
              (setq option '|value|)))
          (cond
            ((eq option '|abbreviations|)
              (if (null vl)
                  (|listConstructorAbbreviations|)
                  (dolist (v vl) (|abbQuery| (|opOf| v))))))
            ((eq option '|operations|) (|displayOperations| vl))
            ((eq option '|macros|) (|displayMacros| vl))
            ((eq option '|names|) (|displayWorkspaceNames|))
            (t (|displayProperties| option l))))
        (|sayMessage|
         (append
          '(" )display keyword arguments are")
          (mapcar #'(lambda (x) (format nil "~%      ~a" x)) |$displayOptions|)
          (format nil "~% or abbreviations thereof"))))))
```

29.2.3 defun abbQuery

```
[getdatabase p967]
[sayKeyedMsg p331]
```

— defun **abbQuery** —

```
(defun |abbQuery| (x)
  (let (abb)
    (cond
      ((setq abb (getdatabase x 'abbreviation))
        (|sayKeyedMsg| 's2iz0001 (list abb (getdatabase x 'constructorkind) x)))
      ((setq abb (getdatabase x 'constructor))
        (|sayKeyedMsg| 's2iz0001 (list x (getdatabase abb 'constructorkind) abb)))
      (t
        (|sayKeyedMsg| 's2iz0003 (list x))))))
```

29.2.4 defun **displayOperations**

This function takes a list of operation names. If the list is null we query the user to see if they want all operations printed. Otherwise we print the information for the requested symbols. [reportOpSymbol p??]

[yesanswer p515]

[sayKeyedMsg p331]

— defun **displayOperations** —

```
(defun |displayOperations| (l)
  (if l
    (dolist (op l) (|reportOpSymbol| op))
    (if (yesanswer)
      (dolist (op (|allOperations|)) (|reportOpSymbol| op))
      (|sayKeyedMsg| 's2iz0059 nil))))
```

29.2.5 defun **yesanswer**

This is a trivial function to simplify the logic of displaySpad2Cmd. If the user didn't supply an argument to the)display op command we ask if they wish to have all information about all Axiom operations displayed. If the answer is either Y or YES we return true else nil.

[string2id-n p??]

[upcase p??]

[queryUserKeyedMsg p??]

— defun **yesanswer** —

```
(defun yesanswer ()
  (member
    (string2id-n (upcase (|queryUserKeyedMsg| 's2iz0058 nil)) 1) '(y yes)))
```

29.2.6 defun displayMacros

```
[getInterpMacroNames p??]
[getParserMacroNames p433]
[remdup p??]
[sayBrightly p??]
[member p1004]
[displayParserMacro p444]
[seq p??]
[exit p??]
[displayMacro p433]
```

— defun displayMacros —

```
(defun |displayMacros| (names)
  (let (imacs pmacs macros first)
    (setq imacs (|getInterpMacroNames|))
    (setq pmacs (|getParserMacroNames|))
    (if names
      (setq macros names)
      (setq macros (append imacs pmacs)))
    (setq macros (remdup macros))
    (cond
      ((null macros) (|sayBrightly| " There are no Axiom macros."))
      (t
       (setq first t)
       (do ((t0 macros (cdr t0)) (macro nil))
         ((or (atom t0) (progn (setq macro (car t0)) nil)) nil)
         (seq
          (exit
           (cond
            ((|member| macro pmacs)
             (cond
              (first (|sayBrightly|
                    (cons '|%l| (cons "User-defined macros:" nil))) (setq first nil)))
              (|displayParserMacro| macro))
            ((|member| macro imacs) '|iterate|)
            (t (|sayBrightly|
                (cons " "
                  (cons '|%b|
                    (cons macro
```

```

      (cons '|%d| (cons " is not a known Axiom macro." nil)))))))))
(setq first t)
(do ((t1 macros (cdr t1)) (macro nil))
  ((or (atom t1) (progn (setq macro (car t1)) nil)) nil)
  (seq
   (exit
    (cond
     ((|member| macro imacs)
      (cond
       ((|member| macro pmacs) '|iterate|)
       (t
        (cond
         (first
          (|sayBrightly|
           (cons '|%1|
            (cons "System-defined macros:" nil))) (setq first nil)))
          (|displayMacro| macro))))
      ((|member| macro pmacs) '|iterate|))))
    nil))))

```

29.2.7 defun sayExample

This function expects 2 arguments, the documentation string and the name of the operation. It searches the documentation string for `++X` lines. These lines are examples lines for functions. They look like ordinary `++` comments and fit into the ordinary comment blocks. So, for example, in the `plot.spad.pamphlet` file we find the following function signature:

```

plot: (F -> F,R) -> %
++ plot(f,a..b) plots the function \spad{f(x)}
++ on the interval \spad{[a,b]}.
++
++X fp:=(t:DFLOAT):DFLOAT +-> sin(t)
++X plot(fp,-1.0..1.0)$PLOT

```

This function splits out and prints the lines that begin with `++X`.

A minor complication of printing the examples is that the lines have been processed into internal compiler format. Thus the lines that read:

```

++X fp:=(t:DFLOAT):DFLOAT +-> sin(t)
++X plot(fp,-1.0..1.0)$PLOT

```

are actually stored as one long line containing the example lines

```

"\indented{1}{plot(\spad{f},{a..\spad{b}}) plots the function

```

```

\\spad{f(x)} \\indented{1}{on the interval \\spad{[a,{b}]}.}
\\blankline
\\spad{X} fp:=(t:DFLOAT):DFLOAT +-> sin(\\spad{t})
\\spad{X} plot(\\spad{fp},{\\spad{-1}.0..1.0)}\\$PLOT"

```

So when we have an example line starting with ++X, it gets converted to the compiler to `\\spad{X}`. So each example line is delimited by `\\spad{X}`.

The compiler also removes the newlines so if there is a subsequent `\\spad{X}` in the docstring then it implies multiple example lines and we loop over them, splitting them up at the delimiter.

If there is only one then we clean it up and print it. [cleanupLine p??]
[sayNewLine p??]

— defun sayExample —

```

(defun sayExample (docstring)
  (let (line point)
    (when (setq point (search "spad{X}" docstring))
      (setq line (subseq docstring (+ point 8)))
      (do ((mark (search "spad{X}" line) (search "spad{X}" line)))
          ((null mark))
          (princ (cleanupLine (subseq line 0 mark)))
          (|sayNewLine|)
          (setq line (subseq line (+ mark 8))))
      (princ (cleanupLine line))
      (|sayNewLine|)
      (|sayNewLine|))))

```

—————

29.2.8 defun cleanupLine

This function expects example lines in internal format that has been partially processed to remove the prefix. Thus we get lines that look like:

```

fp:=(t:DFLOAT):DFLOAT +-> sin(\\spad{t})
plot(\\spad{fp},{\\spad{-1}.0..1.0)}\\$PLOT

```

It removes all instances of `{}`, and `\`, and unwraps the `spad{}` call, leaving only the argument.

We return lines that look like:

```

fp:=(t:DFLOAT):DFLOAT +-> sin(t)
plot(fp,-1.0..1.0)$PLOT

```

which is hopefully exactly what the user wrote.

The compiler inserts {} as a space so we remove it. We remove all of the \ characters. We remove all of the `spad{...}` delimiters which will occur around other `spad` variables. Technically we should search recursively for the matching delimiter rather than the next brace but the problem does not arise in practice.

— **defun cleanupLine 0** —

```
(defun cleanupLine (line)
  (do ((mark (search "{}" line) (search "{}" line)))
      ((null mark))
      (setq line
        (concatenate 'string (subseq line 0 mark) (subseq line (+ mark 2))))))
  (do ((mark (search "\\\" line) (search "\\\" line)))
      ((null mark))
      (setq line
        (concatenate 'string (subseq line 0 mark) (subseq line (+ mark 1))))))
  (do ((mark (search "spad{" line) (search "spad{" line)))
      ((null mark))
      (let (left point mid right)
        (setq left (subseq line 0 mark))
        (setq point (search "}" line :start2 mark))
        (setq mid (subseq line (+ mark 5) point))
        (setq right (subseq line (+ point 1)))
        (setq line (concatenate 'string left mid right))))
  line)
```

Chapter 30

)edit help page Command

30.1 edit help page man page

— edit.help —

```
=====
A.9. )edit
=====
```

User Level Required: interpreter

Command Syntax:

```
- )edit [filename]
```

Command Description:

This command is used to edit files. It works in conjunction with the)read and)compile commands to remember the name of the file on which you are working. By specifying the name fully, you can edit any file you wish. Thus

```
)edit /u/julius/matrix.input
```

will place you in an editor looking at the file /u/julius/matrix.input. By default, the editor is vi, but if you have an EDITOR shell environment variable defined, that editor will be used. When AXIOM is running under the X Window System, it will try to open a separate xterm running your editor if it thinks one is necessary. For example, under the Korn shell, if you issue

```
export EDITOR=emacs
```

then the emacs editor will be used by)edit.

If you do not specify a file name, the last file you edited, read or compiled will be used. If there is no ‘last file’ you will be placed in the editor editing an empty unnamed file.

It is possible to use the `)system` command to edit a file directly. For example,

```
)system emacs /etc/rc.tcpip
```

calls emacs to edit the file.

Also See:

- o `)system`
- o `)compile`
- o `)read`

1

30.2 Functions

30.2.1 `defun edit`

[editSpad2Cmd p522]

— `defun edit` —

```
(defun |edit| (l) (|editSpad2Cmd| l))
```

30.2.2 `defun editSpad2Cmd`

[pathname p998]
 [pathnameDirectory p998]
 [pathnameType p996]
 [\$FINDFILE p??]
 [pathnameName p996]
 [editFile p523]
 [updateSourceFiles p524]
 [/editfile p493]

¹ “system” (?? p ??) “read” (42.1.1 p 620)

— defun editSpad2Cmd —

```
(defun |editSpad2Cmd| (l)
  (let (olddir filetypes ll rc)
    (declare (special /editfile))
    (setq l (cond ((null l) /editfile) (t (car l))))
    (setq l (|pathname| l))
    (setq olddir (|pathnameDirectory| l))
    (setq filetypes
      (cond
        ((|pathnameType| l) (list (|pathnameType| l)))
        ((eq |$UserLevel| '|interpreter|) '("input" "INPUT" "spad" "SPAD"))
        ((eq |$UserLevel| '|compiler|) '("input" "INPUT" "spad" "SPAD"))
        (t '("input" "INPUT" "spad" "SPAD" "boot" "BOOT"
              "lisp" "LISP" "meta" "META"))))
    (setq ll
      (cond
        ((string= olddir "")
         (|pathname| ($findfile (|pathnameName| l) filetypes)))
        (t l)))
    (setq l (|pathname| ll))
    (setq /editfile l)
    (setq rc (|editFile| l))
    (|updateSourceFiles| l)
    rc))
```

—————

30.2.3 defun Implement the)edit command

```
[strconc p??]
[namestring p996]
[pathname p998]
[obey p??]
```

— defun editFile —

```
(defun |editFile| (file)
  (cond
    ((member (intern "WIN32" (find-package 'keyword)) *features*)
     (obey (strconc "notepad " (|namestring| (|pathname| file)))))
    (t
     (obey
      (strconc "$AXIOM/lib/SPAEDIT " (|namestring| (|pathname| file)))))))
```

—————

30.2.4 defun updateSourceFiles

```
[pathname p998]
[pathnameName p996]
[pathnameType p996]
[makeInputFilename p939]
[member p1004]
[pathnameTypeId p997]
[insert p??]
[$sourceFiles p??]
```

— **defun updateSourceFiles** —

```
(defun |updateSourceFiles| (arg)
  (declare (special |$sourceFiles|))
  (setq arg (|pathname| arg))
  (setq arg (|pathname| (list (|pathnameName| arg) (|pathnameType| arg) "*")))
  (when (and (makeInputFilename arg)
             (|member| (|pathnameTypeId| arg) '(boot lisp meta)))
    (setq |$sourceFiles| (|insert| arg |$sourceFiles|)))
  arg)
```

Chapter 31

)fin help page Command

31.1 fin help page man page

— fin.help —

```
=====
A.10. )fin
=====
```

User Level Required: development

Command Syntax:

-)fin

Command Description:

This command is used by AXIOM developers to leave the AXIOM system and return to the underlying Lisp system. To return to AXIOM, issue the ‘‘(spad)’’ function call to Lisp.

Also See:

- o)pquit
- o)quit

1

¹ “pquit” (40.2.1 p 612) “quit” (41.2.1 p 616)

31.1.1 defun Exit from the interpreter to lisp

```
[spad-reader p??]  
[eof p??]
```

— defun fin 0 —

```
(defun |fin| ()  
  (setq *eof* t)  
  (throw 'spad_reader nil))
```

—————

31.2 Functions

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

Chapter 32

)frame help page Command

32.1 frame help page man page

— frame.help —

```
=====
A.11. )frame
=====
```

User Level Required: interpreter

Command Syntax:

```
- )frame new frameName
- )frame drop [frameName]
- )frame next
- )frame last
- )frame names
- )frame import frameName [objectName1 [objectName2 ...] ]
- )set message frame on | off
- )set message prompt frame
```

Command Description:

A frame can be thought of as a logical session within the physical session that you get when you start the system. You can have as many frames as you want, within the limits of your computer's storage, paging space, and so on. Each frame has its own step number, environment and history. You can have a variable named a in one frame and it will have nothing to do with anything that might be called a in any other frame.

Some frames are created by the HyperDoc program and these can have pretty

strange names, since they are generated automatically. To find out the names of all frames, issue

```
)frame names
```

It will indicate the name of the current frame.

You create a new frame ‘‘quark’’ by issuing

```
)frame new quark
```

The history facility can be turned on by issuing either `)set history on` or `)history on`. If the history facility is on and you are saving history information in a file rather than in the AXIOM environment then a history file with filename `quark.akh` will be created as you enter commands. If you wish to go back to what you were doing in the ‘‘initial’’ frame, use

```
)frame next
```

or

```
)frame last
```

to cycle through the ring of available frames to get back to ‘‘initial’’.

If you want to throw away a frame (say ‘‘quark’’), issue

```
)frame drop quark
```

If you omit the name, the current frame is dropped.

If you do use frames with the history facility on and writing to a file, you may want to delete some of the older history files. These are directories, so you may want to issue a command like `rm -r quark.akh` to the operating system.

You can bring things from another frame by using `)frame import`. For example, to bring the `f` and `g` from the frame ‘‘quark’’ to the current frame, issue

```
)frame import quark f g
```

If you want everything from the frame ‘‘quark’’, issue

```
)frame import quark
```

You will be asked to verify that you really want everything.

There are two `)set` flags to make it easier to tell where you are.

```
)set message frame on | off
```

will print more messages about frames when it is set on. By default, it is off.

```
)set message prompt frame
```

will give a prompt that looks like

```
initial (1) ->
```

when you start up. In this case, the frame name and step make up the prompt.

Also See:

- o)history
- o)set

1

32.2 Variables Used

The frame mechanism uses several dollar variables.

32.2.1 Primary variables

Primary variables are those which exist solely to make the frame mechanism work.

The `$interpreterFrameName` contains a symbol which is the name of the current frame in use.

The `$interpreterFrameRing` contains a list of all of the existing frames. The first frame on the list is the “current” frame. When AXIOMsys is started directly there is only one frame named “initial”.

If the system is started under sman (using the axiom shell script, for example), there are two frames, “initial” and “frame0”. In this case, “frame0” is the current frame. This can cause subtle problems because functions defined in the axiom initialization file (`.axiom.input`) will be defined in frame “initial” but the current frame will be “frame0”. They will appear to be undefined. However, if the user does “)frame next” they can switch to the “initial” frame and see the functions correctly defined.

The `$frameMessages` variable controls when frame messages will be displayed. The variable is initially NIL. It can be set on (T) or off (NIL) using the system command:

```
)set message frame on | off
```

¹ “history” (34.4.7 p 560) “set” (44.37.1 p 785)

Setting frame messages on will output a line detailing the current frame after every output is complete.

32.2.2 Used variables

The frame collects and uses a few top level variables. These are: `$InteractiveFrame`, `$IOindex`, `$HiFiAccess`, `$HistList`, `$HistListLen`, `$HistListAct`, `$HistRecord`, `$internalHistoryTable`, and `$localExposureData`.

These variables can also be changed by the frame mechanism when the user requests changing to a different frame.

32.3 Data Structures

32.3.1 Frames and the Interpreter Frame Ring

Axiom has the notion of “frames”. A frame is a data structure which holds all the vital data from an Axiom session. There can be multiple frames and these live in a top-level variable called `$interpreterFrameRing`. This variable holds a circular list of frames. The parts of a frame and their initial, default values are:

<code>\$interpreterFrameName</code>	a string, named on creation
<code>\$InteractiveFrame</code>	(list (list nil))
<code>\$IOindex</code>	an integer, 1
<code>\$HiFiAccess</code>	<code>\$HiFiAccess</code> , see the variable description
<code>\$HistList</code>	<code>\$HistList</code> , see the variable description
<code>\$HistListLen</code>	<code>\$HistListLen</code> , see the variable description
<code>\$HistListAct</code>	<code>\$HistListAct</code> , see the variable description
<code>\$HistRecord</code>	<code>\$HistRecord</code> , see the variable description
<code>\$internalHistoryTable</code>	nil
<code>\$localExposureData</code>	a copy of <code>\$localExposureData</code>

32.4 Accessor Functions

These could be macros but we wish to export them to the API code in the algebra so we keep them as functions.

32.4.1 0th Frame Component – `frameName`

32.4.2 `defun frameName`

— `defun frameName 0` —

```
(defun frameName (frame)
  (car frame))
```

32.4.3 1st Frame Component – frameInteractive

— defun frameInteractive 0 —

```
(defun frameInteractive (frame)
  (nth 1 frame))
```

32.4.4 2nd Frame Component – frameIOIndex

— defun frameIOIndex 0 —

```
(defun frameIOIndex (frame)
  (nth 2 frame))
```

32.4.5 3rd Frame Component – frameHiFiAccess

— defun frameHiFiAccess 0 —

```
(defun frameHiFiAccess (frame)
  (nth 3 frame))
```

32.4.6 4th Frame Component – frameHistList

— defun frameHistList 0 —

```
(defun frameHistList (frame)
  (nth 4 frame))
```

32.4.7 5th Frame Component – frameHistListLen

— defun frameHistListLen 0 —

```
(defun frameHistListLen (frame)
  (nth 5 frame))
```

32.4.8 6th Frame Component – frameHistListAct

— defun frameHistListAct 0 —

```
(defun frameHistListAct (frame)
  (nth 6 frame))
```

32.4.9 7th Frame Component – frameHistRecord

— defun frameHistRecord 0 —

```
(defun frameHistRecord (frame)
  (nth 7 frame))
```

32.4.10 8th Frame Component – frameHistoryTable

— defun frameHistoryTable 0 —

```
(defun frameHistoryTable (frame)
  (nth 8 frame))
```

32.4.11 9th Frame Component – frameExposureData

— defun frameExposureData 0 —

```
(defun frameExposureData (frame)
  (nth 9 frame))
```

32.5 Functions

32.5.1 Initializing the Interpreter Frame Ring

Now that we know what a frame looks like we need a function to initialize the list of frames. This function sets the initial frame name to “initial” and creates a list of frames containing an empty frame. This list is the interpreter frame ring and is not actually circular but is managed as a circular list.

As a final step we update the world from this frame. This has the side-effect of resetting all the important global variables to their initial values.

```
[emptyInterpreterFrame p534]
[updateFromCurrentInterpreterFrame p536]
[$interpreterFrameName p??]
[$interpreterFrameRing p??]
```

— defun initializeInterpreterFrameRing —

```
(defun |initializeInterpreterFrameRing| ()
  "Initializing the Interpreter Frame Ring"
  (declare (special |$interpreterFrameName| |$interpreterFrameRing|))
  (setq |$interpreterFrameName| '|initial|)
  (setq |$interpreterFrameRing|
    (list (|emptyInterpreterFrame| |$interpreterFrameName|)))
  (|updateFromCurrentInterpreterFrame|)
  nil)
```

32.5.2 Creating a List of all of the Frame Names

This function simply walks across the frame in the frame ring and returns a list of the name of each frame. [`$interpreterFrameRing p??`]

— **defun frameNames 0** —

```
(defun |frameNames| ()
  "Creating a List of all of the Frame Names"
  (declare (special |$interpreterFrameRing|))
  (mapcar #'frameName |$interpreterFrameRing|))
```

32.5.3 Get Named Frame Environment (aka Interactive)

If the frame is found we return the environment portion of the frame otherwise we construct an empty environment and return it. The initial values of an empty frame are created here. This function returns a single frame that will be placed in the frame ring. [`frameInteractive p??`]

— **defun frameEnvironment** —

```
(defun |frameEnvironment| (fname)
  "Get Named Frame Environment (aka Interactive)"
  (let ((frame (|findFrameInRing| fname)))
    (if frame
      (frameInteractive frame)
      (list (list nil)))))
```

32.5.4 Create a new, empty Interpreter Frame

```
[|$HiFiAccess p710|
|$HistList p??|
|$HistListLen p??|
|$HistListAct p??|
|$HistRecord p??|
|$localExposureDataDefault p670|
```

— **defun emptyInterpreterFrame 0** —

```
(defun |emptyInterpreterFrame| (name)
```

```
"Create a new, empty Interpreter Frame"
(declare (special |$HiFiAccess| |$HistList| |$HistListLen| |$HistListAct|
| $HistRecord| |$localExposureDataDefault|))
(list name                               ; frame name
  (list (list nil))                     ; environment
  1                                     ; $IOindex
  |$HiFiAccess|
  |$HistList|
  |$HistListLen|
  |$HistListAct|
  |$HistRecord|
  nil                                  ; $internalHistoryTable
  (copy-seq |$localExposureDataDefault|))) ; $localExposureData
```

32.5.5 Collecting up the Environment into a Frame

We can collect up all the current environment information into one frame element with this call. It creates a list of the current values of the global variables and returns this as a frame element.

```
[$interpreterFrameName p??]
[$InteractiveFrame p??]
[$IOindex p??]
[$HiFiAccess p710]
[$HistList p??]
[$HistListLen p??]
[$HistListAct p??]
[$HistRecord p??]
[$internalHistoryTable p??]
[$localExposureData p670]
```

— defun createCurrentInterpreterFrame 0 —

```
(defun |createCurrentInterpreterFrame| ()
  "Collecting up the Environment into a Frame"
  (declare (special |$interpreterFrameName| |$InteractiveFrame| |$IOindex|
| $HiFiAccess| |$HistList| |$HistListLen| |$HistListAct| |$HistRecord|
| $internalHistoryTable| |$localExposureData|))
  (list
    |$interpreterFrameName|
    |$InteractiveFrame|
    |$IOindex|
    |$HiFiAccess|
    |$HistList|
    |$HistListLen|
```



```

|$HistListAct|
|$HistRecord|
|$internalHistoryTable|
|$localExposureData|))

```

32.5.6 Update from the Current Frame

The frames are kept on a circular list. The first element on that list is known as “the current frame”. This will initialize all of the interesting interpreter data structures from that frame.

```

[sayMessage p??]
[$interpreterFrameRing p??]
[$interpreterFrameName p??]
[$InteractiveFrame p??]
[$IOindex p??]
[$HiFiAccess p710]
[$HistList p??]
[$HistListLen p??]
[$HistListAct p??]
[$HistRecord p??]
[$internalHistoryTable p??]
[$localExposureData p670]
[$frameMessages p718]

```

— defun updateFromCurrentInterpreterFrame —

```

(defun |updateFromCurrentInterpreterFrame| ()
  "Update from the Current Frame"
  (let (tmp1)
    (declare (special |$interpreterFrameRing| |$interpreterFrameName|
      |$InteractiveFrame| |$IOindex| |$HiFiAccess| |$HistList| |$HistListLen|
      |$HistListAct| |$HistRecord| |$internalHistoryTable| |$localExposureData|
      |$frameMessages|))
    (setq tmp1 (first |$interpreterFrameRing|))
    (setq |$interpreterFrameName| (nth 0 tmp1))
    (setq |$InteractiveFrame|      (nth 1 tmp1))
    (setq |$IOindex|               (nth 2 tmp1))
    (setq |$HiFiAccess|            (nth 3 tmp1))
    (setq |$HistList|              (nth 4 tmp1))
    (setq |$HistListLen|           (nth 5 tmp1))
    (setq |$HistListAct|           (nth 6 tmp1))
    (setq |$HistRecord|            (nth 7 tmp1))
    (setq |$internalHistoryTable|  (nth 8 tmp1))
    (setq |$localExposureData|     (nth 9 tmp1))
    (when |$frameMessages|

```

```
(|sayMessage|
  '( "    Current interpreter frame is called"
    ,#(|bright| |$interpreterFrameName|))))))
```

32.5.7 Find a Frame in the Frame Ring by Name

Each frame contains its name as the 0th element. We simply walk all the frames and if we find one we return it. [boot-equal p??]

```
[frameName p530]
[$interpreterFrameRing p??]
```

— defun findFrameInRing 0 —

```
(defun |findFrameInRing| (name)
  "Find a Frame in the Frame Ring by Name"
  (let (result)
    (declare (special |$interpreterFrameRing|))
    (dolist (frame |$interpreterFrameRing|)
      (when (boot-equal (frameName frame) name)
        (setq result frame)))
    result))
```

32.5.8 Update the Current Interpreter Frame

This function collects the normal contents of the world into a frame object, places it first on the frame list, and then sets the current values of the world from the frame object.

```
[createCurrentInterpreterFrame p535]
[updateFromCurrentInterpreterFrame p536]
[$interpreterFrameRing p??]
```

— defun updateCurrentInterpreterFrame —

```
(defun |updateCurrentInterpreterFrame| ()
  "Update the Current Interpreter Frame"
  (declare (special |$interpreterFrameRing|))
  (rplaca |$interpreterFrameRing| (|createCurrentInterpreterFrame|))
  (|updateFromCurrentInterpreterFrame|))
```

32.5.9 Move to the next Interpreter Frame in Ring

This function updates the current frame to make sure all of the current information is recorded. If there are more frame elements in the list then this will destructively move the current frame to the end of the list, that is, assume the frame list reads (1 2 3) this function will destructively change it to (2 3 1). Note: the `nconc2` function destructively inserts the second list at the end of the first. [`nconc2 p??`]

[`updateFromCurrentInterpreterFrame p536`]

[`$interpreterFrameRing p??`]

— **defun nextInterpreterFrame** —

```
(defun |nextInterpreterFrame| ()
  "Move to the next Interpreter Frame in Ring"
  (declare (special |$interpreterFrameRing|))
  (when (cdr |$interpreterFrameRing|)
    (setq |$interpreterFrameRing|
      (nconc2 (cdr |$interpreterFrameRing|)
        (list (car |$interpreterFrameRing|))))
    (|updateFromCurrentInterpreterFrame|)))
```

—————

32.5.10 Change to the Named Interpreter Frame

[`updateCurrentInterpreterFrame p537`]

[`findFrameInRing p537`]

[`nremove p??`]

[`updateFromCurrentInterpreterFrame p536`]

[`$interpreterFrameRing p??`]

— **defun changeToNamedInterpreterFrame** —

```
(defun |changeToNamedInterpreterFrame| (name)
  "Change to the Named Interpreter Frame"
  (let (frame)
    (declare (special |$interpreterFrameRing|))
    (|updateCurrentInterpreterFrame|)
    (setq frame (|findFrameInRing| name))
    (when frame
      (setq |$interpreterFrameRing|
        (cons frame (nremove |$interpreterFrameRing| frame)))
      (|updateFromCurrentInterpreterFrame|))))
```

—————

32.5.11 Move to the previous Interpreter Frame in Ring

```
[updateCurrentInterpreterFrame p537]
[nconc2 p??]
[updateFromCurrentInterpreterFrame p536]
[$interpreterFrameRing p??]
```

— **defun previousInterpreterFrame** —

```
(defun |previousInterpreterFrame| ()
  "Move to the previous Interpreter Frame in Ring"
  (let (tmp1 1 b)
    (declare (special |$interpreterFrameRing|))
    (|updateCurrentInterpreterFrame|)
    (when (cdr |$interpreterFrameRing|)
      (setq tmp1 (reverse |$interpreterFrameRing|))
      (setq 1 (car tmp1))
      (setq b (nreverse (cdr tmp1)))
      (setq |$interpreterFrameRing| (nconc2 (cons 1 nil) b))
      (|updateFromCurrentInterpreterFrame|))))
```

32.5.12 Add a New Interpreter Frame

```
[boot-equal p??]
[framename p??]
[throwKeyedMsg p??]
[updateCurrentInterpreterFrame p537]
[initHistList p559]
[emptyInterpreterFrame p534]
[updateFromCurrentInterpreterFrame p536]
[$erase p??]
[histFileName p558]
[$interpreterFrameRing p??]
```

— **defun addNewInterpreterFrame** —

```
(defun |addNewInterpreterFrame| (name)
  "Add a New Interpreter Frame"
  (declare (special |$interpreterFrameRing|))
  (if (null name)
    (|throwKeyedMsg| 's2iz0018 nil) ; you must provide a name for new frame
    (progn
      (|updateCurrentInterpreterFrame|)
      (dolist (f |$interpreterFrameRing|)
```

```

      (when (boot-equal name (frameName f)) ; existing frame with same name
        (|throwKeyedMsg| 's2iz0019 (list name))))
(|initHistList|)
(setq |$interpreterFrameRing|
  (cons (|emptyInterpreterFrame| name) |$interpreterFrameRing|))
(|updateFromCurrentInterpreterFrame|)
($erase (|histFileName|))))

```

32.5.13 Close an Interpreter Frame

```

[nequal p??]
[frameName p??]
[throwKeyedMsg p??]
[$erase p??]
[makeHistFileName p557]
[updateFromCurrentInterpreterFrame p536]
[$interpreterFrameRing p??]
[$interpreterFrameName p??]

```

— defun closeInterpreterFrame —

```

(defun |closeInterpreterFrame| (name)
  "Close an Interpreter Frame"
  (declare (special |$interpreterFrameRing| |$interpreterFrameName|))
  (let (ifr found)
    (if (null (cdr |$interpreterFrameRing|))
      (if (and name (nequal name |$interpreterFrameName|))
        (|throwKeyedMsg| 's2iz0020 ; 1 frame left. not the correct name.
          (cons |$interpreterFrameName| nil))
        (|throwKeyedMsg| 's2iz0021 nil)) ; only 1 frame left, not closed
      (progn
        (if (null name)
          (setq |$interpreterFrameRing| (cdr |$interpreterFrameRing|))
          (progn
            (setq found nil)
            (setq ifr nil)
            (dolist (f |$interpreterFrameRing|)
              (if (or found (nequal name (frameName f)))
                (setq ifr (cons f ifr)))
              (setq found t)))
            (if (null found)
              (|throwKeyedMsg| 's2iz0022 (cons name nil))
              (progn
                ($erase (|makeHistFileName| name))
                (setq |$interpreterFrameRing| (nreverse ifr)))))))

```

```
(|updateFromCurrentInterpreterFrame|))))))
```

32.5.14 Display the Frame Names

```
[bright p??]  
[frameName p??]  
[sayKeyedMsg p331]  
[$interpreterFrameRing p??]
```

— **defun displayFrameNames** —

```
(defun |displayFrameNames| ()  
  "Display the Frame Names"  
  (declare (special |$interpreterFrameRing|))  
  (let (t1)  
    (setq t1  
      (mapcar #'(lambda (f) '(|%1| "      " ,@(|bright| (frameName f))))  
              |$interpreterFrameRing|))  
    (|sayKeyedMsg| 's2iz0024 (list (apply #'append t1)))))
```

32.5.15 Import items from another frame

```
[member p1004]  
[frameNames p534]  
[throwKeyedMsg p??]  
[boot-equal p??]  
[frameName p??]  
[frameEnvironment p534]  
[upcase p??]  
[queryUserKeyedMsg p??]  
[string2id-n p??]  
[importFromFrame p541]  
[sayKeyedMsg p331]  
[clearCmdParts p484]  
[seq p??]  
[exit p??]  
[putHist p568]  
[get p??]  
[getalist p??]  
[$interpreterFrameRing p??]
```

— defun importFromFrame —

```

(defun |importFromFrame| (args)
  "Import items from another frame"
  (prog (tmp1 fname fenv x v props vars plist prop val m)
    (declare (special |$interpreterFrameRing|))
    (when (and args (atom args)) (setq args (cons args nil)))
    (if (null args)
      (|throwKeyedMsg| 'S2IZ0073 nil) ; missing frame name
      (progn
        (setq tmp1 args)
        (setq fname (car tmp1))
        (setq args (cdr tmp1))
        (cond
          ((null (|member| fname (|frameNames|)))
            (|throwKeyedMsg| 'S2IZ0074 (cons fname nil))) ; not frame name
          ((boot-equal fname (frameName (car |$interpreterFrameRing|)))
            (|throwKeyedMsg| 'S2IZ0075 NIL)) ; cannot import from curr frame
          (t
            (setq fenv (|frameEnvironment| fname))
            (cond
              ((null args)
                (setq x
                  (upcase (|queryUserKeyedMsg| 'S2IZ0076 (cons fname nil))))
                  ; import everything?
              (cond
                ((member (string2id-n x 1) '(y yes))
                  (setq vars nil)
                  (do ((tmp0 (caar fenv) (cdr tmp0)) (tmp1 nil))
                    ((or (atom tmp0)
                        (progn (setq tmp1 (car tmp0)) nil)
                        (progn
                          (progn
                            (setq v (car tmp1))
                            (setq props (cdr tmp1))
                            tmp1
                            nil))
                      nil))
                  (cond
                    ((eq v '|--macros|)
                     (do ((tmp2 props (cdr tmp2))
                         (tmp3 nil))
                       ((or (atom tmp2)
                           (progn (setq tmp3 (car tmp2)) nil)
                           (progn
                             (progn (setq m (car tmp3)) tmp3)
                             nil))
                     nil)
                     (setq vars (cons m vars))))
                (t
                  nil))
              (t
                nil))
            (setq vars (cons m vars))))))

```

```

      (t (setq vars (cons v vars))))))
    (|importFromFrame| (cons fname vars)))
  (t
    (|sayKeyedMsg| 'S2IZ0077 (cons fname nil))))))
(t
  (do ((tmp4 args (cdr tmp4)) (v nil))
    ((or (atom tmp4) (progn (setq v (car tmp4)) nil)) nil)
    (seq
      (exit
        (progn
          (setq plist (getalist (caar fenv) v))
          (cond
            (plist
              (|clearCmdParts| (cons '|propert| (cons v nil)))
              (do ((tmp5 plist (cdr tmp5)) (tmp6 nil))
                ((or (atom tmp5)
                    (progn (setq tmp6 (car tmp5)) nil)
                    (progn
                      (progn
                        (setq prop (car tmp6))
                        (setq val (cdr tmp6))
                        tmp6)
                      nil)))
                nil)
              (seq
                (exit (|putHist| v prop val |$InteractiveFrame|))))
                ((setq m (|get| '|--macros--| v fenv))
                  (|putHist| '|--macros--| v m |$InteractiveFrame|))
                (t
                  (|sayKeyedMsg| 'S2IZ0079 ; frame not found
                    (cons v (cons fname nil))))))))))
    (|sayKeyedMsg| 'S2IZ0078 ; import complete
      (cons fname nil)))))))))

```

32.5.16 The top level frame command

[frameSpad2Cmd p544]

— defun frame —

```

(defun |frame| (l)
  "The top level frame command"
  (|frameSpad2Cmd| l))

```

32.5.17 The top level frame command handler

```
[throwKeyedMsg p??]
[helpSpad2Cmd p550]
[selectOptionLC p459]
[pairp p??]
[qcdr p??]
[qcar p??]
[object2Identifier p??]
[frameSpad2Cmd drop (vol9)]
[closeInterpreterFrame p540]
[import p??]
[importFromFrame p541]
[last p??]
[previousInterpreterFrame p539]
[names p??]
[displayFrameNames p541]
[new p??]
[addNewInterpreterFrame p539]
[next p38]
[nextInterpreterFrame p538]
[$options p??]
```

— defun frameSpad2Cmd —

```
(defun |frameSpad2Cmd| (args)
  "The top level frame command handler"
  (let (frameArgs arg a)
    (declare (special |$options|))
    (setq frameArgs '(|drop| |import| |last| |names| |new| |next|))
    (cond
      (|$options|
       (|throwKeyedMsg| 'S2IZ0016 ; frame command does not take options
        (cons ")frame" nil)))
      ((null args) (|helpSpad2Cmd| (cons '|frame| nil)))
      (t
       (setq arg (|selectOptionLC| (car args) frameArgs '|optionError|))
       (setq args (cdr args))
       (when (and (pairp args)
                  (eq (qcdr args) nil)
                  (progn (setq a (qcar args)) t))
        (setq args a))
       (when (atom args) (setq args (|object2Identifier| args)))
       (case arg
         (|drop|
          (if (and args (pairp args))
              (|throwKeyedMsg| 'S2IZ0017 ; not a valid frame name
               (cons args nil))
```

```

      (|closeInterpreterFrame| args)))
(|import| (|importFromFrame| args))
(|last| (|previousInterpreterFrame|))
(|names| (|displayFrameNames|))
(|new|
  (if (and args (pairp args))
    (|throwKeyedMsg| 'S2IZ0017 ; not a valid frame name
      (cons args nil))
    (|addNewInterpreterFrame| args)))
(|next| (|nextInterpreterFrame|))
(t nil))))))

```

32.6 Frame File Messages

— Frame File Messages —

S2IZ0016

The %1b system command takes arguments but no options.

S2IZ0017

%1b is not a valid frame name

S2IZ0018

You must provide a name for the new frame.

S2IZ0019

You cannot use the name %1b for a new frame because an existing frame already has that name.

S2IZ0020

There is only one frame active and therefore that cannot be closed. Furthermore, the frame name you gave is not the name of the current frame. The current frame is called %1b .

S2IZ0021

The current frame is the only active one. Issue %b)clear all %d to clear its contents.

S2IZ0022

There is no frame called %1b and so your command cannot be processed.

S2IZ0024

The names of the existing frames are: %1 %l

The current frame is the first one listed.

S2IZ0073

%b)frame import %d must be followed by the frame name. The names of objects in that frame can then optionally follow the frame name. For example,

%ceon %b)frame import calculus %d %ceoff

imports all objects in the %b calculus %d frame, and

%ceon %b)frame import calculus epsilon delta %d %ceoff
imports the objects named %b epsilon %d and %b delta %d from the
frame %b calculus %d .

Please note that if the current frame contained any information
about objects with these names, then that information would be
cleared before the import took place.

S2IZ0074

You cannot import anything from the frame %1b because that is not
the name of an existing frame.

S2IZ0075

You cannot import from the current frame (nor is there a need!).

S2IZ0076

User verification required:

do you really want to import everything from the frame %1b ?

If so, please enter %b y %d or %b yes %d :

S2IZ0077

On your request, AXIOM will not import everything from frame %1b.

S2IZ0078

Import from frame %1b is complete. Please issue %b)display all %d
if you wish to see the contents of the current frame.

S2IZ0079

AXIOM cannot import %1b from frame %2b because it cannot be found.

Chapter 33

)help help page Command

33.1 help help page man page

— help.help —

```
=====
A.12.  )help
=====
```

User Level Required: interpreter

Command Syntax:

-)help
-)help commandName
-)help syntax

Command Description:

This command displays help information about system commands. If you issue

)help

then this very text will be shown. You can also give the name or abbreviation of a system command to display information about it. For example,

)help clear

will display the description of the)clear system command.

The command

)help syntax

will give further information about the Axiom language syntax.

All this material is available in the AXIOM User Guide and in HyperDoc. In HyperDoc, choose the Commands item from the Reference menu.

```
=====
A.1. Introduction
=====
```

System commands are used to perform AXIOM environment management. Among the commands are those that display what has been defined or computed, set up multiple logical AXIOM environments (frames), clear definitions, read files of expressions and commands, show what functions are available, and terminate AXIOM.

Some commands are restricted: the commands

```
)set userlevel interpreter
)set userlevel compiler
)set userlevel development
```

set the user-access level to the three possible choices. All commands are available at development level and the fewest are available at interpreter level. The default user-level is interpreter. In addition to the)set command (discussed in description of command)set) you can use the HyperDoc settings facility to change the user-level. Click on [Settings] here to immediately go to the settings facility.

Each command listing begins with one or more syntax pattern descriptions plus examples of related commands. The syntax descriptions are intended to be easy to read and do not necessarily represent the most compact way of specifying all possible arguments and options; the descriptions may occasionally be redundant.

All system commands begin with a right parenthesis which should be in the first available column of the input line (that is, immediately after the input prompt, if any). System commands may be issued directly to AXIOM or be included in .input files.

A system command argument is a word that directly follows the command name and is not followed or preceded by a right parenthesis. A system command option follows the system command and is directly preceded by a right parenthesis. Options may have arguments: they directly follow the option. This example may make it easier to remember what is an option and what is an argument:

```
)syscmd arg1 arg2 )opt1 opt1arg1 opt1arg2 )opt2 opt2arg1 ...
```

In the system command descriptions, optional arguments and options are enclosed in brackets (`'['` and `']'`). If an argument or option name is in italics, it is meant to be a variable and must have some actual value substituted for it when the system command call is made. For example, the syntax pattern description

```
)read fileName [quietly]
```

would imply that you must provide an actual file name for `fileName` but need not use the `)quietly` option. Thus

```
)read matrix.input
```

is a valid instance of the above pattern.

System command names and options may be abbreviated and may be in upper or lower case. The case of actual arguments may be significant, depending on the particular situation (such as in file names). System command names and options may be abbreviated to the minimum number of starting letters so that the name or option is unique. Thus

```
)s Integer
```

is not a valid abbreviation for the `)set` command, because both `)set` and `)show` begin with the letter `'s'`. Typically, two or three letters are sufficient for disambiguating names. In our descriptions of the commands, we have used no abbreviations for either command names or options.

In some syntax descriptions we use a vertical line `'|'` to indicate that you must specify one of the listed choices. For example, in

```
)set output fortran on | off
```

only `on` and `off` are acceptable words for following `boot`. We also sometimes use `'...'` to indicate that additional arguments or options of the listed form are allowed. Finally, in the syntax descriptions we may also list the syntax of related commands.

```
=====
Other help topics
=====
```

Available help topics are:

abbreviations	assignment	blocks	browse	boot	cd
clear	clef	close	collection	compile	describe
display	edit	fin	for	frame	help
history	if	iterate	leave	library	lisp
load	ltrace	parallel	pquit	quit	read
repeat	savesystem	set	show	spool	suchthat

synonym system syntax trace undo what
while

Available algebra help topics are:

—————

33.2 Functions

33.2.1 The top level help command

[helpSpad2Cmd p550]

— defun help —

```
(defun |help| (l)
  "The top level help command"
  (|helpSpad2Cmd| l))
```

—————

33.2.2 The top level help command handler

[newHelpSpad2Cmd p550]

[sayKeyedMsg p331]

— defun helpSpad2Cmd —

```
(defun |helpSpad2Cmd| (args)
  "The top level help command handler"
  (unless (|newHelpSpad2Cmd| args)
    (|sayKeyedMsg| 's2iz0025 (cons args nil))))
```

—————

33.2.3 defun newHelpSpad2Cmd

[makeInputFilename p939]

[obey p??]

[concat p1003]

[namestring p996]

```
[make-instream p937]
[say p??]
[poundsign p??]
[sayKeyedMsg p331]
[pname p1001]
[selectOptionLC p459]
[$syscommands p424]
[$useFullScreenHelp p709]
```

— **defun newHelpSpad2Cmd** —

```
(defun |newHelpSpad2Cmd| (args)
  (let (sarg arg narg helpfile filestream line)
    (declare (special $syscommands |$useFullScreenHelp|))
    (when (null args) (setq args (list '???)))
    (if (> (|#| args) 1)
      (|sayKeyedMsg| 's2iz0026 nil)
      (progn
        (setq sarg (pname (car args)))
        (cond
          ((string= sarg "?") (setq args (list '|help|)))
          ((string= sarg "%") (setq args (list '|history|)))
          ((string= sarg "%%") (setq args (list '|history|)))
          (t nil))
        (setq arg (|selectOptionLC| (car args) $syscommands nil))
        (cond ((null arg) (setq arg (car args))))
        (setq narg (pname arg))
        (cond
          ((null (setq helpfile (makeInputFilename (list narg "help"))))
           nil)
          (|$useFullScreenHelp|
           (obey (concat "$AXIOM/lib/SPAEDIT " (|namestring| helpfile))) t)
          (t
           (setq filestream (make-instream helpfile))
           (do ((line (|read-line| filestream nil) (|read-line| filestream nil)))
               ((null line) (shut filestream))
               (say line))))))))))
```

Chapter 34

)history help page Command

34.1 history help page man page

— history.help —

```
=====
A.13. )history
=====
```

User Level Required: interpreter

Command Syntax:

```
- )history )on
- )history )off
- )history )write historyInputFileName
- )history )show [n] [both]
- )history )save savedHistoryName
- )history )restore [savedHistoryName]
- )history )reset
- )history )change n
- )history )memory
- )history )file
- %
- %% (n)
- )set history on | off
```

Command Description:

The history facility within AXIOM allows you to restore your environment to that of another session and recall previous computational results. Additional commands allow you to review previous input lines and to create an .input

file of the lines typed to AXIOM.

AXIOM saves your input and output if the history facility is turned on (which is the default). This information is saved if either of

```
)set history on
)history )on
```

has been issued. Issuing either

```
)set history off
)history )off
```

will discontinue the recording of information.

Whether the facility is disabled or not, the value of % in AXIOM always refers to the result of the last computation. If you have not yet entered anything, % evaluates to an object of type Variable('%'). The function %% may be used to refer to other previous results if the history facility is enabled. In that case, %(n) is the output from step n if n > 0. If n < 0, the step is computed relative to the current step. Thus %(-1) is also the previous step, %(-2), is the step before that, and so on. If an invalid step number is given, AXIOM will signal an error.

The environment information can either be saved in a file or entirely in memory (the default). Each frame (description of command)frame) has its own history database. When it is kept in a file, some of it may also be kept in memory for efficiency. When the information is saved in a file, the name of the file is of the form FRAME.akh where 'FRAME' is the name of the current frame. The history file is placed in the current working directory (see description of command)cd). Note that these history database files are not text files (in fact, they are directories themselves), and so are not in human-readable format.

The options to the)history command are as follows:

```
)change n
    will set the number of steps that are saved in memory to n. This option
    only has effect when the history data is maintained in a file. If you
    have issued )history )memory (or not changed the default) there is no
    need to use )history )change.
```

```
)on
    will start the recording of information. If the workspace is not empty,
    you will be asked to confirm this request. If you do so, the workspace
    will be cleared and history data will begin being saved. You can also
    turn the facility on by issuing )set history on.
```

```
)off
    will stop the recording of information. The )history )show command will
```

not work after issuing this command. Note that this command may be issued to save time, as there is some performance penalty paid for saving the environment data. You can also turn the facility off by issuing `)set history off`.

`)file`

indicates that history data should be saved in an external file on disk.

`)memory`

indicates that all history data should be kept in memory rather than saved in a file. Note that if you are computing with very large objects it may not be practical to keep this data in memory.

`)reset`

will flush the internal list of the most recent workspace calculations so that the data structures may be garbage collected by the underlying Lisp system. Like `)history)change`, this option only has real effect when history data is being saved in a file.

`)restore [savedHistoryName]`

completely clears the environment and restores it to a saved session, if possible. The `)save` option below allows you to save a session to a file with a given name. If you had issued `)history)save jacobi` the command `)history)restore jacobi` would clear the current workspace and load the contents of the named saved session. If no saved session name is specified, the system looks for a file called `last.axh`.

`)save savedHistoryName`

is used to save a snapshot of the environment in a file. This file is placed in the current working directory (see description of command `)cd`). Use `)history)restore` to restore the environment to the state preserved in the file. This option also creates an input file containing all the lines of input since you created the workspace frame (for example, by starting your AXIOM session) or last did a `)clear all` or `)clear completely`.

`)show [n] [both]`

can show previous input lines and output results. `)show` will display up to twenty of the last input lines (fewer if you haven't typed in twenty lines). `)show n` will display up to `n` of the last input lines. `)show both` will display up to five of the last input lines and output results. `)show n both` will display up to `n` of the last input lines and output results.

`)write historyInputFile`

creates an `.input` file with the input lines typed since the start of the session/frame or the last `)clear all` or `)clear completely`. If `historyInputFileName` does not contain a period (`'.'`) in the filename, `.input` is appended to it. For example, `)history)write chaos` and `)history)write chaos.input` both write the input lines to a file called `chaos.input` in your current working directory. If you issued one or more

)undo commands,)history)write eliminates all input lines backtracked over as a result of)undo. You can edit this file and then use)read to have AXIOM process the contents.

Also See:

- o)frame
- o)read
- o)set
- o)undo

1

History recording is done in two different ways:

- all changes in variable bindings (i.e. previous values) are written to `$HistList`, which is a circular list
- all new bindings (including the binding to `%`) are written to a file called `histFileName()` one older session is accessible via the file `$oldHistFileName()`

34.2 Initialized history variables

The following global variables are used:

`$HistList`, `$HistListLen` and `$HistListAct` which is the actual number of “undoable” steps)

`$HistRecord` collects the input line, all variable bindings and the output of a step, before it is written to the file `histFileName()`.

`$HiFiAccess` is a flag, which is reset by `)history)off`

The result of step `n` can be accessed by `%n`, which is translated into a call of `fetchOutput(n)`. The `updateHist` is called after every interpreter step. The `putHist` function records all changes in the environment to `$HistList` and `$HistRecord`.

34.2.1 defvar \$oldHistoryFileName

— initvars —

```
(defvar |$oldHistoryFileName| ' |last| "vm/370 filename name component")
```

¹ “frame” (32.5.16 p 543) “read” (42.1.1 p 620) “set” (44.37.1 p 785) “undo” (51.3.6 p 886)

34.2.2 defvar \$historyFileType

— initvars —

```
(defvar |$historyFileType| ' |axh|      "vm/370 filename type component")
```

—

34.2.3 defvar \$historyDirectory

— initvars —

```
(defvar |$historyDirectory| 'A      "vm/370 filename disk component")
```

—

34.2.4 defvar \$useInternalHistoryTable

— initvars —

```
(defvar |$useInternalHistoryTable| t  "t means keep history in core")
```

—

34.3 Data Structures**34.4 Functions****34.4.1 defun makeHistFileName**

```
[makePathname p998]
```

— defun makeHistFileName —

```
(defun |makeHistFileName| (fname)
  (|makePathname| fname |$historyFileType| |$historyDirectory|))
```

—

34.4.2 defun oldHistFileName

```
[makeHistFileName p557]
[$oldHistoryFileName p556]
```

— **defun oldHistFileName** —

```
(defun |oldHistFileName| ()
  (declare (special |$oldHistoryFileName|))
  (|makeHistFileName| |$oldHistoryFileName|))
```

—————

34.4.3 defun histFileName

```
[makeHistFileName p557]
[$interpreterFrameName p??]
```

— **defun histFileName** —

```
(defun |histFileName| ()
  (declare (special |$interpreterFrameName|))
  (|makeHistFileName| |$interpreterFrameName|))
```

—————

34.4.4 defun histInputFileName

```
[makePathname p998]
[$interpreterFrameName p??]
[$historyDirectory p557]
```

— **defun histInputFileName** —

```
(defun |histInputFileName| (fn)
  (declare (special |$interpreterFrameName| |$historyDirectory|))
  (if (null fn)
    (|makePathname| |$interpreterFrameName| 'input |$historyDirectory|)
    (|makePathname| fn 'input |$historyDirectory|)))
```

—————

34.4.5 defun initHist

```
[initHistList p559]
[oldHistFileName p558]
[histFileName p558]
[histFileErase p596]
[makeInputFilename p939]
[$replace p??]
[$useInternalHistoryTable p557]
[$HiFiAccess p710]
```

— **defun initHist** —

```
(defun |initHist| ()
  (let (oldFile newFile)
    (declare (special |$useInternalHistoryTable| |$HiFiAccess|))
    (if |$useInternalHistoryTable|
      (|initHistList|)
      (progn
        (setq oldFile (|oldHistFileName|))
        (setq newFile (|histFileName|))
        (|histFileErase| oldFile)
        (when (makeInputFilename newFile) (replaceFile oldFile newFile))
        (setq |$HiFiAccess| t)
        (|initHistList|))))))
```

34.4.6 defun initHistList

```
[$HistListLen p??]
[$HistList p??]
[$HistListAct p??]
[$HistRecord p??]
```

— **defun initHistList** —

```
(defun |initHistList| ()
  (let (li)
    (declare (special |$HistListLen| |$HistList| |$HistListAct| |$HistRecord|))
    (setq |$HistListLen| 20)
    (setq |$HistList| (list nil))
    (setq li |$HistList|)
    (do ((i 1 (1+ i)))
      ((> i |$HistListLen|) nil)
      (setq li (cons nil li))))
```



```
(rplacd |$HistList| li)
(setq |$HistListAct| 0)
(setq |$HistRecord| nil)))
```

34.4.7 The top level history command

```
[sayKeyedMsg p331]
[historySpad2Cmd p560]
[$options p??]
```

— defun history —

```
(defun |history| (l)
  "The top level history command"
  (declare (special |$options|))
  (if (or l (null |$options|))
    (|sayKeyedMsg| 's2ih0006 nil) ; syntax error
    (|historySpad2Cmd|)))
```

34.4.8 The top level history command handler

```
[selectOptionLC p459]
[member p1004]
[sayKeyedMsg p331]
[initHistList p559]
[upcase p??]
[queryUserKeyedMsg p??]
[string2id-n p??]
[histFileErase p596]
[histFileName p558]
[clearSpad2Cmd p480]
[disableHist p581]
[setHistoryCore p562]
[resetInCoreHist p566]
[saveHistory p573]
[showHistory p??]
[changeHistListLen p567]
[restoreHistory p575]
[writeInputLines p565]
[seq p??]
```

```
[exit p??]
[options p??]
[HiFiAccess p710]
[IOindex p??]
```

— defun historySpad2Cmd —

```
(defun |historySpad2Cmd| ()
  "The top level history command handler"
  (let (histOptions opts opt optargs x)
    (declare (special |$options| |$HiFiAccess| |$IOindex|))
    (setq histOptions
      '(|on| |off| |yes| |no| |change| |reset| |restore| |write|
        |save| |show| |file| |memory|))
    (setq opts
      (prog (tmp1)
        (setq tmp1 nil)
        (return
          (do ((tmp2 |$options| (cdr tmp2)) (tmp3 nil))
              ((or (atom tmp2)
                    (progn
                      (setq tmp3 (car tmp2))
                      nil)
                    (progn
                      (progn
                        (setq opt (car tmp3))
                        (setq optargs (cdr tmp3))
                        tmp3)
                      nil)))
              (nreverse0 tmp1)))
          (setq tmp1
            (cons
              (cons
                (|selectOptionLC| opt histOptions '|optionError|)
                optargs)
              tmp1))))))
    (do ((tmp4 opts (cdr tmp4)) (tmp5 nil))
        ((or (atom tmp4)
              (progn
                (setq tmp5 (car tmp4))
                nil)
              (progn
                (progn
                  (setq opt (car tmp5))
                  (setq optargs (cdr tmp5))
                  tmp5)
                nil)))
        nil)
    (seq
```

```

(exit
(cond
  ((member| opt '(|on| |yes|))
    (cond
      (|$HiFiAccess|
        (|sayKeyedMsg| 'S2IH0007 nil)) ; history already on
      ((eq| |$IOindex| 1)
        (setq |$HiFiAccess| t)
        (|initHistList|)
        (|sayKeyedMsg| 'S2IH0008 nil)) ; history now on
      (t
        (setq x ; really want to turn history on?
          (upcase (|queryUserKeyedMsg| 'S2IH0009 nil)))
        (cond
          ((member (string2id-n x 1) '(Y YES))
            (|histFileErase| (|histFileName|))
            (setq |$HiFiAccess| t)
            (setq |$options| nil)
            (|clearSpad2Cmd| '(|all|))
            (|sayKeyedMsg| 'S2IH0008 nil) ; history now on
            (|initHistList|))
          (t
            (|sayKeyedMsg| 'S2IH0010 nil)))))) ; history still off
    ((member| opt '(|off| |no|))
      (cond
        ((null |$HiFiAccess|)
          (|sayKeyedMsg| 'S2IH0011 nil)) ; history already off
        (t
          (setq |$HiFiAccess| nil)
          (|disableHist|)
          (|sayKeyedMsg| 'S2IH0012 nil)))) ; history now off
      ((eq opt '|file|) (|setHistoryCore| nil))
      ((eq opt '|memory|) (|setHistoryCore| t))
      ((eq opt '|reset|) (|resetInCoreHist|))
      ((eq opt '|save|) (|saveHistory| optargs))
      ((eq opt '|show|) (|showHistory| optargs))
      ((eq opt '|change|) (|changeHistListLen| (car optargs)))
      ((eq opt '|restore|) (|restoreHistory| optargs))
      ((eq opt '|write|) (|writeInputLines| optargs 1))))))
'|done|))

```

34.4.9 defun setHistoryCore

We case on the inCore argument value

If history is already on and is kept in the same location as requested (file or memory)

then complain.

If history is not in use then start using the file or memory as requested. This is done by simply setting the `$useInternalHistoryTable` to the requested value, where T means use memory and NIL means use a file. We tell the user.

If history should be in memory, that is `inCore` is not NIL, and the history file already contains information we read the information from the file, store it in memory, and erase the history file. We modify `$useInternalHistoryTable` to T to indicate that we're maintaining the history in memory and tell the user.

Otherwise history must be on and in memory. We erase any old history file and then write the in-memory history to a new file

```
[boot-equal p??]
[sayKeyedMsg p331]
[nequal p??]
[rkeyids p??]
[histFileName p558]
[readHiFi p579]
[disableHist p581]
[histFileErase p596]
[rdefiostream p??]
[spadrwrite p583]
[object2Identifier p??]
[rshut p??]
[$useInternalHistoryTable p557]
[$internalHistoryTable p??]
[$HiFiAccess p710]
[$IOindex p??]
```

— **defun setHistoryCore** —

```
(defun |setHistoryCore| (inCore)
  (let (l vec str n rec)
    (declare (special |$useInternalHistoryTable| |$internalHistoryTable|
                      |$HiFiAccess| |$IOindex|))
    (cond
      ((boot-equal inCore |$useInternalHistoryTable|)
       (if inCore
           (|sayKeyedMsg| 's2ih0030 nil) ; memory history already in use
           (|sayKeyedMsg| 's2ih0029 nil))) ; file history already in use
      ((null |$HiFiAccess|)
       (setq |$useInternalHistoryTable| inCore)
       (if inCore
           (|sayKeyedMsg| 's2ih0032 nil) ; use memory history
           (|sayKeyedMsg| 's2ih0031 nil))) ; use file history
      (inCore
       (setq |$internalHistoryTable| nil))
```

```

(cond
  ((nequal |$IOindex| 0)
    (setq l (length (rkeyids (|histFileName|))))
    (do ((i 1 (1+ i)))
      ((> i l) nil)
      (setq vec (unwind-protect (|readHiFi| i) (|disableHist|)))
      (setq |$internalHistoryTable|
        (cons (cons i vec) |$internalHistoryTable|))
      (|histFileErase| (|histFileName|))))
  (setq |$useInternalHistoryTable| t)
  (|sayKeyedMsg| 'S2IH0032 nil)) ; use memory history
(t
  (setq |$HiFiAccess| nil)
  (|histFileErase| (|histFileName|))
  (setq str
    (rdefiostream
      (cons
        '(mode . output)
        (cons
          (cons 'file (|histFileName|))
          nil))))
  (do ((tmp0 (reverse |$internalHistoryTable|) (cdr tmp0))
      (tmp1 nil))
    ((or (atom tmp0)
      (progn
        (setq tmp1 (car tmp0))
        nil)
      (progn
        (progn
          (setq n (car tmp1))
          (setq rec (cdr tmp1))
          tmp1)
        nil))
      nil)
    (spadrwrite (|object2Identifier| n) rec str))
  (rshut str)
  (setq |$HiFiAccess| t)
  (setq |$internalHistoryTable| nil)
  (setq |$useInternalHistoryTable| nil)
  (|sayKeyedMsg| 's2ih0031 nil)))) ; use file history

```

34.4.10 defvar \$underbar

Also used in the output routines.

— initvars —

```
(defvar underbar "_")
```

34.4.11 defun writeInputLines

```
[sayKeyedMsg p331]
[throwKeyedMsg p??]
[size p1001]
[spaddifference p??]
[concat p1003]
[substring p??]
[readHiFi p579]
[histInputFileName p558]
[histFileErase p596]
[defiostream p938]
[nequal p??]
[namestring p996]
[shut p938]
[underbar p564]
[$HiFiAccess p710]
[$IOindex p??]
```

— defun writeInputLines —

```
(defun |writeInputLines| (fn initial)
  (let (maxn breakChars vec1 k svec done n lineList file inp)
    (declare (special underbar |$HiFiAccess| |$IOindex|))
    (cond
      ((null |$HiFiAccess|) (|sayKeyedMsg| 's2ih0013 nil)) ; history is not on
      ((null fn) (|throwKeyedMsg| 's2ih0038 nil)) ; missing file name
      (t
       (setq maxn 72)
       (setq breakChars (cons ' | (cons '+ nil)))
       (do ((tmp0 (spaddifference |$IOindex| 1))
           (i initial (+ i 1)))
           ((> i tmp0) nil)
           (setq vec1 (car (|readHiFi| i)))
           (when (stringp vec1) (setq vec1 (cons vec1 nil)))
           (dolist (vec vec1)
             (setq n (size vec))
             (do ()
                 ((null (> n maxn)) nil)
                 (setq done nil)
                 (do ((j 1 (1+ j)))
                     ((or (> j maxn) (null (null done))) nil))
```

```

      (setq k (spaddifference (1+ maxn) j))
      (when (member (elt vec k) breakChars)
        (setq svec (concat (substring vec 0 (1+ k)) underbar))
        (setq lineList (cons svec lineList))
        (setq done t)
        (setq vec (substring vec (1+ k) nil))
        (setq n (size vec))))
      (when done (setq n 0))
      (setq lineList (cons vec lineList))))
(setq file (|histInputFileName| fn))
(|histFileErase| file)
(setq inp
  (defiostream
    (cons
      '(mode . output)
      (cons (cons 'file file) nil)) 255 0))
(dolist (x (|removeUndoLines| (nreverse lineList)))
  (write-line x inp))
(cond
  ((nequal fn '|redo|)
   (|sayKeyedMsg| 's2ih0014 ; edit this file to see input lines
    (list (|namestring| file)))))
(shut inp)
nil)))

```

34.4.12 defun resetInCoreHist

```

[$HistListAct p??]
[$HistListLen p??]
[$HistList p??]

```

— defun resetInCoreHist —

```

(defun |resetInCoreHist| ()
  (declare (special |$HistListAct| |$HistListLen| |$HistList|))
  (setq |$HistListAct| 0)
  (do ((i 1 (1+ i)))
    ((> i |$HistListLen|) nil)
    (setq |$HistList| (cdr |$HistList|))
    (rplaca |$HistList| nil)))

```

34.4.13 defun changeHistListLen

```
[sayKeyedMsg p331]
[spaddifference p??]
[$HistListLen p??]
[$HistList p??]
[$HistListAct p??]
```

— **defun changeHistListLen** —

```
(defun |changeHistListLen| (n)
  (let (dif 1)
    (declare (special |$HistListLen| |$HistList| |$HistListAct|))
    (if (null (integerp n))
      (|sayKeyedMsg| 's2ih0015 (list n)) ; only positive integers
      (progn
        (setq dif (spaddifference n |$HistListLen|))
        (setq |$HistListLen| n)
        (setq l (cdr |$HistList|))
        (cond
          ((> dif 0)
            (do ((i 1 (1+ i)))
              ((> i dif) nil)
              (setq l (cons nil l))))
          ((minusp dif)
            (do ((tmp0 (spaddifference dif))
              ((i 1 (1+ i)))
              ((> i tmp0) nil)
              (setq l (cdr l)))
              (cond
                ((> |$HistListAct| n) (setq |$HistListAct| n))
                (t nil))))
            (rplacd |$HistList| l)
            '|done|))))))
```

34.4.14 defun updateHist

```
[startTimingProcess p??]
[updateInCoreHist p568]
[writeHiFi p580]
[disableHist p581]
[updateCurrentInterpreterFrame p537]
[stopTimingProcess p??]
[$IOindex p??]
[$HiFiAccess p710]
```



```
[$HistRecord p??]
[$mkTestInputStack p??]
[$currentLine p??]
```

— **defun updateHist** —

```
(defun |updateHist| ()
  (declare (special |$IOindex| |$HiFiAccess| |$HistRecord| |$mkTestInputStack|
    |$currentLine|))
  (when |$IOindex|
    (|startTimingProcess| '|history|)
    (|updateInCoreHist|)
    (when |$HiFiAccess|
      (unwind-protect (|writeHiFi|) (|disableHist|))
      (setq |$HistRecord| nil))
    (incf |$IOindex|)
    (|updateCurrentInterpreterFrame|)
    (setq |$mkTestInputStack| nil)
    (setq |$currentLine| nil)
    (|stopTimingProcess| '|history|)))
```

34.4.15 defun updateInCoreHist

```
[$HistList p??]
[$HistListLen p??]
[$HistListAct p??]
```

— **defun updateInCoreHist** —

```
(defun |updateInCoreHist| ()
  (declare (special |$HistList| |$HistListLen| |$HistListAct|))
  (setq |$HistList| (cdr |$HistList|))
  (rplaca |$HistList| nil)
  (when (> |$HistListLen| |$HistListAct|)
    (setq |$HistListAct| (1+ |$HistListAct|))))
```

34.4.16 defun putHist

```
[recordOldValue p570]
[get p??]
[recordNewValue p569]
```

```
[putIntSymTab p??]
[$HiFiAccess p710]
```

— defun putHist —

```
(defun |putHist| (x prop val e)
  (declare (special |$HiFiAccess|))
  (when (null (eq x '%)) (|recordOldValue| x prop (|get| x prop e)))
  (when |$HiFiAccess| (|recordNewValue| x prop val))
  (|putIntSymTab| x prop val e))
```

—————

34.4.17 defun recordNewValue

```
[startTimingProcess p??]
[recordNewValue0 p569]
[stopTimingProcess p??]
```

— defun recordNewValue —

```
(defun |recordNewValue| (x prop val)
  (|startTimingProcess| '|history|)
  (|recordNewValue0| x prop val)
  (|stopTimingProcess| '|history|))
```

—————

34.4.18 defun recordNewValue0

```
[assq p1006]
[$HistRecord p??]
```

— defun recordNewValue0 —

```
(defun |recordNewValue0| (x prop val)
  (let (p1 p2 p)
    (declare (special |$HistRecord|))
    (if (setq p1 (assq x |$HistRecord|))
      (if (setq p2 (assq prop (cdr p1)))
        (rplacd p2 val)
        (rplacd p1 (cons (cons prop val) (cdr p1))))
      (progn
        (setq p (cons x (list (cons prop val))))
```

```
(setq |$HistRecord| (cons p |$HistRecord|))))))
```

34.4.19 defun recordOldValue

```
[startTimingProcess p??]
[recordOldValue0 p570]
[stopTimingProcess p??]
[assq p1006]
```

— defun recordOldValue —

```
(defun |recordOldValue| (x prop val)
  (|startTimingProcess| '|history|)
  (|recordOldValue0| x prop val)
  (|stopTimingProcess| '|history|))
```

34.4.20 defun recordOldValue0

```
[$HistList p??]
```

— defun recordOldValue0 —

```
(defun |recordOldValue0| (x prop val)
  (let (p1 p)
    (declare (special |$HistList|))
    (when (setq p1 (assq x (car |$HistList|)))
      (when (null (assq prop (cdr p1)))
        (rplacd p1 (cons (cons prop val) (cdr p1)))))
    (setq p (cons x (list (cons prop val))))
    (rplaca |$HistList| (cons p (car |$HistList|)))))
```

34.4.21 defun undoInCore

```
[undoChanges p571]
[spaddifference p??]
[readHiFi p579]
```

```
[disableHist p581]
[assq p1006]
[sayKeyedMsg p331]
[putHist p568]
[updateHist p567]
[$HistList p??]
[$HistListLen p??]
[$IOindex p??]
[$HiFiAccess p710]
[$InteractiveFrame p??]
```

— **defun undoInCore** —

```
(defun |undoInCore| (n)
  (let (li vec p p1 val)
    (declare (special |$HistList| |$HistListLen| |$IOindex| |$HiFiAccess|
                      |$InteractiveFrame|))
    (setq li |$HistList|)
    (do ((i n (+ i 1)))
        ((> i |$HistListLen|) nil)
      (setq li (cdr li)))
    (|undoChanges| li)
    (setq n (spaddifference (spaddifference |$IOindex| n) 1))
    (and
      (> n 0)
      (if |$HiFiAccess|
        (progn
          (setq vec (cdr (unwind-protect (|readHiFi| n) (|disableHist|))))
          (setq val
            (and
              (setq p (assq '% vec))
              (setq p1 (assq '|value| (cdr p)))
              (cdr p1))))
          (|sayKeyedMsg| 's2ih0019 (cons n nil)))) ; no history file
      (setq |$InteractiveFrame| (|putHist| '% '|value| val |$InteractiveFrame|))
      (|updateHist|)))
```

34.4.22 defun undoChanges

```
[boot-equal p??]
[undoChanges p571]
[putHist p568]
[$HistList p??]
[$InteractiveFrame p??]
```

— **defun undoChanges** —

```
(defun |undoChanges| (li)
  (let (x)
    (declare (special |$HistList| |$InteractiveFrame|))
    (when (null (boot-equal (cdr li) |$HistList|)) (|undoChanges| (cdr li)))
    (dolist (p1 (car li))
      (setq x (car p1))
      (dolist (p2 (cdr p1))
        (|putHist| x (car p2) (cdr p2) |$InteractiveFrame|))))))
```

34.4.23 defun undoFromFile

```
[seq p??]
[exit p??]
[recordOldValue p570]
[recordNewValue p569]
[readHiFi p579]
[disableHist p581]
[putHist p568]
[assq p1006]
[updateHist p567]
[$InteractiveFrame p??]
[$HiFiAccess p710]
```

— **defun undoFromFile** —

```
(defun |undoFromFile| (n)
  (let (var1 prop vec x p p1 val)
    (declare (special |$InteractiveFrame| |$HiFiAccess|))
    (do ((tmp0 (caar |$InteractiveFrame|) (cdr tmp0)) (tmp1 nil))
      ((or (atom tmp0)
            (progn (setq tmp1 (car tmp0)) nil)
            (progn
              (progn
                (setq x (car tmp1))
                (setq var1 (cdr tmp1))
                tmp1)
              nil))
          nil)
      (seq
        (exit
          (do ((tmp2 var1 (cdr tmp2)) (p nil))
```

```

      ((or (atom tmp2) (progn (setq p (car tmp2)) nil)) nil)
    (seq
      (exit
        (progn
          (setq prop (car p))
          (setq val (cdr p))
          (when val
            (progn
              (when (null (eq x '%))
                (|recordOldValue| x prop val))
              (when (|$HiFiAccess|
                (|recordNewValue| x prop val))
                (rplacd p nil))))))))))
  (do ((i 1 (1+ i)))
    ((> i n) nil)
    (setq vec
      (unwind-protect (cdr (|readHiFi| i)) (|disableHist|)))
    (do ((tmp3 vec (cdr tmp3)) (p1 nil))
      ((or (atom tmp3) (progn (setq p1 (car tmp3)) nil)) nil)
      (setq x (car p1))
      (do ((tmp4 (cdr p1) (cdr tmp4)) (p2 nil))
        ((or (atom tmp4) (progn (setq p2 (car tmp4)) nil)) nil)
        (setq (|$InteractiveFrame|
          (|putHist| x (car p2) (CDR p2) (|$InteractiveFrame|))))
      (setq val
        (and
          (setq p (assq '% vec))
          (setq p1 (assq '|value| (cdr p)))
          (cdr p1)))
      (setq (|$InteractiveFrame| (|putHist| '% '|value| val (|$InteractiveFrame|))
        (|updateHist|)))

```

34.4.24 defun saveHistory

```

[sayKeyedMsg p331]
[makeInputFilename p939]
[histFileName p558]
[throwKeyedMsg p??]
[makeHistFileName p557]
[histInputFileName p558]
[writeInputLines p565]
[histFileErase p596]
[rdefiostream p??]
[spadrwrite0 p582]
[object2Identifier p??]

```

```
[rshut p??]
[namestring p996]
[$seen p??]
[$HiFiAccess p710]
[$useInternalHistoryTable p557]
[$internalHistoryTable p??]
```

— **defun saveHistory** —

```
(defun |saveHistory| (fn)
  (let (|$seen| savefile inputfile saveStr n rec val)
    (declare (special |$seen| |$HiFiAccess| |$useInternalHistoryTable|
                      |$internalHistoryTable|))
    (setq |$seen| (make-hash-table :test #'eq))
    (cond
      ((null |$HiFiAccess|)
       (|sayKeyedMsg| 's2ih0016 nil)) ; the history file is not on
      ((and (null |$useInternalHistoryTable|)
            (null (makeInputFilename (|histFileName|))))
       (|sayKeyedMsg| 's2ih0022 nil)) ; no history saved yet
      ((null fn)
       (|throwKeyedMsg| 's2ih0037 nil)) ; need to specify a history filename
      (t
       (setq savefile (|makeHistFileName| fn))
       (setq inputfile (|histInputFileName| fn))
       (|writeInputLines| fn 1)
       (|histFileErase| savefile)
       (when |$useInternalHistoryTable|
        (setq saveStr
              (rdefiostream
               (cons '(mode . output)
                     (cons (cons 'file savefile) nil))))
        (do ((tmp0 (reverse |$internalHistoryTable|) (cdr tmp0))
            (tmp1 nil))
            ((or (atom tmp0)
                 (progn (setq tmp1 (car tmp0)) nil)
                 (progn
                  (progn
                   (setq n (car tmp1))
                   (setq rec (cdr tmp1))
                   tmp1)
                  nil))
             nil)
          (setq val (spadrwrite0 (|object2Identifier| n) rec saveStr))
          (when (eq val '|writifyFailed|)
            (|sayKeyedMsg| 's2ih0035 ; can't save the value of step
                          (list n inputfile))))
        (rshut saveStr))
       (|sayKeyedMsg| 's2ih0018 ; saved history file is
```

```
(cons (|namestring| savefile) nil))
nil)))
```

34.4.25 defun restoreHistory

```
[pairp p??]
[qcdr p??]
[qcar p??]
[identp p1003]
[throwKeyedMsg p??]
[makeHistFileName p557]
[putHist p568]
[makeInputFilename p939]
[sayKeyedMsg p331]
[namestring p996]
[clearSpad2Cmd p480]
[histFileName p558]
[histFileErase p596]
[$fcopy p??]
[rkeyids p??]
[readHiFi p579]
[disableHist p581]
[updateInCoreHist p568]
[get p??]
[rempropI p??]
[clearCmdSortedCaches p481]
[$options p??]
[$internalHistoryTable p??]
[$HiFiAccess p710]
[$e p??]
[$useInternalHistoryTable p557]
[$InteractiveFrame p??]
[$oldHistoryFileName p556]
```

— defun restoreHistory —

```
(defun |restoreHistory| (fn)
  (let (|$options| fnq restfile curfile l oldInternal vec line x a)
    (declare (special |$options| |$internalHistoryTable| |$HiFiAccess| |$e|
      |$useInternalHistoryTable| |$InteractiveFrame| |$oldHistoryFileName|))
    (cond
      ((null fn) (setq fnq |$oldHistoryFileName|))
      ((and (pairp fn)
```



```

      (eq (qcdr fn) nil)
      (progn
        (setq fnq (qcar fn))
        t)
      (identp fnq))
    (setq fnq fnq))
  (t (|throwKeyedMsg| 's2ih0023 (cons fnq nil))) ; invalid filename
(setq restfile (|makeHistFileName| fnq))
(if (null (makeInputFilename restfile))
    (|sayKeyedMsg| 's2ih0024 ; file does not exist
      (cons (|namestring| restfile) nil))
    (progn
      (setq |$options| nil)
      (|clearSpad2Cmd| '(|all|))
      (setq curfile (|histFileName|))
      (|histFileErase| curfile)
      ($fcopy restfile curfile)
      (setq l (length (rkeyids curfile)))
      (setq |$HiFiAccess| t)
      (setq oldInternal |$useInternalHistoryTable|)
      (setq |$useInternalHistoryTable| nil)
      (when oldInternal (setq |$internalHistoryTable| nil))
      (do ((i 1 (1+ i)))
          ((> i l) nil)
          (setq vec (unwind-protect (|readHiFi| i) (|disableHist|)))
          (when oldInternal
            (setq |$internalHistoryTable|
              (cons (cons i vec) |$internalHistoryTable|)))
            (setq line (car vec))
            (dolist (p1 (cdr vec))
              (setq x (car p1))
              (do ((tmp1 (cdr p1) (cdr tmp1)) (p2 nil))
                  ((or (atom tmp1) (progn (setq p2 (car tmp1)) nil)) nil)
                  (setq |$InteractiveFrame|
                    (|putHist| x
                      (car p2) (cdr p2) |$InteractiveFrame|)))
                (|updateInCoreHist|))
              (setq |$e| |$InteractiveFrame|)
              (do ((tmp2 (caar |$InteractiveFrame|) (cdr tmp2)) (tmp3 nil))
                  ((or (atom tmp2)
                      (progn
                        (setq tmp3 (car tmp2))
                        nil)
                      (progn
                        (setq a (car tmp3))
                        tmp3
                        nil))
                      nil)
                  (when (|get| a '||localModemap| |$InteractiveFrame|)

```

```

(|rempropI| a '|localModemap|)
(|rempropI| a '|localVars|)
(|rempropI| a '|mapBody|)))
(setq |$IOindex| (1+ 1))
(setq |$useInternalHistoryTable| oldInternal)
(|sayKeyedMsg| 'S2IH0025 ; workspace restored
  (cons (|namestring| restfile) nil))
(|clearCmdSortedCaches|
  nil)))

```

34.4.26 defun setIOindex

[*\$IOindex* *p??*]

— defun setIOindex —

```

(defun |setIOindex| (n)
  (declare (special |$IOindex|))
  (setq |$IOindex| n))

```

34.4.27 defun showInput

[*tab* *p??*]
 [*readHiFi* *p579*]
 [*disableHist* *p581*]
 [*sayMSG* *p333*]

— defun showInput —

```

(defun |showInput| (mini maxi)
  (let (vec l)
    (do ((|ind| mini (+ |ind| 1)))
      ((> |ind| maxi) nil)
      (setq vec (unwind-protect (|readHiFi| |ind|) (|disableHist|)))
      (cond
        ((> 10 |ind|) (tab 2))
        ((> 100 |ind|) (tab 1))
        (t nil))
      (setq l (car vec))
      (if (stringp l)
          (|sayMSG| (list "    [" |ind| "]" " (car vec)))))

```

```
(progn
  (|sayMSG| (list "    [" |ind| "]" "))
  (do ((tmp0 1 (cdr tmp0)) (ln nil))
    ((or (atom tmp0) (progn (setq ln (car tmp0)) nil)) nil)
    (|sayMSG| (list "        " ln))))))
```

34.4.28 defun showInOut

```
[assq p1006]
[spadPrint p??]
[objValUnwrap p??]
[objMode p??]
[readHiFi p579]
[disableHist p581]
[sayMSG p333]
```

— defun showInOut —

```
(defun |showInOut| (mini maxi)
  (let (vec Alist triple)
    (do ((ind mini (+ ind 1)))
      ((> ind maxi) nil)
      (setq vec (unwind-protect (|readHiFi| ind) (|disableHist|)))
      (|sayMSG| (cons (car vec) nil))
      (cond
        ((setq Alist (assq '% (cdr vec)))
         (setq triple (cdr (assq 'value| (cdr Alist))))
         (setq |$IOindex| ind)
         (|spadPrint| (|objValUnwrap| triple) (|objMode| triple)))))))
```

34.4.29 defun fetchOutput

```
[boot-equal p??]
[spaddifference p??]
[getI p??]
[throwKeyedMsg p??]
[readHiFi p579]
[disableHist p581]
[assq p1006]
```

— defun fetchOutput —

```

(defun |fetchOutput| (n)
  (let (vec Alist val)
    (cond
      ((and (boot-equal n (spaddifference 1)) (setq val (|getI| '% '|value|))))
      val)
    (|$HiFiAccess|
      (setq n
        (cond
          ((minusp n) (+ |$IOindex| n))
          (t n)))
        (cond
          ((>= n |$IOindex|)
            (|throwKeyedMsg| 'S2IH0001 (cons n nil))) ; no step n yet
          (> 1 n)
            (|throwKeyedMsg| 's2ih0002 (cons n nil))) ; only nonzero steps
          (t
            (setq vec (unwind-protect (|readHiFi| n) (|disableHist|)))
            (cond
              ((setq Alist (assq '% (cdr vec)))
                (cond
                  ((setq val (cdr (assq '|value| (cdr Alist))))
                    val)
                  (t
                     (|throwKeyedMsg| 's2ih0003 (cons n nil)))))) ; no step value
              (t (|throwKeyedMsg| 's2ih0003 (cons n nil)))))) ; no step value
            (t (|throwKeyedMsg| 's2ih0004 nil)))))) ; history not on

```

34.4.30 Read the history file using index n

```

[assoc p??]
[keyedSystemError p??]
[qcdr p??]
[rdefiostream p??]
[histFileName p558]
[spadrread p583]
[object2Identifier p??]
[rshut p??]
[$useInternalHistoryTable p557]
[$internalHistoryTable p??]

```

— defun readHiFi —

```

(defun |readHiFi| (n)
  "Read the history file using index n"
  (let (pair HiFi vec)

```

```

(declare (special |$useInternalHistoryTable| |$internalHistoryTable|))
(if |$useInternalHistoryTable|
  (progn
    (setq pair (|assoc| n |$internalHistoryTable|))
    (if (atom pair)
      (|keyedSystemError| 's2ih0034 nil) ; missing element
      (setq vec (qcdr pair))))
  (progn
    (setq HiFi
      (rdefiostream
        (cons
          '(mode . input)
          (cons
            (cons 'file (|histFileName|)) nil))))
    (setq vec (spadrread (|object2Identifier| n) HiFi))
    (rshut HiFi)))
vec))

```

34.4.31 Write information of the current step to history file

```

[rdefiostream p??]
[histFileName p558]
[spadrwrite p583]
[object2Identifier p??]
[rshut p??]
[$useInternalHistoryTable p557]
[$internalHistoryTable p??]
[$IOindex p??]
[$HistRecord p??]
[$currentLine p??]

```

— defun writeHiFi —

```

(defun |writeHiFi| ()
  "Writes information of the current step to history file"
  (let (HiFi)
    (declare (special |$useInternalHistoryTable| |$internalHistoryTable|
      |$IOindex| |$HistRecord| |$currentLine|))
    (if |$useInternalHistoryTable|
      (setq |$internalHistoryTable|
        (cons
          (cons |$IOindex|
            (cons |$currentLine| |$HistRecord|))
          |$internalHistoryTable|))
      (progn

```

```

(setq HiFi
  (rdefiostream
    (cons
      '(mode . output)
      (cons (cons 'file (|histFileName|)) nil))))
(spadrwrite (|object2Identifier| |$IOindex|)
  (cons |$currentLine| |$HistRecord|) HiFi)
(rshut HiFi))))

```

34.4.32 Disable history if an error occurred

[histFileErase p596]
 [histFileName p558]
 [\$HiFiAccess p710]

— defun disableHist —

```

(defun |disableHist| ()
  "Disable history if an error occurred"
  (declare (special |$HiFiAccess|))
  (cond
    ((null |$HiFiAccess|)
     (|histFileErase| (|histFileName|)))
    (t nil)))

```

34.4.33 defun writeHistModesAndValues

[get p??]
 [putHist p568]
 [\$InteractiveFrame p??]

— defun writeHistModesAndValues —

```

(defun |writeHistModesAndValues| ()
  (let (a x)
    (declare (special |$InteractiveFrame|))
    (do ((tmp0 (caar |$InteractiveFrame|) (cdr tmp0)) (tmp1 nil))
      ((or (atom tmp0)
           (progn
              (setq tmp1 (car tmp0))
              nil))

```

```

      (progn
        (progn
          (setq a (car tmp1))
          tmp1)
        nil))
    nil)
  (cond
    ((setq x (|get| a '|value| |$InteractiveFrame|))
     (|putHist| a '|value| x |$InteractiveFrame|))
    ((setq x (|get| a '|mode| |$InteractiveFrame|))
     (|putHist| a '|mode| x |$InteractiveFrame|))))))

```

34.5 Lisplib output transformations

Lisplib output transformations

Some types of objects cannot be saved by LISP/VM in lisplibs. These functions transform an object to a writable form and back.

34.5.1 defun spadrwrite0

```

[safeWritify p585]
[rwrite p582]

```

— defun spadrwrite0 —

```

(defun spadrwrite0 (vec item stream)
  (let (val)
    (setq val (|safeWritify| item))
    (if (eq val '|writifyFailed|)
        val
        (progn
          (|rwrite| vec val stream)
          item))))

```

34.5.2 defun Random write to a stream

```

[rwrite p582]
[pname p1001]
[identp p1003]

```

— defun rwrite —

```
(defun |rwrite| (key val stream)
  (when (identp key) (setq key (pname key)))
  (rwrite key val stream)))
```

—————

34.5.3 defun spadrwrite

```
[spadrwrite0 p582]
[throwKeyedMsg p??]
```

— defun spadrwrite —

```
(defun spadrwrite (vec item stream)
  (let (val)
    (setq val (spadrwrite0 vec item stream))
    (if (eq val '|writifyFailed|)
        (|throwKeyedMsg| 's2ih0036 nil) ; cannot save value to file
        item)))
```

—————

34.5.4 defun spadrread

```
[dewritify p593]
[rread p583]
```

— defun spadrread —

```
(defun spadrread (vec stream)
  (|dewritify| (|rread| vec stream nil)))
```

—————

34.5.5 defun Random read a key from a stream

```
RREAD takes erroval to return if key is missing [rread p583]
[identp p1003]
[pname p1001]
```

— defun rread —


```
(defun |rread| (key rstream errorval)
  (when (identp key) (setq key (pname key)))
  (rread key rstream errorval))
```

34.5.6 defun unwritable?

```
[pairp p??]
[vecp p??]
[placep p??]
```

— defun unwritable? —

```
(defun |unwritable?| (ob)
  (cond
    ((or (pairp ob) (vecp ob)) nil)
    ((or (compiled-function-p ob) (hash-table-p ob)) t)
    ((or (placep ob) (readtablep ob)) t)
    ((floatp ob) t)
    (t nil)))
```

34.5.7 defun writifyComplain

Create a full isomorphic object which can be saved in a lisplib. Note that `dewritify(writify(x))` preserves `UEQUALity` of hashtables. `HASHTABLEs` go both ways. `READTABLEs` cannot presently be transformed back. [sayKeyedMsg p331]
 [\$writifyComplained p??]

— defun writifyComplain —

```
(defun |writifyComplain| (s)
  (declare (special |$writifyComplained|))
  (unless |$writifyComplained|
    (setq |$writifyComplained| t)
    (|sayKeyedMsg| 's2ih0027 (list s)))) ; cannot save value
```

34.5.8 defun safeWritify

```
[writifyTag p??]
[writify p588]
```

— **defun safeWritify** —

```
(defun |safeWritify| (ob)
  (catch '|writifyTag| (|writify| ob)))
```

—————

34.5.9 defun writify,writifyInner

```
[writifyTag p??]
[seq p??]
[exit p??]
[hget p1000]
[pairst p??]
[qcar p??]
[qcdr p??]
[spadClosure? p589]
[writify,writifyInner p585]
[hput p1000]
[qrplaca p??]
[qrplacd p??]
[vecp p??]
[isDomainOrPackage p849]
[mkEvaluable p??]
[devaluate p??]
[qvmaxindex p??]
[qsetvelt p??]
[qvelt p??]
[constructor? p??]
[hkeys p1000]
[hashtable-class p??]
[placep p??]
[boot-equal p??]
[$seen p??]
[$NonNullStream p589]
[$NullStream p590]
```

— **defun writify,writifyInner** —

```
(defun |writify,writifyInner| (ob)
```

```

(prog (e name tmp1 tmp2 tmp3 x qcar qcdr d n keys nob)
  (declare (special |$seen| |$NonNullStream| |$NullStream|))
  (return
    (seq
      (when (null ob) (exit nil))
      (when (setq e (hget |$seen| ob)) (exit e))
      (when (pairp ob)
        (exit
          (seq
            (setq qcar (qcar ob))
            (setq qcdr (qcdr ob))
            (when (setq name (|spadClosure?| ob))
              (exit
                (seq
                  (setq d (|writify,writifyInner| (qcdr ob)))
                  (setq nob
                    (cons 'writified!!
                      (cons 'spadclosure
                        (cons d (cons name nil))))))
                  (hput |$seen| ob nob)
                  (hput |$seen| nob nob)
                  (exit nob))))
              (when
                (and
                  (and (pairp ob)
                    (eq (qcar ob) 'lambda-closure)
                    (progn
                      (setq tmp1 (qcdr ob))
                      (and (pairp tmp1)
                        (progn
                          (setq tmp2 (qcdr tmp1))
                          (and
                            (pairp tmp2)
                            (progn
                              (setq tmp3 (qcdr tmp2))
                              (and (pairp tmp3)
                                (progn
                                  (setq x (qcar tmp3))
                                  t)))))))))) x)
                  (exit
                    (throw '|writifyTag| '|writifyFailed|)))
                (setq nob (cons qcar qcdr))
                (hput |$seen| ob nob)
                (hput |$seen| nob nob)
                (setq qcar (|writify,writifyInner| qcar))
                (setq qcdr (|writify,writifyInner| qcdr))
                (qrplaca nob qcar)
                (qrplacd nob qcdr)
                (exit nob))))
            (when (vecp ob)

```

```

(exit
  (seq
    (when (|isDomainOrPackage| ob)
      (setq d (|mkEvalable| (|devalue| ob)))
      (setq nob (list 'writified!! 'devaluated (|writify,writifyInner| d)))
      (hput |$seen| ob nob)
      (hput |$seen| nob nob)
      (exit nob))
    (setq n (qvmaxindex ob))
    (setq nob (make-array (1+ n)))
    (hput |$seen| ob nob)
    (hput |$seen| nob nob)
    (do ((i 0 (=! i)))
      (> i n) nil)
      (qsetvelt nob i (|writify,writifyInner| (qvelt ob i)))
    (exit nob))))
(when (eq ob 'writified!!)
  (exit
    (cons 'writified!! (cons 'self nil))))
(when (|constructor?| ob)
  (exit ob))
(when (compiled-function-p ob)
  (exit
    (throw 'writifyTag| 'writifyFailed|)))
(when (hash-table-p ob)
  (setq nob (cons 'writified!! nil))
  (hput |$seen| ob nob)
  (hput |$seen| nob nob)
  (setq keys (hkeys ob))
  (qrplacd nob
    (cons
      'hashtable
      (cons
        (hashtable-class ob)
        (cons
          (|writify,writifyInner| keys)
          (cons
            (prog (tmp0)
              (setq tmp0 nil)
              (return
                (do ((tmp1 keys (cdr tmp1)) (k nil))
                  ((or (atom tmp1)
                      (progn
                        (setq k (car tmp1))
                        nil))
                    (nreverse0 tmp0))
              (setq tmp0
                (cons (|writify,writifyInner| (hget ob k)) tmp0))))
            nil))))))
    (exit nob))

```

```

(when (placep ob)
  (setq nob (cons 'writified!! (cons 'place nil)))
  (hput |$seen| ob nob)
  (hput |$seen| nob nob)
  (exit nob))
(when (readtablep ob)
  (exit
   (throw '|writifyTag| '|writifyFailed|)))
(when (stringp ob)
  (exit
   (seq
    (when (eq ob |$NullStream|)
      (exit
       (cons 'writified!! (cons 'nullstream nil)))))
    (when (eq ob |$NonNullStream|)
      (exit
       (cons 'writified!! (cons 'nonnullstream nil)))))
    (exit ob))))
(when (floatp ob)
  (exit
   (seq
    (when (boot-equal ob (read-from-string (princ-to-string ob)))
      (exit ob))
    (exit
     (cons 'writified!!
      (cons 'float
       (cons ob
        (multiple-value-list (integer-decode-float ob))))))))))
(exit ob))))

```

—————

34.5.10 defun writify

[ScanOrPairVec p594]
 [function p??]
 [writify,writifyInner p585]
 [\$seen p??]
 [\$writifyComplained p??]

— defun writify —

```

(defun |writify| (ob)
  (let (|$seen| |$writifyComplained|)
    (declare (special |$seen| |$writifyComplained|))
    (if (null (|ScanOrPairVec| (|function| |unwritable?|) ob))
        ob

```

```
(progn
  (setq |$seen| (make-hash-table :test #'eq))
  (setq |$writifyComplained| nil)
  (|writify,writifyInner| ob))))
```

34.5.11 defun spadClosure?

```
[qcar p??]
[bpname p??]
[qcdr p??]
[vecp p??]
```

— defun spadClosure? —

```
(defun |spadClosure?| (ob)
  (let (fun name vec)
    (setq fun (qcar ob))
    (if (null (setq name (bpname fun)))
        nil
        (progn
          (setq vec (qcdr ob))
          (if (null (vecp vec))
              nil
              name))))))
```

34.5.12 defun dewritify,is?

— defun dewritify,is? —

```
(defun |dewritify,is?| (a)
  (eq a 'writified!))
```

34.5.13 defvar \$NonNullStream

— initvars —

```
(defvar |$NonNullStream| "NonNullStream")
```

34.5.14 defvar \$NullStream

— initvars —

```
(defvar |$NullStream| "NullStream")
```

34.5.15 defun dewritify,dewritifyInner

```
[seq p??]
[exit p??]
[hget p1000]
[pairp p??]
[intp p??]
[gensymmer p??]
[error p??]
[poundsign p??]
[nequal p??]
[hput p1000]
[dewritify,dewritifyInner p590]
[concat p1003]
[vmread p??]
[make-instream p937]
[spaddifference p??]
[qcar p??]
[qcdr p??]
[qrplaca p??]
[qrplacd p??]
[vecp p??]
[qvmaxindex p??]
[qsetvelt p??]
[qvelt p??]
[$seen p??]
[$NullStream p590]
[$NonNullStream p589]
```

— defun dewritify,dewritifyInner —

```

(defun |dewritify,dewritifyInner| (ob)
  (prog (e type oname f vec name tmp1 signif expon sign fval qcar qcdr n nob)
    (declare (special |$seen| |$NullStream| |$NonNullStream|))
    (return
      (seq
        (when (null ob)
          (exit nil))
        (when (setq e (hget |$seen| ob))
          (exit e))
        (when (and (pairp ob) (eq (car ob) 'writified!!))
          (exit
            (seq
              (setq type (elt ob 1))
              (when (eq type 'self)
                (exit 'writified!!))
              (when (eq type 'bpi)
                (exit
                  (seq
                    (setq oname (elt ob 2))
                    (setq f
                      (seq
                        (when (integerp oname) (exit (eval (gensymmer oname))))
                        (exit (symbol-function oname))))
                    (when (null (compiled-function-p f))
                      (exit (|error| "A required BPI does not exist.")))
                    (when (and (> (|#| ob) 3) (nequal (sxhash f) (elt ob 3)))
                      (exit (|error| "A required BPI has been redefined.")))
                    (hput |$seen| ob f)
                    (exit f))))
                (when (eq type 'hashtable)
                  (exit
                    (seq
                      (setq nob (make-hash-table :test #'equal))
                      (hput |$seen| ob nob)
                      (hput |$seen| nob nob)
                      (do ((tmp0 (elt ob 3) (cdr tmp0))
                          (k nil)
                          (tmp1 (elt ob 4) (cdr tmp1))
                          (e nil))
                        ((or (atom tmp0)
                            (progn
                              (setq k (car tmp0))
                              nil)
                              (atom tmp1)
                              (progn
                                (setq e (car tmp1))
                                nil))
                             nil)
                          (seq
                            (exit

```



```

      (hput nob (|dewritify,dewritifyInner| k)
        (|dewritify,dewritifyInner| e))))
    (exit nob)))
(when (eq type 'devaluated)
  (exit
    (seq
      (setq nob (eval (|dewritify,dewritifyInner| (elt ob 2))))
      (hput |$seen| ob nob)
      (hput |$seen| nob nob)
      (exit nob))))
(when (eq type 'spadclosure)
  (exit
    (seq
      (setq vec (|dewritify,dewritifyInner| (elt ob 2)))
      (setq name (ELT ob 3))
      (when (null (fboundp name))
        (exit
          (|error|
            (concat "undefined function: " (symbol-name name))))))
      (setq nob (cons (symbol-function name) vec))
      (hput |$seen| ob nob)
      (hput |$seen| nob nob)
      (exit nob))))
(when (eq type 'place)
  (exit
    (seq
      (setq nob (vmread (make-instream nil)))
      (hput |$seen| ob nob)
      (hput |$seen| nob nob)
      (exit nob))))
(when (eq type 'readtable)
  (exit (|error| "Cannot de-writify a read table.")))
(when (eq type 'nullstream)
  (exit |$NullStream|))
(when (eq type 'nonnullstream)
  (exit |$NonNullStream|))
(when (eq type 'float)
  (exit
    (seq
      (progn
        (setq tmp1 (cddr ob))
        (setq fval (car tmp1))
        (setq signif (cadr tmp1))
        (setq expon (caddr tmp1))
        (setq sign (caddr tmp1))
        tmp1)
      (setq fval (scale-float (float signif fval) expon))
      (when (minusp sign)
        (exit (spaddifference fval)))
      (exit fval))))

```

```

      (exit (|error| "Unknown type to de-writify."))))
    (when (pairp ob)
      (exit
        (seq
          (setq qcar (qcar ob))
          (setq qcdr (qcdr ob))
          (setq nob (cons qcar qcdr))
          (hput |$seen| ob nob)
          (hput |$seen| nob nob)
          (qrplaca nob (|dewritify,dewritifyInner| qcar))
          (qrplacd nob (|dewritify,dewritifyInner| qcdr))
          (exit nob))))
    (when (vecp ob)
      (exit
        (seq
          (setq n (qvmaxindex ob))
          (setq nob (make-array (1+ n)))
          (hput |$seen| ob nob)
          (hput |$seen| nob nob)
          (do ((i 0 (1+ i)))
              ((> i n) nil)
            (seq
              (exit
                (qsetvelt nob i
                  (|dewritify,dewritifyInner| (qvelt ob i))))))
          (exit nob))))
    (exit ob))))

```

34.5.16 defun dewritify

```

[ScanOrPairVec p594]
[function p??]
[dewritify,dewritifyInner p590]
[$seen p??]

```

— defun dewritify —

```

(defun |dewritify| (ob)
  (let (|$seen|)
    (declare (special |$seen|))
    (if (null (|ScanOrPairVec| (|function| |dewritify,is?|) ob))
        ob
      (progn
        (setq |$seen| (make-hash-table :test #'eq))
        (|dewritify,dewritifyInner| ob))))

```

34.5.17 defun ScanOrPairVec,ScanOrInner

```
[ScanOrPairVecAnswer p??]
[hget p1000]
[pairp p??]
[hput p1000]
[ScanOrPairVec,ScanOrInner p594]
[qcar p??]
[qcdr p??]
[vecp p??]
[$seen p??]
```

— defun ScanOrPairVec,ScanOrInner —

```
(defun |ScanOrPairVec,ScanOrInner| (f ob)
  (declare (special |$seen|))
  (when (hget |$seen| ob) nil)
  (when (pairp ob)
    (hput |$seen| ob t)
    (|ScanOrPairVec,ScanOrInner| f (qcar ob))
    (|ScanOrPairVec,ScanOrInner| f (qcdr ob)))
  (when (vecp ob)
    (hput |$seen| ob t)
    (do ((tmp0 (spaddifference (|#| ob) 1)) (i 0 (1+ i)))
        ((> i tmp0) nil)
        (|ScanOrPairVec,ScanOrInner| f (elt ob i))))
  (when (funcall f ob) (throw '|ScanOrPairVecAnswer| t))
  nil)
```

34.5.18 defun ScanOrPairVec

```
[ScanOrPairVecAnswer p??]
[ScanOrPairVec,ScanOrInner p594]
[$seen p??]
```

— defun ScanOrPairVec —

```
(defun |ScanOrPairVec| (f ob)
  (let (|$seen|)
```

```
(declare (special |$seen|))
(setq |$seen| (make-hash-table :test #'eq))
(catch '|ScanOrPairVecAnswer| (|ScanOrPairVec,ScanOrInner| f ob)))
```

34.5.19 defun gensymInt

```
[gensymp p??]
[error p??]
[pname p1001]
[charDigitVal p595]
```

— defun gensymInt —

```
(defun |gensymInt| (g)
  (let (p n)
    (if (null (gensymp g))
        (|error| "Need a GENSYM")
        (progn
          (setq p (pname g))
          (setq n 0)
          (do ((tmp0 (spaddifference (|#| p) 1)) (i 2 (1+ i)))
              ((> i tmp0) nil)
              (setq n (+ (* 10 n) (|charDigitVal| (elt p i))))
              n))))))
```

34.5.20 defun charDigitVal

```
[spaddifference p??]
[error p??]
```

— defun charDigitVal —

```
(defun |charDigitVal| (c)
  (let (digits n)
    (setq digits "0123456789")
    (setq n (spaddifference 1))
    (do ((tmp0 (spaddifference (|#| digits) 1)) (i 0 (1+ i)))
        ((or (> i tmp0) (null (minusp n))) nil)
        (if (char= c (elt digits i))
            (setq n i)
            nil)))
```

```
(if (minusp n)
  (|error| "Character is not a digit")
  n)))
```

34.5.21 defun histFileErase

— defun histFileErase —

```
(defun |histFileErase| (file)
  (when (probe-file file) (delete-file file)))
```

34.6 History File Messages

— History File Messages —

```
S2IH0001
  You have not reached step %1b yet, and so its value cannot be
  supplied.
S2IH0002
  Cannot supply value for step %1b because 1 is the first step.
S2IH0003
  Step %1b has no value.
S2IH0004
  The history facility is not on, so you cannot use %b %% %d .
S2IH0006
  You have not used the correct syntax for the %b history %d command.
  Issue %b )help history %d for more information.
S2IH0007
  The history facility is already on.
S2IH0008
  The history facility is now on.
S2IH0009
  Turning on the history facility will clear the contents of the
  workspace.
  Please enter %b y %d or %b yes %d if you really want to do this:
S2IH0010
  The history facility is still off.
S2IH0011
```

The history facility is already off.
S2IH0012
The history facility is now off.
S2IH0013
The history facility is not on, so the .input file containing your user input cannot be created.
S2IH0014
Edit %b %1 %d to see the saved input lines.
S2IH0015
The argument %b n %d for %b)history)change n must be a nonnegative integer and your argument, %1b , is not one.
S2IH0016
The history facility is not on, so no information can be saved.
S2IH0018
The saved history file is %1b .
S2IH0019
There is no history file, so value of step %1b is undefined.
S2IH0022
No history information had been saved yet.
S2IH0023
%1b is not a valid filename for the history file.
S2IH0024
History information cannot be restored from %1b because the file does not exist.
S2IH0025
The workspace has been successfully restored from the history file %1b .
S2IH0026
The history facility command %1b cannot be performed because the history facility is not on.
S2IH0027
A value containing a %1b is being saved in a history file or a compiled input file INLIB. This type is not yet usable in other history operations. You might want to issue %b)history)off %d
S2IH0029
History information is already being maintained in an external file (and not in memory).
S2IH0030
History information is already being maintained in memory (and not in an external file).
S2IH0031
When the history facility is active, history information will be maintained in a file (and not in an internal table).
S2IH0032
When the history facility is active, history information will be maintained in memory (and not in an external file).
S2IH0034
Missing element in internal history table.

S2IH0035

Can't save the value of step number %1b. You can re-generate this value
by running the input file %2b.

S2IH0036

The value specified cannot be saved to a file.

S2IH0037

You must specify a file name to the history save command

S2IH0038

You must specify a file name to the history write command

Chapter 35

)include help page Command

35.1 include help page man page

— include.help —

User Level Required: interpreter

Command Syntax:

```
)include filename
```

Command Description:

The)include command can be used in .input files to place the contents of another file inline with the current file. The path can be an absolute or relative pathname.

—————

35.2 Functions

35.2.1 defun ncloopInclude1

[ncloopIncFileName p600]
[ncloopInclude p600]

— defun ncloopInclude1 —


```
(defun |ncloopInclude1| (name n)
  (let (a)
    (if (setq a (|ncloopIncFileName| name))
        (|ncloopInclude| a n)
        n)))
```

35.2.2 Returns the first non-blank substring of the given string

```
[incFileName p600]
[concat p1003]
```

— **defun ncloopIncFileName** —

```
(defun |ncloopIncFileName| (string)
  "Returns the first non-blank substring of the given string"
  (let (fn)
    (unless (setq fn (|incFileName| string))
      (write-line (concat string " not found"))))
  fn))
```

35.2.3 Open the include file and read it in

The `ncloopInclude0` function is part of the parser and lives in `int-top.boot`. [ncloopInclude0 p74]

— **defun ncloopInclude** —

```
(defun |ncloopInclude| (name n)
  "Open the include file and read it in"
  (with-open-file (st name) (|ncloopInclude0| st name n)))
```

35.2.4 Return the include filename

Given a string we return the first token from the string which is the first non-blank substring. [incBiteOff p601]

— **defun incFileName** —

```
(defun |incFileName| (x)
  "Return the include filename"
  (car (|incBiteOff| x)))
```

35.2.5 Return the next token

Takes a sequence and returns the a list of the first token and the remaining string characters. If there are no remaining string characters the second string is of length 0. Effectively it "bites off" the first token in the string. If the string only 0 or more blanks it returns nil.

— **defun incBiteOff** —

```
(defun |incBiteOff| (x)
  "Return the next token"
  (let (blank nonblank)
    (setq x (string x))
    (when (setq nonblank (position #\space x :test-not #'char=))
      (setq blank (position #\space x :start nonblank))
      (if blank
        (list (subseq x nonblank blank) (subseq x blank))
        (list (subseq x nonblank) ""))))))
```

Chapter 36

)library help page Command

36.1 library help page man page

— library.help —

```
=====
A.14. )library
=====
```

User Level Required: interpreter

Command Syntax:

-)library libName1 [libName2 ...]
-)library)dir dirName
-)library)only objName1 [objlib2 ...]
-)library)noexpose

Command Description:

This command replaces the)load system command that was available in AXIOM releases before version 2.0. The)library command makes available to AXIOM the compiled objects in the libraries listed.

For example, if you)compile dopler.spad in your home directory, issue)library dopler to have AXIOM look at the library, determine the category and domain constructors present, update the internal database with various properties of the constructors, and arrange for the constructors to be automatically loaded when needed. If the)noexpose option has not been given, the constructors will be exposed (that is, available) in the current frame.

If you compiled a file you will have an NRLIB present, for example,

DOPLER.NRLIB, where DOPLER is a constructor abbreviation. The command)library DOPLER will then do the analysis and database updates as above.

To tell the system about all libraries in a directory, use)library)dir dirName where dirName is an explicit directory. You may specify ‘.’ as the directory, which means the current directory from which you started the system or the one you set via the)cd command. The directory name is required.

You may only want to tell the system about particular constructors within a library. In this case, use the)only option. The command)library dopler)only Test1 will only cause the Test1 constructor to be analyzed, autoloaded, etc..

Finally, each constructor in a library are usually automatically exposed when the)library command is used. Use the)noexpose option if you not want them exposed. At a later time you can use)set expose add constructor to expose any hidden constructors.

Note for AXIOM beta testers: At various times this command was called)local and)with before the name)library became the official name.

Also See:

- o)cd
- o)compile
- o)frame
- o)set

¹ “cd” (?? p ??) “frame” (32.5.16 p 543) “set” (44.37.1 p 785)

Chapter 37

)lisp help page Command

37.1 lisp help page man page

— lisp.help —

```
=====
A.15. )lisp
=====
```

User Level Required: development

Command Syntax:

-)lisp [lispExpression]

Command Description:

This command is used by AXIOM system developers to have single expressions evaluated by the Lisp system on which AXIOM is built. The `lispExpression` is read by the Lisp reader and evaluated. If this expression is not complete (unbalanced parentheses, say), the reader will wait until a complete expression is entered.

Since this command is only useful for evaluating single expressions, the `)fin` command may be used to drop out of AXIOM into Lisp.

Also See:

- o)system
- o)boot
- o)fin

1

37.2 Functions

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

¹ “system” (?? p ??) “boot” (5.1.8 p 25) “fin” (31.1.1 p 526)

Chapter 38

)load help page Command

38.1 load help page man page

— load.help —

```
=====
A.16. )load
=====
```

User Level Required: interpreter

Command Description:

This command is obsolete. Use)library instead.

—

38.1.1 defun The)load command (obsolete)

We keep this command around in case anyone has the original Axiom book. [sayKeyedMsg p331]

— defun load —

```
(defun |load| (ignore)
  (declare (ignore ignore))
  (|sayKeyedMsg| 'S2IU0003 nil))
```

—

Chapter 39

)ltrace help page Command

39.1 ltrace help page man page

— ltrace.help —

```
=====
A.17. )ltrace
=====
```

User Level Required: development

Command Syntax:

This command has the same arguments as options as the)trace command.

Command Description:

This command is used by AXIOM system developers to trace Lisp or BOOT functions. It is not supported for general use.

Also See:

- o)boot
- o)lisp
- o)trace

1

¹ "boot" (5.1.8 p 25) "lisp" (?? p ??) "trace" (50.1.7 p 821)

39.1.1 defun The top level)ltrace function

[trace p821]

— defun ltrace —

(defun |ltrace| (arg) (|ltrace| arg))

—————

39.2 Variables Used

39.3 Functions

Chapter 40

)pquit help page Command

40.1 pquit help page man page

— pquit.help —

```
=====
A.18. )pquit
=====
```

User Level Required: interpreter

Command Syntax:

-)pquit

Command Description:

This command is used to terminate AXIOM and return to the operating system. Other than by redoing all your computations or by using the)history)restore command to try to restore your working environment, you cannot return to AXIOM in the same state.

)pquit differs from the)quit in that it always asks for confirmation that you want to terminate AXIOM (the ‘p’ is for ‘protected’). When you enter the)pquit command, AXIOM responds

Please enter y or yes if you really want to leave the interactive
environment and return to the operating system:

If you respond with y or yes, you will see the message

You are now leaving the AXIOM interactive environment.

Issue the command `axiom` to the operating system to start a new session.

and AXIOM will terminate and return you to the operating system (or the environment from which you invoked the system). If you responded with something other than `y` or `yes`, then the message

You have chosen to remain in the AXIOM interactive environment.

will be displayed and, indeed, AXIOM would still be running.

Also See:

- o `)fin`
- o `)history`
- o `)close`
- o `)quit`
- o `)system`

1

40.2 Functions

40.2.1 The top level `pquit` command

[`pquitSpad2Cmd` p612]

— `defun pquit` —

```
(defun |pquit| ()
  "The top level pquit command"
  (|pquitSpad2Cmd|))
```

40.2.2 The top level `pquit` command handler

[`quitSpad2Cmd` p616]
 [`$quitCommandType` p777]

— `defun pquitSpad2Cmd` —

¹ “`fin`” (31.1.1 p 526) “`history`” (34.4.7 p 560) “`close`” (24.2.2 p 488) “`quit`” (41.2.1 p 616) “`system`” (?? p ??)

```
(defun |pquitSpad2Cmd| ()  
  "The top level pquit command handler"  
  (let ((|quitCommandType| '|protected|))  
    (declare (special |quitCommandType|))  
    (|quitSpad2Cmd|)))
```

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

Chapter 41

)quit help page Command

41.1 quit help page man page

— quit.help —

```
=====
A.19. )quit
=====
```

User Level Required: interpreter

Command Syntax:

-)quit
-)set quit protected | unprotected

Command Description:

This command is used to terminate AXIOM and return to the operating system. Other than by redoing all your computations or by using the)history)restore command to try to restore your working environment, you cannot return to AXIOM in the same state.

)quit differs from the)pquit in that it asks for confirmation only if the command

)set quit protected

has been issued. Otherwise,)quit will make AXIOM terminate and return you to the operating system (or the environment from which you invoked the system).

The default setting is)set quit protected so that)quit and)pquit behave in

the same way. If you do issue

```
)set quit unprotected
```

we suggest that you do not (somehow) assign)quit to be executed when you press, say, a function key.

Also See:

- o)fin
- o)history
- o)close
- o)pquit
- o)system

1

41.2 Functions

41.2.1 The top level quit command

[quitSpad2Cmd p616]

— defun quit —

```
(defun |quit| ()
  "The top level quit command"
  (|quitSpad2Cmd|))
```

41.2.2 The top level quit command handler

[upcase p??]
 [queryUserKeyedMsg p??]
 [string2id-n p??]
 [leaveScratchpad p617]
 [sayKeyedMsg p331]
 [tersyscommand p432]
 [\$quitCommandType p777]

¹ “fin” (31.1.1 p 526) “history” (34.4.7 p 560) “close” (24.2.2 p 488) “pquit” (40.2.1 p 612) “system” (?? p ??)

— defun quitSpad2Cmd —

```
(defun |quitSpad2Cmd| ()
  "The top level quit command handler"
  (declare (special |$quitCommandType|))
  (if (eq |$quitCommandType| '|protected|)
      (let (x)
        (setq x (upcase (|queryUserKeyedMsg| 's2iz0031 nil)))
        (when (member (string2id-n x 1) '(y yes)) (|leaveScratchpad|))
        (|sayKeyedMsg| 's2iz0032 nil)
        (tersyscommand))
      (|leaveScratchpad|)))
```

—————

41.2.3 Leave the Axiom interpreter

— defun leaveScratchpad —

```
(defun |leaveScratchpad| ()
  "Leave the Axiom interpreter"
  (bye))
```

—————

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

Chapter 42

)read help page Command

42.1 read help page man page

— read.help —

```
=====
A.20. )read
=====
```

User Level Required: interpreter

Command Syntax:

-)read [fileName]
-)read [fileName] [)quiet] [)ifthere]

Command Description:

This command is used to read .input files into AXIOM. The command

)read matrix.input

will read the contents of the file matrix.input into AXIOM. The ‘.input’ file extension is optional. See the AXIOM User Guide index for more information about .input files.

This command remembers the previous file you edited, read or compiled. If you do not specify a file name, the previous file will be read.

The)ifthere option checks to see whether the .input file exists. If it does not, the)read command does nothing. If you do not use this option and the file does not exist, you are asked to give the name of an existing .input

file.

The `)quiet` option suppresses output while the file is being read.

Also See:

- o `)compile`
- o `)edit`
- o `)history`

1

42.1.1 defun The `)read` command

[readSpad2Cmd p620]

— **defun read** —

```
(defun |read| (arg) (|readSpad2Cmd| arg))
```

42.1.2 defun Implement the `)read` command

[selectOptionLC p459]
 [optionError p429]
 [pathname p998]
 [pathnameTypeId p997]
 [makePathname p998]
 [pathnameName p996]
 [mergePathnames p997]
 [findfile p??]
 [throwKeyedMsg p??]
 [namestring p996]
 [upcase p??]
 [member p1004]
 [/read p622]
 [\$InteractiveMode p24]
 [\$findfile p??]
 [\$UserLevel p784]
 [\$options p??]
 [/editfile p493]

¹ “edit” (30.2.1 p 522) “history” (34.4.7 p 560)

— defun readSpad2Cmd —

```
(defun |readSpad2Cmd| (arg)
  (prog (|$InteractiveModel| fullopt ifthere quiet ef devFTs fileTypes
        ll ft upft fs)
    (declare (special |$InteractiveModel| $findfile |$UserLevel| |$options|
                      /editfile))
    (setq |$InteractiveModel| t)
    (dolist (opt |$options|)
      (setq fullopt
        (|selectOptionLC| (caar opt) '(|quiet| |test| |ifthere|) '|optionError|))
      (cond
        ((eq fullopt '|ifthere|) (setq ifthere t))
        ((eq fullopt '|quiet|) (setq quiet t))))
    (setq ef (|pathname| /editfile))
    (when (eq (|pathnameTypeId| ef) 'spad)
      (setq ef (|makePathname| (|pathnameName| ef) "*" "*")))
    (if arg
      (setq arg (|mergePathnames| (|pathname| arg) ef))
      (setq arg ef))
    (setq devFTs '("input" "INPUT" "boot" "BOOT" "lisp" "LISP"))
    (setq fileTypes
      (cond
        ((eq |$UserLevel| '|interpreter|) '("input" "INPUT"))
        ((eq |$UserLevel| '|compiler|) '("input" "INPUT"))
        (t devFTs)))
    (setq ll ($findfile arg fileTypes))
    (unless ll
      (if ifthere
        (return nil)
        (|throwKeyedMsg| 'S2IL0003 (list (|namestring| arg)))))
    (setq ll (|pathname| ll))
    (setq ft (|pathnameType| ll))
    (setq upft (upcase ft))
    (cond
      ((null (|member| upft fileTypes))
        (setq fs (|namestring| arg))
        (if (|member| upft devFTs)
          (|throwKeyedMsg| 'S2IZ0033 (list fs))
          (|throwKeyedMsg| 'S2IZ0034 (list fs))))
      (t
        (setq /editfile ll)
        (when (string= upft "BOOT") (setq |$InteractiveModel| nil))
        (/read ll quiet)))))
```

42.1.3 defun /read

[p??]
 [/editfile p493]

— defun /read —

```
(defun /read (l q)
  (declare (special /editfile))
  (setq /editfile l)
  (cond
    (q (/rq))
    (t (/rf)) )
  (flag |boot-NewKEY| 'key)
  (|terminateSystemCommand|)
  (|spadPrompt|))
```

—————▶

Chapter 43

)savesystem help page Command

43.1 savesystem help page man page

— savesystem.help —

```
=====
A.8. )savesystem
=====
```

User Level Required: interpreter

Command Syntax:

```
- )savesystem filename
```

Command Description:

This command is used to save an AXIOM image to disk. This creates an executable file which, when started, has everything loaded into it that was there when the image was saved. Thus, after executing commands which cause the loading of some packages, the command:

```
)savesystem /tmp/savesys
```

will create an image that can be restarted with the UNIX command:

```
axiom -ws /tmp/savesys
```


This new system will not need to reload the packages and domains that were already loaded when the system was saved.

There is currently a restriction that only systems started with the command "AXIOMsys" may be saved.

43.1.1 defun The *)savesystem* command

[nequal p??]
 [helpSpad2Cmd p550]
 [spad-save p945]

— **defun savesystem** —

```
(defun |savesystem| (arg)
  (if (or (nequal (|#| arg) 1) (null (symbolp (car arg))))
      (|helpSpad2Cmd| '(|savesystem|))
      (spad-save (symbol-name (car arg)))))
```

Chapter 44

)set help page Command

44.1 set help page man page

— set.help —

```
=====
A.21. )set
=====
```

User Level Required: interpreter

Command Syntax:

-)set
-)set label1 [... labelN]
-)set label1 [... labelN] newValue

Command Description:

The)set command is used to view or set system variables that control what messages are displayed, the type of output desired, the status of the history facility, the way AXIOM user functions are cached, and so on. Since this collection is very large, we will not discuss them here. Rather, we will show how the facility is used. We urge you to explore the)set options to familiarize yourself with how you can modify your AXIOM working environment. There is a HyperDoc version of this same facility available from the main HyperDoc menu. Click [\[here\]](#) to go to it.

The)set command is command-driven with a menu display. It is tree-structured. To see all top-level nodes, issue)set by itself.

)set

Variables with values have them displayed near the right margin. Subtrees of selections have “...” displayed in the value field. For example, there are many kinds of messages, so issue `)set message` to see the choices.

```
)set message
```

The current setting for the variable that displays whether computation times are displayed is visible in the menu displayed by the last command. To see more information, issue

```
)set message time
```

This shows that time printing is on now. To turn it off, issue

```
)set message time off
```

As noted above, not all settings have so many qualifiers. For example, to change the `)quit` command to being unprotected (that is, you will not be prompted for verification), you need only issue

```
)set quit unprotected
```

Also See:

- o `)quit`

1

44.2 Overview

This section contains tree of information used to initialize the `)set` command in the interpreter. The current list is:

Variable	Description	Current Value

<code>compile</code>	Library compiler options	...
<code>breakmode</code>	execute break processing on error	break
<code>expose</code>	control interpreter constructor exposure	...
<code>functions</code>	some interpreter function options	...
<code>fortran</code>	view and set options for FORTRAN output	...
<code>kernel</code>	library functions built into the kernel for efficiency	...
<code>hyperdoc</code>	options in using HyperDoc	...

¹“quit” (41.2.1 p 616)

help	view and set some help options	...
history	save workspace values in a history file	on
messages	show messages for various system features	...
naglink	options for NAGLink	...
output	view and set some output options	...
quit	protected or unprotected quit	unprotected
streams	set some options for working with streams	...
system	set some system development variables	...
userlevel	operation access level of system user	development

Variables with current values of ... have further sub-options.
 For example, issue `)set system` to see what the options are
 for system. For more information, issue `)help set .`

44.3 Variables Used

44.4 Functions

44.4.1 Initialize the set variables

The argument `settree` is initially the `$setOption` variable. The fourth element is a union-style switch symbol. The fifth element is usually a variable to set. The sixth element is a subtree to recurse for the `TREE` switch. The seventh element is usually the default value. For more detailed explanations see the list structure section 44.5. [sayMSG p333]

[literals p??]

[translateYesNo2TrueFalse p632]

[tree p??]

[initializeSetVariables p627]

— **defun initializeSetVariables** —

```
(defun |initializeSetVariables| (settree)
  "Initialize the set variables"
  (dolist (setdata settree)
    (case (fourth setdata)
      (function
        (if (functionp (fifth setdata))
            (funcall (fifth setdata) '|%initialize%|)
            (|sayMSG| (concatenate 'string "  Function not implemented. "
                                   (package-name *package*) ":" (string (fifth setdata))))))
      (integer (set (fifth setdata) (seventh setdata)))
      (string (set (fifth setdata) (seventh setdata)))
      (literals
        (set (fifth setdata) (|translateYesNo2TrueFalse| (seventh setdata))))
      (tree (|initializeSetVariables| (sixth setdata))))))
```

44.4.2 Reset the workspace variables

```
[copy p??]
[initializeSetVariables p627]
[/countlist p??]
[/editfile p493]
[/sourcefiles p??]
[/pretty p??]
[/spacelist p??]
[/timerlist p??]
[$sourceFiles p??]
[$existingFiles p??]
[$functionTable p481]
[$boot p25]
[$compileMapFlag p??]
[$echoLineStack p??]
[$operationNameList p??]
[$slamFlag p??]
[$CommandSynonymAlist p458]
[$InitialCommandSynonymAlist p456]
[$UserAbbreviationsAlist p??]
[$msgAlist p328]
[$msgDatabase p??]
[$msgDatabaseName p328]
[$dependeeClosureAlist p??]
[$IOindex p??]
[$coerceIntByMapCounter p??]
[$e p??]
[$env p??]
[$setOptions p??]
```

— **defun resetWorkspaceVariables** —

```
(defun |resetWorkspaceVariables| ()
  "Reset the workspace variables"
  (declare (special /countlist /editfile /sourcefiles |$sourceFiles| /pretty
    /spacelist /timerlist |$existingFiles| |$functionTable| $boot
    |$compileMapFlag| |$echoLineStack| |$operationNameList| |$slamFlag| | |
    |$CommandSynonymAlist| |$InitialCommandSynonymAlist|
    |$UserAbbreviationsAlist| |$msgAlist| |$msgDatabase| |$msgDatabaseName|
    |$dependeeClosureAlist| |$IOindex| |$coerceIntByMapCounter| |$e| |$env|
    |$setOptions|))
```

```

(setq /countlist nil)
(setq /editfile nil)
(setq /sourcefiles nil)
(setq |$sourceFiles| nil)
(setq /pretty nil)
(setq /spacelist nil)
(setq /timerlist nil)
(setq |$existingFiles| (make-hash-table :test #'equal))
(setq |$functionTable| nil)
(setq $boot nil)
(setq |$compileMapFlag| nil)
(setq |$echoLineStack| nil)
(setq |$operationNameList| nil)
(setq |$slamFlag| nil)
(setq |$CommandSynonymAlist| (copy |$InitialCommandSynonymAlist|))
(setq |$UserAbbreviationsAlist| nil)
(setq |$msgAlist| nil)
(setq |$msgDatabase| nil)
(setq |$msgDatabaseName| nil)
(setq |$dependeeClosureAlist| nil)
(setq |$I0index| 1)
(setq |$coerceIntByMapCounter| 0)
(setq |$e| (cons (cons nil nil) nil))
(setq |$env| (cons (cons nil nil) nil))
(|initializeSetVariables| |$setOptions|))

```

44.4.3 Display the set option information

```

[displaySetVariableSettings p631]
[centerAndHighlight p??]
[concat p1003]
[object2String p??]
[specialChar p936]
[sayBrightly p??]
[bright p??]
[sayMSG p333]
[boot-equal p??]
[sayMessage p??]
[eval p??]
[literals p??]
[translateTrueFalse2YesNo p633]
[$linelength p751]

```

— defun displaySetOptionInformation —

```

(defun |displaySetOptionInformation| (arg setdata)
  "Display the set option information"
  (let (current)
    (declare (special $linelength))
    (cond
      ((eq (fourth setdata) 'tree)
        (|displaySetVariableSettings| (sixth setdata) (first setdata)))
      (t
        (|centerAndHighlight|
          (concat "The " (|object2String| arg) " Option"
            $linelength (|specialChar| 'hbar)))
        (|sayBrightly|
          '(|%l| ,@( |bright| "Description:") ,(second setdata)))
        (case (fourth setdata)
          (function
            (terpri)
            (if (functionp (fifth setdata))
              (funcall (fifth setdata) '|%describe%|')
              (|sayMSG| " Function not implemented.")))
          (integer
            (|sayMessage|
              '(" The" ,@( |bright| arg) "option"
                " may be followed by an integer in the range"
                ,@( |bright| (elt (sixth setdata) 0)) "to"
                |%l| ,@( |bright| (elt (sixth setdata) 1)) "inclusive."
                " The current setting is" ,@( |bright| (|eval| (fifth setdata))))))
          (string
            (|sayMessage|
              '(" The" ,@( |bright| arg) "option"
                " is followed by a string enclosed in double quote marks."
                '|%l| " The current setting is"
                ,@( |bright| (list '|'| (|eval| (fifth setdata)) '|'|))))))
          (literals
            (|sayMessage|
              '(" The" ,@( |bright| arg) "option"
                " may be followed by any one of the following:"))
            (setq current
              (|translateTrueFalse2YesNo| (|eval| (fifth setdata))))
            (dolist (name (sixth setdata))
              (if (boot-equal name current)
                (|sayBrightly| '(" ->" ,@( |bright| (|object2String| name))))
                (|sayBrightly| (list " " (|object2String| name))))
              (|sayMessage| " The current setting is indicated."))))))

```

44.4.4 Display the set variable settings

```
[concat p1003]
[object2String p??]
[centerAndHighlight p??]
[sayBrightly p??]
[say p??]
[fillerSpaces p20]
[specialChar p936]
[pairp p??]
[concat p1003]
[satisfiesUserLevel p431]
[spaddifference p??]
[poundsign p??]
[eval p??]
[bright p??]
[literals p??]
[translateTrueFalse2YesNo p633]
[tree p??]
[$linelength p751]
```

— defun displaySetVariableSettings —

```
(defun |displaySetVariableSettings| (settree label)
  "Display the set variable settings"
  (let (setoption opt subtree subname)
    (declare (special $linelength))
    (if (eq label '|')
      (setq label ")set")
      (setq label (concat " " (|object2String| label) " ")))
    (|centerAndHighlight|
     (concat "Current Values of" label " Variables") $linelength '| |)
    (terpri)
    (|sayBrightly|
     (list "Variable" "Description"
           "Current Value" ))
    (say (|fillerSpaces| $linelength (|specialChar| '|hbar|)))
    (setq subtree nil)
    (dolist (setdata settree)
      (when (|satisfiesUserLevel| (third setdata))
        (setq setoption (|object2String| (first setdata)))
        (setq setoption
         (concat setoption
          (|fillerSpaces| (spaddifference 13 (|#| setoption)) " ")
          (second setdata)))
        (setq setoption
         (concat setoption
          (|fillerSpaces| (spaddifference 55 (|#| setoption)) " ")))
```



```

(case (fourth setdata)
  (function
    (setq opt
      (if (functionp (fifth setdata))
          (funcall (fifth setdata) '|%display%|)
          "unimplemented"))
    (cond
      ((pairp opt)
        (setq opt
          (do ((t2 opt (cdr t2)) t1 (o nil))
              ((or (atom t2) (progn (setq o (car t2)) nil)) t1)
              (setq t1 (append t1 (cons o (cons " " nil)))))))
        (|sayBrightly| (|concat| setoption '|%b| opt '|%d|)))
      (string
        (setq opt (|object2String| (|eval| (fifth setdata))))
        (|sayBrightly| '(,setoption ,@(|bright| opt))))
      (integer
        (setq opt (|object2String| (|eval| (fifth setdata))))
        (|sayBrightly| '(,setoption ,@(|bright| opt))))
      (literals
        (setq opt (|object2String|
                    (|translateTrueFalse2YesNo| (|eval| (fifth setdata)))))
        (|sayBrightly| '(,setoption ,@(|bright| opt))))
      (TREE
        (|sayBrightly| '(,setoption ,@(|bright| "..."))
          (setq subtree t)
          (setq subname (|object2String| (first setdata))))))
    (terpri)
    (when subtree
      (|sayBrightly|
        ("Variables with current values of" ,@(|bright| "...")
         "have further sub-options. For example,")
      (|sayBrightly|
        ("issue" ,@(|bright| ")set ") ,subname
         " to see what the options are for" ,@(|bright| subname) "."
         |%l| "For more information, issue" ,@(|bright| ")help set") ".")))))

```

44.4.5 Translate options values to t or nil

[member p1004]

— defun translateYesNo2TrueFalse —

```

(defun |translateYesNo2TrueFalse| (x)
  "Translate options values to t or nil"
  (cond

```

```
((|member| x '(|yes| |on|)) t)
(|member| x '(|no| |off|)) nil)
(t x)))
```

44.4.6 Translate t or nil to option values

— defun translateTrueFalse2YesNo —

```
(defun |translateTrueFalse2YesNo| (x)
  "Translate t or nil to option values"
  (cond
    ((eq x t) '|on|)
    ((null x) '|off|)
    (t x)))
```

44.5 The list structure

The structure of each list item consists of 7 items. Consider this example:

```
(userlevel
  "operation access level of system user"
  interpreter
  LITERALS
  $UserLevel
  (interpreter compiler development)
  development)
```

The list looks like (the names in bold are accessor names that can be found in **property.lisp.pamphlet**[1]. Look for "setName".):

- 1** *Name* the keyword the user will see. In this example the user would say "**)set output userlevel**".
- 2** *Label* the message the user will see. In this example the user would see "operation access level of system user".
- 3** *Level* the level where the command will be accepted. There are three levels: interpreter, compiler, development. These commands are restricted to keep the user from causing damage.

4 Type a symbol, one of **FUNCTION**, **INTEGER**, **STRING**, **LITERALS**, **FILENAME** or **TREE**.

5 *Var*

FUNCTION is the function to call

INTEGER is the variable holding the current user setting.

STRING is the variable holding the current user setting.

LITERALS variable which holds the current user setting.

FILENAME is the variable that holds the current user setting.

TREE

6 *Leaf*

FUNCTION is the list of all possible values

INTEGER is the range of possible values

STRING is a list of all possible values

LITERALS is a list of all of the possible values

FILENAME is the function to check the filename

TREE

7 *Def* is the default value

FUNCTION is the default setting

INTEGER is the default setting

STRING is the default setting

LITERALS is the default setting

FILENAME is the default value

TREE

44.6 breakmode

----- The breakmode Option -----

Description: execute break processing on error

The breakmode option may be followed by any one of the following:

```
nobreak
-> break
query
resume
```

```
fastlinks
quit
```

The current setting is indicated.

44.6.1 defvar \$BreakMode

— initvars —

```
(defvar |$BreakMode| ' |nobreak| "execute break processing on error")
```

— breakmode —

```
(|breakmode|
 "execute break processing on error"
 |interpreter|
 LITERALS
 |$BreakMode|
 (|nobreak| |break| |query| |resume| |fastlinks| |quit|)
 |nobreak|) ; needed to avoid possible startup looping
```

44.7 debug

Current Values of debug Variables

Variable	Description	Current Value

lambdatype	Show type information for #1 syntax	off
dalymode	Interpret leading open paren as lisp	off

— debug —

```
(|debug|
 "debug options"
 |interpreter|
 TREE
 |novar|
```

```
(
\getchunk{debuglambdtype}
\getchunk{debugdalymode}
))
```

44.8 debug lambda type

----- The lambdtype Option -----

Description: Show type information for #1 syntax

44.8.1 defvar \$lambdtype

— initvars —

```
(defvar $lambdtype nil "show type information for #1 syntax")
```

— debuglambdtype —

```
(|lambdtype|
 "show type information for #1 syntax"
 |interpreter|
 LITERALS
 $lambdtype
 (|on| |off|)
 |off|)
```

44.9 debug dalymode

The `$dalymode` variable is used in a case statement in `intloopReadConsole`. This variable can be set to any non-nil value. When not nil the interpreter will send any line that begins with an “(” to be sent to the underlying lisp. This is useful for debugging Axiom. The normal value of this variable is NIL.

This variable was created as an alternative to prefixing every lisp command with `)lisp`. When doing a lot of debugging this is tedious and error prone. This variable was created to shortcut

that process. Clearly it breaks some semantics of the language accepted by the interpreter as parens are used for grouping expressions.

----- The dalymode Option -----

Description: Interpret leading open paren as lisp

44.9.1 defvar \$dalymode

— initvars —

```
(defvar $dalymode nil "Interpret leading open paren as lisp")
```

— debugdalymode —

```
(|dalymode|
 "Interpret leading open paren as lisp"
 |interpreter|
 LITERALS
 $dalymode
 (|on| |off|)
 |off|)
```

44.10 compile

Current Values of compiler Variables

Variable	Description	Current Value
output	library in which to place compiled code	
input	controls libraries from which to load compiled code	

— compile —

```
(|compiler|
 "Library compiler options")
```

```

|interpreter|
TREE
|novar|
(
\getchunk{compileoutput}
\getchunk{compileinput}
))

```

44.11 compile output

----- The output Option -----

Description: library in which to place compiled code

— compileoutput —

```

(|output|
 "library in which to place compiled code"
|interpreter|
FUNCTION
|setOutputLibrary|
NIL
|htSetOutputLibrary|
)

```

44.12 Variables Used

44.13 Functions

44.13.1 The set output command handler

```

[poundsign p??]
[describeOutputLibraryArgs p639]
[filep p??]
[openOutputLibrary p640]
[$outputLibraryName p??]

```

— defun setOutputLibrary —

```
(defun |setOutputLibrary| (arg)
  "The set output command handler"
  (let (fn)
    (declare (special |$outputLibraryName|))
    (cond
      ((eq arg '|%initialize%|) (setq |$outputLibraryName| nil))
      ((eq arg '|%display%|) (or |$outputLibraryName| "user.lib"))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?)) (/= (|#| arg) 1))
      (|describeOutputLibraryArgs|))
    (t
     (when (probe-file (setq fn (princ-to-string (car arg))))
       (setq fn (truename fn)))
     (|openOutputLibrary| (setq |$outputLibraryName| fn))))))
```

44.13.2 Describe the set output library arguments

[sayBrightly p??]

— defun describeOutputLibraryArgs —

```
(defun |describeOutputLibraryArgs| ()
  "Describe the set output library arguments"
  (|sayBrightly| (list
    '|%b| "set compile output library"
    '|%d| "is used to tell the compiler where to place"
    '|%l| "compiled code generated by the library compiler. By default it goes"
    '|%l| "in a file called"
    '|%b| "user.lib"
    '|%d| "in the current directory.")))
```

44.13.3 defvar \$output-library

— initvars —

```
(defvar output-library nil)
```

44.13.4 Open the output library

The input-libraries and output-library are now truename based. [dropInputLibrary p643]
 [output-library p639]
 [input-libraries p642]

— defun openOutputLibrary —

```
(defun |openOutputLibrary| (lib)
  "Open the output library"
  (declare (special output-library input-libraries))
  (|dropInputLibrary| lib)
  (setq output-library (truename lib))
  (push output-library input-libraries))
```

—————

44.14 compile input

----- The input Option -----

Description: controls libraries from which to load compiled code

)set compile input add library is used to tell AXIOM to add library to the front of the path which determines where compiled code is loaded from.
)set compile input drop library is used to tell AXIOM to remove library from this path.

— compileinput —

```
(|input|
  "controls libraries from which to load compiled code"
  |interpreter|
  FUNCTION
  |setInputLibrary|
  NIL
  |htSetInputLibrary|)
```

—————

44.15 Variables Used

44.16 Functions

44.16.1 The set input library command handler

The input-libraries is now maintained as a list of truenames. [describeInputLibraryArgs p642]

```
[pairp p??]
[qcar p??]
[qcdr p??]
[selectOptionLC p459]
[addInputLibrary p642]
[dropInputLibrary p643]
[setInputLibrary p641]
[input-libraries p642]
```

— defun setInputLibrary —

```
(defun |setInputLibrary| (arg)
  "The set input library command handler"
  (declare (special input-libraries))
  (let (tmp1 filename act)
    (cond
      ((eq arg '|%initialize%|) t)
      ((eq arg '|%display%|) (mapcar #'namestring input-libraries))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeInputLibraryArgs|))
      ((and (pairp arg)
            (progn
              (setq act (qcar arg))
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                   (eq (qcdr tmp1) nil)
                   (progn (setq filename (qcar tmp1)) t))))
       (setq act (|selectOptionLC| act '(|add| |drop|) nil)))
      (cond
        ((eq act '|add|)
         (|addInputLibrary| (truenome (princ-to-string filename))))
        ((eq act '|drop|)
         (|dropInputLibrary| (truenome (princ-to-string filename))))))
    (t (|setInputLibrary| nil))))
```

—

44.16.2 Describe the set input library arguments

[sayBrightly p??]

— defun describeInputLibraryArgs —

```
(defun |describeInputLibraryArgs| ()
  "Describe the set input library arguments"
  (|sayBrightly| (list
    '|%b| ")set compile input add library"
    '|%d| "is used to tell AXIOM to add"
    '|%b| "library"
    '|%d| "to"
    '|%l| "the front of the path used to find compile code."
    '|%l|
    '|%b| ")set compile input drop library"
    '|%d| "is used to tell AXIOM to remove"
    '|%b| "library"
    '|%d|
    '|%l| "from this path.")))
```

—————

44.16.3 Add the input library to the list

The input-libraries variable is now maintained as a list of truenames. [dropInputLibrary p643]

[input-libraries p642]

— defun addInputLibrary —

```
(defun |addInputLibrary| (lib)
  "Add the input library to the list"
  (declare (special input-libraries))
  (|dropInputLibrary| lib)
  (push (trueName lib) input-libraries))
```

—————

44.16.4 defvar \$input-libraries

— initvars —

```
(defvar input-libraries nil)
```

44.16.5 Drop an input library from the list

[input-libraries p642]

— defun dropInputLibrary —

```
(defun |dropInputLibrary| (lib)
  "Drop an input library from the list"
  (declare (special input-libraries))
  (setq input-libraries (delete (truename lib) input-libraries :test #'equal)))
```

44.17 expose

----- The expose Option -----

Description: control interpreter constructor exposure

The following groups are explicitly exposed in the current frame (called initial):

```
        basic
categories
        naglink
        anna
```

The following constructors are explicitly exposed in the current frame:

```
        there are no explicitly exposed constructors
```

The following constructors are explicitly hidden in the current frame:

```
        there are no explicitly hidden constructors
```

When)set expose is followed by no arguments, the information you now see is displayed. When followed by the initialize argument, the exposure group data in the file interp.exposed is read and is then available. The arguments add and drop are used to add or drop exposure groups or explicit constructors from the local frame exposure data. Issue

```
        )set expose add    or    )set expose drop
for more information.
```

— expose —

```
(|expose|
 "control interpreter constructor exposure"
 |interpreter|
 FUNCTION
 |setExpose|
 NIL
 |htSetExpose|)
```

44.18 Variables Used

NOTE: If you add new algebra you must also update this list otherwise the new algebra won't be loaded by the interpreter when needed.

44.18.1 defvar \$globalExposureGroupAlist

— initvars —

```
(defvar |$globalExposureGroupAlist|
 '(
 ;;define the groups |basic| |naglink| |anna| |categories| |Hidden| |defaults|
 (|basic|
 (|AffineAlgebraicSetComputeWithGroebnerBasis| . AFALGGRO)
 (|AffineAlgebraicSetComputeWithResultant| . AFALGRES)
 (|AffinePlane| . AFFPL)
 (|AffinePlaneOverPseudoAlgebraicClosureOfFiniteField| . AFFPLPS)
 (|AffineSpace| . AFFSP)
 (|AlgebraicManipulations| . ALGMANIP)
 (|AlgebraicNumber| . AN)
 (|AlgFactor| . ALGFACT)
 (|AlgebraicMultFact| . ALGMFACT)
 (|AlgebraPackage| . ALGPKG)
 (|AlgebraGivenByStructuralConstants| . ALGSC)
 (|Any| . ANY)
 (|AnyFunctions1| . ANY1)
 (|ApplicationProgramInterface| . API)
 (|ArrayStack| . ASTACK)
 (|AssociatedJordanAlgebra| . JORDAN)
 (|AssociatedLieAlgebra| . LIE)
 (|AttachPredicates| . PMPRED)
 (|AxiomServer| . AXSERV)
 (|BalancedBinaryTree| . BBTREE)
```

```

(|BasicOperator| . BOP)
(|BasicOperatorFunctions1| . BOP1)
(|Bezier| . BEZIER)
(|BinaryExpansion| . BINARY)
(|BinaryFile| . BINFILE)
(|BinarySearchTree| . BSTREE)
(|BinaryTournament| . BTOURN)
(|BinaryTree| . BTREE)
(|Bits| . BITS)
(|BlasLevelOne| . BLAS1)
(|BlowUpPackage| . BLUPACK)
(|BlowUpWithHamburgerNoether| . BLHN)
(|BlowUpWithQuadTrans| . BLQT)
(|Boolean| . BOOLEAN)
(|CardinalNumber| . CARD)
(|CartesianTensor| . CARTEN)
(|CartesianTensorFunctions2| . CARTEN2)
(|Character| . CHAR)
(|CharacterClass| . CCLASS)
(|CharacteristicPolynomialPackage| . CHARPOL)
(|CliffordAlgebra| . CLIF)
(|Color| . COLOR)
(|CommonDenominator| . CDEN)
(|Commutator| . COMM)
(|Complex| . COMPLEX)
(|ComplexDoubleFloatMatrix| . CDFMAT)
(|ComplexDoubleFloatVector| . CDFVEC)
(|ComplexFactorization| . COMPFAC)
(|ComplexFunctions2| . COMPLEX2)
(|ComplexRootPackage| . CMPLXRT)
(|ComplexTrigonometricManipulations| . CTRIGMNP)
(|ContinuedFraction| . CONTFRAC)
(|CoordinateSystems| . COORDSYS)
(|CRAPackage| . CRAPACK)
(|CycleIndicators| . CYCLES)
(|Database| . DBASE)
(|DataList| . DLIST)
(|DecimalExpansion| . DECIMAL)
(|DenavitHartenbergMatrix| . DHMATRIX)
(|Dequeue| . DEQUEUE)
(|DesingTree| . DSTREE)
(|DesingTreePackage| . DTP)
(|DiophantineSolutionPackage| . DIOSP)
(|DirichletRing| . DIRRING)
(|DirectProductFunctions2| . DIRPROD2)
(|DisplayPackage| . DISPLAY)
(|DistinctDegreeFactorize| . DDFACT)
(|Divisor| . DIV)
(|DoubleFloat| . DFLOAT)
(|DoubleFloatMatrix| . DFMAT)

```

```

(|DoubleFloatVector| . DFVEC)
(|DoubleFloatSpecialFunctions| . DFSFUN)
(|DrawComplex| . DRAWCX)
(|DrawNumericHack| . DRAWHACK)
(|DrawOption| . DROPT)
(|EigenPackage| . EP)
(|ElementaryFunctionDefiniteIntegration| . DEFINTEF)
(|ElementaryFunctionLODESolver| . LODEEF)
(|ElementaryFunctionODESolver| . ODEEF)
(|ElementaryFunctionSign| . SIGNEF)
(|ElementaryFunctionStructurePackage| . EFSTRUC)
(|Equation| . EQ)
(|EquationFunctions2| . EQ2)
(|ErrorFunctions| . ERROR)
(|EuclideanGroebnerBasisPackage| . GBEUCLID)
(|Exit| . EXIT)
(|Export3D| . EXP3D)
(|Expression| . EXPR)
(|ExpressionFunctions2| . EXPR2)
(|ExpressionSolve| . EXPRSOL)
(|ExpressionSpaceFunctions2| . ES2)
(|ExpressionSpaceODESolver| . EXPRODE)
(|ExpressionToOpenMath| . OMEXPR)
(|ExpressionToUnivariatePowerSeries| . EXPR2UPS)
(|Factored| . FR)
(|FactoredFunctions2| . FR2)
(|FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber| . FACTEXT)
(|FactorisationOverPseudoAlgebraicClosureOfRationalNumber| . FACTRN)
(|File| . FILE)
(|FileName| . FNAME)
(|FiniteAbelianMonoidRingFunctions2| . FAMR2)
(|FiniteDivisorFunctions2| . FDIV2)
(|FiniteFieldFactorizationWithSizeParseBySideEffect| . FFFACTSE)
(|FiniteField| . FF)
(|FiniteFieldCyclicGroup| . FFCG)
(|FiniteFieldPolynomialPackage2| . FFPOLY2)
(|FiniteFieldNormalBasis| . FFNB)
(|FiniteFieldHomomorphisms| . FFHOM)
(|FiniteFieldSquareFreeDecomposition| . FFSQFR)
(|FiniteLinearAggregateFunctions2| . FLAGG2)
(|FiniteLinearAggregateSort| . FLASORT)
(|FiniteSetAggregateFunctions2| . FSAGG2)
(|FlexibleArray| . FARRAY)
(|Float| . FLOAT)
(|FloatingRealPackage| . FLOATRP)
(|FloatingComplexPackage| . FLOATCP)
(|FourierSeries| . FSERIES)
(|Fraction| . FRAC)
(|FractionalIdealFunctions2| . FRIDEAL2)
(|FractionFreeFastGaussian| . FFFG)

```

```

(|FractionFreeFastGaussianFractions| . FFFGF)
(|FractionFunctions2| . FRAC2)
(|FreeNilpotentLie| . FNLA)
(|FullPartialFractionExpansion| . FPARFRAC)
(|FunctionFieldCategoryFunctions2| . FFCAT2)
(|FunctionSpaceAssertions| . PMASSFS)
(|FunctionSpaceAttachPredicates| . PMPREDFS)
(|FunctionSpaceComplexIntegration| . FSCINT)
(|FunctionSpaceFunctions2| . FS2)
(|FunctionSpaceIntegration| . FSINT)
(|FunctionSpacePrimitiveElement| . FSPRMELT)
(|FunctionSpaceSum| . SUMFS)
(|GaussianFactorizationPackage| . GAUSSFAC)
(|GeneralPackageForAlgebraicFunctionField| . GPAFF)
(|GeneralUnivariatePowerSeries| . GSERIES)
(|GenerateUnivariatePowerSeries| . GENUPS)
(|GnuDraw| . GDRAW)
(|GraphicsDefaults| . GRDEF)
(|GroebnerPackage| . GB)
(|GroebnerFactorizationPackage| . GBF)
(|Guess| . GUESS)
(|GuessAlgebraicNumber| . GUESSAN)
(|GuessFinite| . GUESSF)
(|GuessFiniteFunctions| . GUESSF1)
(|GuessInteger| . GUESSINT)
(|GuessOption| . GOPT)
(|GuessPolynomial| . GUESSP)
(|GuessUnivariatePolynomial| . GUESSUP)
(|HallBasis| . HB)
(|Heap| . HEAP)
(|HexadecimalExpansion| . HEXADEC)
(|HTMLFormat| . HTMLFORM)
(|IdealDecompositionPackage| . IDECOMP)
(|IndexCard| . ICARD)
(|InfClsPt| . ICP)
(|InfiniteProductCharacteristicZero| . INFPRODO)
(|InfiniteProductFiniteField| . INPRODFF)
(|InfiniteProductPrimeField| . INPRODPF)
(|InfiniteTuple| . ITUPLE)
(|InfiniteTupleFunctions2| . ITFUN2)
(|InfiniteTupleFunctions3| . ITFUN3)
(|InfinitelyClosePoint| . INFCLSPT)
(|InfinitelyClosePointOverPseudoAlgebraicClosureOfFiniteField| . INFCLSPS)
(|Infinity| . INFINITY)
(|Integer| . INT)
(|IntegerCombinatoricFunctions| . COMBINAT)
(|IntegerLinearDependence| . ZLINDEP)
(|IntegerNumberTheoryFunctions| . INTHEORY)
(|IntegerPrimesPackage| . PRIMES)
(|IntegerRetractions| . INTRET)

```



```

(|IntegerRoots| . IROOT)
(|IntegrationResultFunctions2| . IR2)
(|IntegrationResultRFToFunction| . IRRF2F)
(|IntegrationResultToFunction| . IR2F)
(|InterfaceGroebnerPackage| . INTERGB)
(|InterpolateFormsPackage| . INTFRSP)
(|IntersectionDivisorPackage| . INTDIVP)
(|Interval| . INTRVL)
(|InventorDataSink| . IVDATA)
(|InventorViewPort| . IVVIEW)
(|InventorRenderPackage| . IVREND)
(|InverseLaplaceTransform| . INVLAPLA)
(|IrrRepSymNatPackage| . IRSN)
(|KernelFunctions2| . KERNEL2)
(|KeyedAccessFile| . KAFILE)
(|LaplaceTransform| . LAPLACE)
(|LazardMorenoSolvingPackage| . LAZM3PK)
(|Library| . LIB)
(|LieSquareMatrix| . LSQM)
(|LinearOrdinaryDifferentialOperator| . LOD0)
(|LinearSystemMatrixPackage| . LSMP)
(|LinearSystemMatrixPackage1| . LSMP1)
(|LinearSystemFromPowerSeriesPackage| . LISYSER)
(|LinearSystemPolynomialPackage| . LSPP)
(|List| . LIST)
(|LinesOpPack| . LOP)
(|ListFunctions2| . LIST2)
(|ListFunctions3| . LIST3)
(|ListToMap| . LIST2MAP)
(|LocalParametrizationOfSimplePointPackage| . LPARSPT)
(|MakeFloatCompiledFunction| . MKFLCFN)
(|MakeFunction| . MKFUNC)
(|MakeRecord| . MKRECORD)
(|MappingPackage1| . MAPPKG1)
(|MappingPackage2| . MAPPKG2)
(|MappingPackage3| . MAPPKG3)
(|MappingPackage4| . MAPPKG4)
(|MathMLFormat| . MMLFORM)
(|Matrix| . MATRIX)
(|MatrixCategoryFunctions2| . MATCAT2)
(|MatrixCommonDenominator| . MCDEN)
(|MatrixLinearAlgebraFunctions| . MATLIN)
(|MergeThing| . MTHING)
(|ModularDistinctDegreeFactorizer| . MDDFACT)
(|ModuleOperator| . MODOP)
(|MonoidRingFunctions2| . MRF2)
(|MoreSystemCommands| . MSYSCMD)
(|MPolyCatFunctions2| . MPC2)
(|MPolyCatRationalFunctionFactorizer| . MPRFF)
(|Multiset| . MSET)

```

```

(|MultivariateFactorize| . MULTFACT)
(|MultivariatePolynomial| . MPOLY)
(|MultFiniteFactorize| . MFINFACT)
(|MyUnivariatePolynomial| . MYUP)
(|MyExpression| . MYEXPR)
(|NeitherSparseOrDensePowerSeries| . NSDPS)
(|NewtonPolygon| . NPOLYGON)
(|NoneFunctions1| . NONE1)
(|NonNegativeInteger| . NNI)
(|NottinghamGroup| . NOTTING)
(|NormalizationPackage| . NORMPK)
(|NormInMonogenicAlgebra| . NORMMA)
(|NumberTheoreticPolynomialFunctions| . NTPOLFN)
(|Numeric| . NUMERIC)
(|NumericalOrdinaryDifferentialEquations| . NUMODE)
(|NumericalQuadrature| . NUMQUAD)
(|NumericComplexEigenPackage| . NCEP)
(|NumericRealEigenPackage| . NREP)
(|NumericContinuedFraction| . NCNTFRAC)
(|Octonion| . OCT)
(|OctonionCategoryFunctions2| . OCTCT2)
(|OneDimensionalArray| . ARRAY1)
(|OneDimensionalArrayFunctions2| . ARRAY12)
(|OnePointCompletion| . ONECOMP)
(|OnePointCompletionFunctions2| . ONECOMP2)
(|OpenMathConnection| . OMCONN)
(|OpenMathDevice| . OMDEV)
(|OpenMathEncoding| . OMENC)
(|OpenMathError| . OMERR)
(|OpenMathErrorKind| . OMERRK)
(|OpenMathPackage| . OMPKG)
(|OpenMathServerPackage| . OMSERVER)
(|OperationsQuery| . OPQUERY)
(|OrderedCompletion| . ORDCOMP)
(|OrderedCompletionFunctions2| . ORDCOMP2)
(|OrdinaryDifferentialRing| . ODR)
(|OrdSetInts| . OSI)
(|OrthogonalPolynomialFunctions| . ORTHPOL)
(|OutputPackage| . OUT)
(|PackageForAlgebraicFunctionField| . PAFF)
(|PackageForAlgebraicFunctionFieldOverFiniteField| . PAFFFF)
(|PackageForPoly| . PFORP)
(|PadeApproximantPackage| . PADEPAC)
(|Palette| . PALETTE)
(|PartialFraction| . PFR)
(|PatternFunctions2| . PATTERN2)
(|ParametricPlaneCurve| . PARPCURV)
(|ParametricSpaceCurve| . PARSCURV)
(|ParametricSurface| . PARSURF)
(|ParametricPlaneCurveFunctions2| . PARPC2)

```

```

(|ParametricSpaceCurveFunctions2| . PARSC2)
(|ParametricSurfaceFunctions2| . PARSU2)
(|ParametrizationPackage| . PARAMP)
(|PartitionsAndPermutations| . PARTPERM)
(|PatternMatch| . PATMATCH)
(|PatternMatchAssertions| . PMASS)
(|PatternMatchResultFunctions2| . PATRES2)
(|PendantTree| . PENDTREE)
(|Permanent| . PERMAN)
(|PermutationGroupExamples| . PGE)
(|PermutationGroup| . PERMGRP)
(|Permutation| . PERM)
(|Pi| . HACKPI)
(|PiCoercions| . PICOERCE)
(|Places| . PLACES)
(|PlacesOverPseudoAlgebraicClosureOfFiniteField| . PLACESPS)
(|Plcs| . PLCS)
(|PointFunctions2| . PTFUNC2)
(|PolyGroebner| . PGROEB)
(|Polynomial| . POLY)
(|PolynomialAN2Expression| . PAN2EXPR)
(|PolynomialComposition| . PCOMP)
(|PolynomialDecomposition| . PDECOMP)
(|PolynomialFunctions2| . POLY2)
(|PolynomialIdeals| . IDEAL)
(|PolynomialPackageForCurve| . PLPKCRV)
(|PolynomialToUnivariatePolynomial| . POLY2UP)
(|PositiveInteger| . PI)
(|PowerSeriesLimitPackage| . LIMITPS)
(|PrimeField| . PF)
(|PrimitiveArrayFunctions2| . PRIMARR2)
(|PrintPackage| . PRINT)
(|ProjectiveAlgebraicSetPackage| . PRJALGPK)
(|ProjectivePlane| . PROJPL)
(|ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField| . PROJPLPS)
(|ProjectiveSpace| . PROJSP)
(|PseudoAlgebraicClosureOfAlgExtOfRationalNumber| . PACEXT)
(|QuadraticForm| . QFORM)
(|QuasiComponentPackage| . QCMPACK)
(|Quaternion| . QUAT)
(|QuaternionCategoryFunctions2| . QUATCT2)
(|QueryEquation| . QEQUAT)
(|Queue| . QUEUE)
(|QuotientFieldCategoryFunctions2| . QFCAT2)
(|RadicalEigenPackage| . REP)
(|RadicalSolvePackage| . SOLVERAD)
(|RadixExpansion| . RADIX)
(|RadixUtilities| . RADUTIL)
(|RandomNumberSource| . RANDSRC)
(|RationalFunction| . RF)

```

```

(|RationalFunctionDefiniteIntegration| . DEFINTRF)
(|RationalFunctionFactor| . RFFACT)
(|RationalFunctionFactorizer| . RFFACTOR)
(|RationalFunctionIntegration| . INTRF)
(|RationalFunctionLimitPackage| . LIMITRF)
(|RationalFunctionSign| . SIGNRF)
(|RationalFunctionSum| . SUMRF)
(|RationalRetractions| . RATRET)
(|RealClosure| . RECLoS)
(|RealPolynomialUtilitiesPackage| . POLUTIL)
(|RealZeroPackage| . REAL0)
(|RealZeroPackageQ| . REALOQ)
(|RecurrenceOperator| . RECOP)
(|RectangularMatrixCategoryFunctions2| . RMCAT2)
(|RegularSetDecompositionPackage| . RSDCMPK)
(|RegularTriangularSet| . REGSET)
(|RegularTriangularSetGcdPackage| . RSETGCD)
(|RepresentationPackage1| . REP1)
(|RepresentationPackage2| . REP2)
(|ResolveLatticeCompletion| . RESLATC)
(|RewriteRule| . RULE)
(|RightOpenIntervalRootCharacterization| . ROIRC)
(|RomanNumeral| . ROMAN)
(|RootsFindingPackage| . RFP)
(|Ruleset| . RULESET)
(|ScriptFormulaFormat| . FORMULA)
(|ScriptFormulaFormat1| . FORMULA1)
(|Segment| . SEG)
(|SegmentBinding| . SEGBIND)
(|SegmentBindingFunctions2| . SEGBIND2)
(|SegmentFunctions2| . SEG2)
(|Set| . SET)
(|SimpleAlgebraicExtensionAlgFactor| . SAEFACT)
(|SimplifyAlgebraicNumberConvertPackage| . SIMPAN)
(|SingleInteger| . SINT)
(|SmithNormalForm| . SMITH)
(|SparseUnivariatePolynomialExpressions| . SUPEXPR)
(|SparseUnivariatePolynomialFunctions2| . SUP2)
(|SpecialOutputPackage| . SPECOUT)
(|SquareFreeRegularSetDecompositionPackage| . SRDCMPK)
(|SquareFreeRegularTriangularSet| . SREGSET)
(|SquareFreeRegularTriangularSetGcdPackage| . SFRGCD)
(|SquareFreeQuasiComponentPackage| . SFQCMK)
(|Stack| . STACK)
(|Stream| . STREAM)
(|StreamFunctions1| . STREAM1)
(|StreamFunctions2| . STREAM2)
(|StreamFunctions3| . STREAM3)
(|StreamTensor| . STNSR)
(|String| . STRING)

```

```

(|SturmHabichtPackage| . SHP)
(|Symbol| . SYMBOL)
(|SymmetricGroupCombinatoricFunctions| . SGCF)
(|SystemSolvePackage| . SYSSOLP)
(|SAERationalFunctionAlgFactor| . SAERFFC)
(|Tableau| . TABLEAU)
(|TaylorSeries| . TS)
(|TaylorSolve| . UTSSOL)
(|TexFormat| . TEX)
(|TexFormat1| . TEX1)
(|TextFile| . TEXTFILE)
(|ThreeDimensionalViewport| . VIEW3D)
(|ThreeSpace| . SPACE3)
(|Timer| . TIMER)
(|TopLevelDrawFunctions| . DRAW)
(|TopLevelDrawFunctionsForAlgebraicCurves| . DRAWCURV)
(|TopLevelDrawFunctionsForCompiledFunctions| . DRAWCFUN)
(|TopLevelDrawFunctionsForPoints| . DRAWPT )
(|TopLevelThreeSpace| . TOPSP)
(|TranscendentalManipulations| . TRMANIP)
(|TransSolvePackage| . SOLVETRA)
(|Tree| . TREE)
(|TrigonometricManipulations| . TRIGMNIP)
(|UnivariateLaurentSeriesFunctions2| . ULS2)
(|UnivariateFormalPowerSeries| . UFPS)
(|UnivariateFormalPowerSeriesFunctions| . UFPS1)
(|UnivariatePolynomial| . UP)
(|UnivariatePolynomialCategoryFunctions2| . UPOLYC2)
(|UnivariatePolynomialCommonDenominator| . UPCDEN)
(|UnivariatePolynomialFunctions2| . UP2)
(|UnivariatePolynomialMultiplicationPackage| . UPMP)
(|UnivariateTaylorSeriesCZero| . UTSZ)
(|UnivariatePuisseuxSeriesFunctions2| . UPXS2)
(|UnivariateTaylorSeriesFunctions2| . UTS2)
(|UniversalSegment| . UNISEG)
(|UniversalSegmentFunctions2| . UNISEG2)
(|UserDefinedVariableOrdering| . UDVO)
(|U32Vector| . U32VEC)
(|Vector| . VECTOR)
(|VectorFunctions2| . VECTOR2)
(|ViewDefaultsPackage| . VIEWDEF)
(|Void| . VOID)
(|WuWenTsunTriangularSet| . WUTSET))
(|naglink|
  (|Asp1| . ASP1)
  (|Asp4| . ASP4)
  (|Asp6| . ASP6)
  (|Asp7| . ASP7)
  (|Asp8| . ASP8)
  (|Asp9| . ASP9)

```

```

(|Asp10| . ASP10)
(|Asp12| . ASP12)
(|Asp19| . ASP19)
(|Asp20| . ASP20)
(|Asp24| . ASP24)
(|Asp27| . ASP27)
(|Asp28| . ASP28)
(|Asp29| . ASP29)
(|Asp30| . ASP30)
(|Asp31| . ASP31)
(|Asp33| . ASP33)
(|Asp34| . ASP34)
(|Asp35| . ASP35)
(|Asp41| . ASP41)
(|Asp42| . ASP42)
(|Asp49| . ASP49)
(|Asp50| . ASP50)
(|Asp55| . ASP55)
(|Asp73| . ASP73)
(|Asp74| . ASP74)
(|Asp77| . ASP77)
(|Asp78| . ASP78)
(|Asp80| . ASP80)
(|FortranCode| . FC)
(|FortranCodePackage1| . FCPAK1)
(|FortranExpression| . FEXPR)
(|FortranMachineTypeCategory| . FMTC)
(|FortranMatrixCategory| . FMC)
(|FortranMatrixFunctionCategory| . FMFUN)
(|FortranOutputStackPackage| . FOP)
(|FortranPackage| . FORT)
(|FortranProgramCategory| . FORTCAT)
(|FortranProgram| . FORTRAN)
(|FortranFunctionCategory| . FORTFN)
(|FortranScalarType| . FST)
(|FortranType| . FT)
(|FortranTemplate| . FTEM)
(|FortranVectorFunctionCategory| . FVFUN)
(|FortranVectorCategory| . FVC)
(|MachineComplex| . MCMPLEX)
(|MachineFloat| . MFLOAT)
(|MachineInteger| . MINT)
(|MultiVariableCalculusFunctions| . MCALCFN)
(|NagDiscreteFourierTransformInterfacePackage| . NAGDIS)
(|NagEigenInterfacePackage| . NAGEIG)
(|NAGLinkSupportPackage| . NAGSP)
(|NagOptimisationInterfacePackage| . NAGOPT)
(|NagQuadratureInterfacePackage| . NAGQUA)
(|NagResultChecks| . NAGRES)
(|NagSpecialFunctionsInterfacePackage| . NAGSPE)

```

```

(|NagPolynomialRootsPackage| . NAGC02)
(|NagRootFindingPackage| . NAGC05)
(|NagSeriesSummationPackage| . NAGC06)
(|NagIntegrationPackage| . NAGD01)
(|NagOrdinaryDifferentialEquationsPackage| . NAGD02)
(|NagPartialDifferentialEquationsPackage| . NAGD03)
(|NagInterpolationPackage| . NAGE01)
(|NagFittingPackage| . NAGE02)
(|NagOptimisationPackage| . NAGE04)
(|NagMatrixOperationsPackage| . NAGF01)
(|NagEigenPackage| . NAGF02)
(|NagLinearEquationSolvingPackage| . NAGF04)
(|NagLapack| . NAGF07)
(|NagSpecialFunctionsPackage| . NAGS)
(|PackedHermitianSequence| . PACKED)
(|Result| . RESULT)
(|SimpleFortranProgram| . SFORT)
(|Switch| . SWITCH)
(|SymbolTable| . SYMTAB)
(|TemplateUtilities| . TEMUTL)
(|TheSymbolTable| . SYMS)
(|ThreeDimensionalMatrix| . M3D))
(|anna|
(|AnnaNumericalIntegrationPackage| . INTPACK)
(|AnnaNumericalOptimizationPackage| . OPTPACK)
(|AnnaOrdinaryDifferentialEquationPackage| . ODEPACK)
(|AnnaPartialDifferentialEquationPackage| . PDEPACK)
(|AttributeButtons| . ATTRBUT)
(|BasicFunctions| . BFUNCT)
(|d01ajfAnnaType| . D01AJFA)
(|d01akfAnnaType| . D01AKFA)
(|d01alfAnnaType| . D01ALFA)
(|d01amfAnnaType| . D01AMFA)
(|d01anfAnnaType| . D01ANFA)
(|d01apfAnnaType| . D01APFA)
(|d01aqfAnnaType| . D01AQFA)
(|d01asfAnnaType| . D01ASFA)
(|d01fcfAnnaType| . D01FCFA)
(|d01gbfAnnaType| . D01GBFA)
(|d01AgentsPackage| . D01AGNT)
(|d01TransformFunctionType| . D01TRNS)
(|d01WeightsPackage| . D01WGTS)
(|d02AgentsPackage| . D02AGNT)
(|d02bbfAnnaType| . D02BBFA)
(|d02bhfAnnaType| . D02BHFA)
(|d02cjfAnnaType| . D02CJFA)
(|d02ejfAnnaType| . D02EJFA)
(|d03AgentsPackage| . D03AGNT)
(|d03eefAnnaType| . D03EEFA)
(|d03fafAnnaType| . D03FAFA)

```

```

(|e04AgentsPackage| . E04AGNT)
(|e04dgmAnnaType| . E04DGFA)
(|e04fdfAnnaType| . E04FDFA)
(|e04gcfAnnaType| . E04GCFA)
(|e04jafAnnaType| . E04JAFA)
(|e04mbfAnnaType| . E04MBFA)
(|e04nafAnnaType| . E04NAFA)
(|e04ucfAnnaType| . E04UCFA)
(|ExpertSystemContinuityPackage| . ESCONT)
(|ExpertSystemContinuityPackage1| . ESCONT1)
(|ExpertSystemToolsPackage| . ESTOOLS)
(|ExpertSystemToolsPackage1| . ESTOOLS1)
(|ExpertSystemToolsPackage2| . ESTOOLS2)
(|NumericalIntegrationCategory| . NUMINT)
(|NumericalIntegrationProblem| . NIPROB)
(|NumericalODEProblem| . ODEPROB)
(|NumericalOptimizationCategory| . OPTCAT)
(|NumericalOptimizationProblem| . OPTPROB)
(|NumericalPDEProblem| . PDEPROB)
(|ODEIntensityFunctionsTable| . ODEIFTBL)
(|IntegrationFunctionsTable| . INTFTBL)
(|OrdinaryDifferentialEquationsSolverCategory| . ODECAT)
(|PartialDifferentialEquationsSolverCategory| . PDECAT)
(|RoutinesTable| . ROUTINE))
(|categories|
(|AbelianGroup| . ABELGRP)
(|AbelianMonoid| . ABELMON)
(|AbelianMonoidRing| . AMR)
(|AbelianSemiGroup| . ABELSG)
(|AffineSpaceCategory| . AFSPCAT)
(|Aggregate| . AGG)
(|Algebra| . ALGEBRA)
(|AlgebraicallyClosedField| . ACF)
(|AlgebraicallyClosedFunctionSpace| . ACFS)
(|ArcHyperbolicFunctionCategory| . AHYP)
(|ArcTrigonometricFunctionCategory| . ATRIG)
(|AssociationListAggregate| . ALAGG)
(|AttributeRegistry| . ATTREG)
(|BagAggregate| . BGAGG)
(|BasicType| . BASTYPE)
(|BiModule| . BMODULE)
(|BinaryRecursiveAggregate| . BRAGG)
(|BinaryTreeCategory| . BTCAT)
(|BitAggregate| . BTAGG)
(|BlowUpMethodCategory| . BLMETCT)
(|CachableSet| . CACHSET)
(|CancellationAbelianMonoid| . CABMON)
(|CharacteristicNonZero| . CHARNZ)
(|CharacteristicZero| . CHARZ)
(|CoercibleTo| . KOERCE)

```



```

(|Collection| . CLAGG)
(|CombinatorialFunctionCategory| . CFCAT)
(|CombinatorialOpsCategory| . COMBOPC)
(|CommutativeRing| . COMRING)
(|ComplexCategory| . COMPCAT)
(|ConvertibleTo| . KONVERT)
(|DequeueAggregate| . DQAGG)
(|DesingTreeCategory| . DSTRCAT)
(|Dictionary| . DIAGG)
(|DictionaryOperations| . DIOPS)
(|DifferentialExtension| . DIFEXT)
(|DifferentialPolynomialCategory| . DPOLCAT)
(|DifferentialRing| . DIFRING)
(|DifferentialVariableCategory| . DVARCAT)
(|DirectProductCategory| . DIRPCAT)
(|DivisionRing| . DIVRING)
(|DivisorCategory| . DIVCAT)
(|DoublyLinkedAggregate| . DLAGG)
(|ElementaryFunctionCategory| . ELEMFUN)
(|Eltable| . ELTAB)
(|EltableAggregate| . ELTAGG)
(|EntireRing| . ENTIRER)
(|EuclideanDomain| . EUCDOM)
(|Evalable| . EVALAB)
(|ExpressionSpace| . ES)
(|ExtensibleLinearAggregate| . ELAGG)
(|ExtensionField| . XF)
(|Field| . FIELD)
(|FieldOfPrimeCharacteristic| . FPC)
(|Finite| . FINITE)
(|FileCategory| . FILECAT)
(|FileNameCategory| . FNCAT)
(|FiniteAbelianMonoidRing| . FAMR)
(|FiniteAlgebraicExtensionField| . FAXF)
(|FiniteDivisorCategory| . FDIVCAT)
(|FiniteFieldCategory| . FFIELDC)
(|FiniteLinearAggregate| . FLAGG)
(|FiniteRankNonAssociativeAlgebra| . FINAALG)
(|FiniteRankAlgebra| . FINRALG)
(|FiniteSetAggregate| . FSAGG)
(|FloatingPointSystem| . FPS)
(|FramedAlgebra| . FRAMALG)
(|FramedNonAssociativeAlgebra| . FRNAALG)
(|FramedNonAssociativeAlgebraFunctions2| . FRNAAF2)
(|FreeAbelianMonoidCategory| . FAMONC)
(|FreeLieAlgebra| . FLALG)
(|FreeModuleCat| . FMCAT)
(|FullyEvalableOver| . FEVALAB)
(|FullyLinearlyExplicitRingOver| . FLINEXP)
(|FullyPatternMatchable| . FPATMAB)

```

```

(|FullyRetractableTo| . FRETRCT)
(|FunctionFieldCategory| . FFCAT)
(|FunctionSpace| . FS)
(|GcdDomain| . GCDDOM)
(|GradedAlgebra| . GRALG)
(|GradedModule| . GRMOD)
(|Group| . GROUP)
(|HomogeneousAggregate| . HOAGG)
(|HyperbolicFunctionCategory| . HYPCAT)
(|IndexedAggregate| . IXAGG)
(|IndexedDirectProductCategory| . IDPC)
(|InfinitelyClosePointCategory| . INFCLCT)
(|InnerEvalable| . IEVALAB)
(|IntegerNumberSystem| . INS)
(|IntegralDomain| . INTDOM)
(|IntervalCategory| . INTCAT)
(|KeyedDictionary| . KDAGG)
(|LazyStreamAggregate| . LZSTAGG)
(|LeftAlgebra| . LALG)
(|LeftModule| . LMODULE)
(|LieAlgebra| . LIECAT)
(|LinearAggregate| . LNAGG)
(|LinearlyExplicitRingOver| . LINEXP)
(|LinearOrdinaryDifferentialOperatorCategory| . LODOCAT)
(|LiouvillianFunctionCategory| . LFCAT)
(|ListAggregate| . LSAGG)
(|LocalPowerSeriesCategory| . LOCPOWC)
(|Logic| . LOGIC)
(|MatrixCategory| . MATCAT)
(|Module| . MODULE)
(|Monad| . MONAD)
(|MonadWithUnit| . MONADWU)
(|Monoid| . MONOID)
(|MonogenicAlgebra| . MONOGEN)
(|MonogenicLinearOperator| . MLO)
(|MultiDictionary| . MDAGG)
(|MultisetAggregate| . MSETAGG)
(|MultivariateTaylorSeriesCategory| . MTSCAT)
(|NonAssociativeAlgebra| . NAALG)
(|NonAssociativeRing| . NASRING)
(|NonAssociativeRng| . NARNG)
(|NormalizedTriangularSetCategory| . NTSCAT)
(|Object| . OBJECT)
(|OctonionCategory| . OC)
(|OneDimensionalArrayAggregate| . A1AGG)
(|OpenMath| . OM)
(|OrderedAbelianGroup| . OAGROUP)
(|OrderedAbelianMonoid| . OAMON)
(|OrderedAbelianMonoidSup| . OAMONS)
(|OrderedAbelianSemiGroup| . OASGP)

```

```

(|OrderedCancellationAbelianMonoid| . OCAMON)
(|OrderedFinite| . ORDFIN)
(|OrderedIntegralDomain| . OINTDOM)
(|OrderedMonoid| . ORDMON)
(|OrderedMultisetAggregate| . OMSAGG)
(|OrderedRing| . ORDRING)
(|OrderedSet| . ORDSET)
(|PAdicIntegerCategory| . PADICCT)
(|PartialDifferentialRing| . PDRING)
(|PartialTranscendentalFunctions| . PTRANFN)
(|Patternable| . PATAB)
(|PatternMatchable| . PATMAB)
(|PermutationCategory| . PERMCAT)
(|PlacesCategory| . PLACESC)
(|PlottablePlaneCurveCategory| . PPCURVE)
(|PlottableSpaceCurveCategory| . PSCURVE)
(|PointCategory| . PTCAT)
(|PolynomialCategory| . POLYCAT)
(|PolynomialFactorizationExplicit| . PFECAT)
(|PolynomialSetCategory| . PSETCAT)
(|PowerSeriesCategory| . PSCAT)
(|PrimitiveFunctionCategory| . PRIMCAT)
(|PrincipalIdealDomain| . PID)
(|PriorityQueueAggregate| . PRQAGG)
(|ProjectiveSpaceCategory| . PRSPCAT)
(|PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory| . PACEXTC)
(|PseudoAlgebraicClosureOfFiniteField| . PACOFF)
(|PseudoAlgebraicClosureOfFiniteFieldCategory| . PACFFC)
(|PseudoAlgebraicClosureOfPerfectFieldCategory| . PACPERC)
(|PseudoAlgebraicClosureOfRationalNumber| . PACRAT)
(|PseudoAlgebraicClosureOfRationalNumberCategory| . PACRATC)
(|QuaternionCategory| . QUATCAT)
(|QueueAggregate| . QUAGG)
(|QuotientFieldCategory| . QFCAT)
(|RadicalCategory| . RADCAT)
(|RealClosedField| . RCFIELD)
(|RealConstant| . REAL)
(|RealNumberSystem| . RNS)
(|RealRootCharacterizationCategory| . RRCC)
(|RectangularMatrixCategory| . RMATCAT)
(|RecursiveAggregate| . RCAGG)
(|RecursivePolynomialCategory| . RPOLCAT)
(|RegularChain| . RGCHAIN)
(|RegularTriangularSetCategory| . RSETCAT)
(|RetractableTo| . RETRACT)
(|RightModule| . RMODULE)
(|Ring| . RING)
(|Rng| . RNG)
(|SegmentCategory| . SEGCAT)
(|SegmentExpansionCategory| . SEGXCAT)

```

```

(|SemiGroup| . SGROUP)
(|SetAggregate| . SETAGG)
(|SetCategory| . SETCAT)
(|SetCategoryWithDegree| . SETCATD)
(|ExpressionCategory| . SEXCAT)
(|SpecialFunctionCategory| . SPFCAT)
(|SquareFreeNormalizedTriangularSetCategory| . SNTSCAT)
(|SquareFreeRegularTriangularSetCategory| . SFRTCAT)
(|SquareMatrixCategory| . SMATCAT)
(|StackAggregate| . SKAGG)
(|StepThrough| . STEP)
(|StreamAggregate| . STAGG)
(|StringAggregate| . SRAGG)
(|StringCategory| . STRICAT)
(|StructuralConstantsPackage| . SCPKG)
(|TableAggregate| . TBAGG)
(|ThreeSpaceCategory| . SPACEC)
(|TranscendentalFunctionCategory| . TRANFUN)
(|TriangularSetCategory| . TSETCAT)
(|TrigonometricFunctionCategory| . TRIGCAT)
(|TwoDimensionalArrayCategory| . ARR2CAT)
(|Type| . TYPE)
(|UnaryRecursiveAggregate| . URAGG)
(|UniqueFactorizationDomain| . UFD)
(|UnivariateLaurentSeriesCategory| . ULSCAT)
(|UnivariateLaurentSeriesConstructorCategory| . ULSCCAT)
(|UnivariatePolynomialCategory| . UPOLYC)
(|UnivariatePowerSeriesCategory| . UPSCAT)
(|UnivariatePuisseuxSeriesCategory| . UPXSCAT)
(|UnivariatePuisseuxSeriesConstructorCategory| . UPXSCCA)
(|UnivariateSkewPolynomialCategory| . OREPCAT)
(|UnivariateTaylorSeriesCategory| . UTSCAT)
(|VectorCategory| . VECTCAT)
(|VectorSpace| . VSPACE)
(|XAlgebra| . XALG)
(|XFreeAlgebra| . XFALG)
(|XPolynomialsCat| . XPOLYC)
(|ZeroDimensionalSolvePackage| . ZDSOLVE))
(|Hidden|
  (|AlgebraicFunction| . AF)
  (|AlgebraicFunctionField| . ALGFF)
  (|AlgebraicHermiteIntegration| . INTHERAL)
  (|AlgebraicIntegrate| . INTALG)
  (|AlgebraicIntegration| . INTAF)
  (|AnonymousFunction| . ANON)
  (|AntiSymm| . ANTISYM)
  (|ApplyRules| . APPRULE)
  (|ApplyUnivariateSkewPolynomial| . APPLYORE)
  (|ArrayStack| . ASTACK)
  (|AssociatedEquations| . ASSOCEQ)

```

```

(|AssociationList| . ALIST)
(|Automorphism| . AUTOMOR)
(|BalancedFactorisation| . BALFACT)
(|BalancedPAdicInteger| . BPADIC)
(|BalancedPAdicRational| . BPADICRT)
(|BezoutMatrix| . BEZOUT)
(|BoundIntegerRoots| . BOUNDZRO)
(|BrillhartTests| . BRILL)
(|ChangeOfVariable| . CHVAR)
(|CharacteristicPolynomialInMonogenicalAlgebra| . CPIMA)
(|ChineseRemainderToolsForIntegralBases| . IBACHIN)
(|CoerceVectorMatrixPackage| . CVMP)
(|CombinatorialFunction| . COMBF)
(|CommonOperators| . COMMONOP)
(|CommuteUnivariatePolynomialCategory| . COMMUPC)
(|ComplexIntegerSolveLinearPolynomialEquation| . CINTSLPE)
(|ComplexPattern| . COMPLPAT)
(|ComplexPatternMatch| . CPMATCH)
(|ComplexRootFindingPackage| . CRFP)
(|ConstantLODE| . ODECONST)
(|CyclicStreamTools| . CSTTOOLS)
(|CyclotomicPolynomialPackage| . CYCLOTOM)
(|DefiniteIntegrationTools| . DFINTTLS)
(|DegreeReductionPackage| . DEGRED)
(|DeRhamComplex| . DERHAM)
(|DifferentialSparseMultivariatePolynomial| . DSMP)
(|DirectProduct| . DIRPROD)
(|DirectProductMatrixModule| . DPMM)
(|DirectProductModule| . DPMO)
(|DiscreteLogarithmPackage| . DLP)
(|DistributedMultivariatePolynomial| . DMP)
(|DoubleResultantPackage| . DBLRESP)
(|DrawOptionFunctions0| . DROPT0)
(|DrawOptionFunctions1| . DROPT1)
(|ElementaryFunction| . EF)
(|ElementaryFunctionsUnivariateLaurentSeries| . EFULS)
(|ElementaryFunctionsUnivariatePuisseuxSeries| . EFUPXS)
(|ElementaryIntegration| . INTEF)
(|ElementaryRischDE| . RDEEF)
(|ElementaryRischDESystem| . RDEEFS)
(|EllipticFunctionsUnivariateTaylorSeries| . ELFUTS)
(|EqTable| . EQTBL)
(|EuclideanModularRing| . EMR)
(|EvaluateCycleIndicators| . EVALCYC)
(|ExponentialExpansion| . EXPEXPAN)
(|ExponentialOfUnivariatePuisseuxSeries| . EXPUPXS)
(|ExpressionSpaceFunctions1| . ES1)
(|ExpressionTubePlot| . EXPRTUBE)
(|ExtAlgBasis| . EAB)
(|FactoredFunctions| . FACTFUNC)

```

```

(|FactoredFunctionUtilities| . FRUTIL)
(|FactoringUtilities| . FACUTIL)
(|FGLMIfCanPackage| . FGLMICPK)
(|FindOrderFinite| . FORDER)
(|FiniteDivisor| . FDIV)
(|FiniteFieldCyclicGroupExtension| . FFCGX)
(|FiniteFieldCyclicGroupExtensionByPolynomial| . FFCGP)
(|FiniteFieldExtension| . FFX)
(|FiniteFieldExtensionByPolynomial| . FFP)
(|FiniteFieldFunctions| . FFF)
(|FiniteFieldNormalBasisExtension| . FFNBX)
(|FiniteFieldNormalBasisExtensionByPolynomial| . FFNBP)
(|FiniteFieldPolynomialPackage| . FFPOLY)
(|FiniteFieldSolveLinearPolynomialEquation| . FFSLPE)
(|FormalFraction| . FORMAL)
(|FourierComponent| . FCOMP)
(|FractionalIdeal| . FRIDEAL)
(|FramedModule| . FRMOD)
(|FreeAbelianGroup| . FAGROUP)
(|FreeAbelianMonoid| . FAMONOID)
(|FreeGroup| . FGROUP)
(|FreeModule| . FM)
(|FreeModule1| . FM1)
(|FreeMonoid| . FMONOID)
(|FunctionalSpecialFunction| . FSPECF)
(|FunctionCalled| . FUNCTION)
(|FunctionFieldIntegralBasis| . FFINTBAS)
(|FunctionSpaceReduce| . FSRED)
(|FunctionSpaceToUnivariatePowerSeries| . FS2UPS)
(|FunctionSpaceToExponentialExpansion| . FS2EXXPX)
(|FunctionSpaceUnivariatePolynomialFactor| . FSUPFACT)
(|GaloisGroupFactorizationUtilities| . GALFACTU)
(|GaloisGroupFactorizer| . GALFACT)
(|GaloisGroupPolynomialUtilities| . GALPOLYU)
(|GaloisGroupUtilities| . GALUTIL)
(|GeneralHenselPackage| . GHENSEL)
(|GeneralDistributedMultivariatePolynomial| . GDMP)
(|GeneralPolynomialGcdPackage| . GENPGCD)
(|GeneralSparseTable| . GSTBL)
(|GenericNonAssociativeAlgebra| . GCNAALG)
(|GenExEuclid| . GENEEZ)
(|GeneralizedMultivariateFactorize| . GENMFACT)
(|GeneralModulePolynomial| . GMPOL)
(|GeneralPolynomialSet| . GPOLSET)
(|GeneralTriangularSet| . GTSET)
(|GenUFactorize| . GENUFACT)
(|GenusZeroIntegration| . INTGO)
(|GosperSummationMethod| . GOSPER)
(|GraphImage| . GRIMAGE)
(|GrayCode| . GRAY)

```

```

(|GroebnerInternalPackage| . GBINTERN)
(|GroebnerSolve| . GROEBSOL)
(|GuessOptionFunctions0| . GOPT0)
(|HashTable| . HASHTBL)
(|Heap| . HEAP)
(|HeuGcd| . HEUGCD)
(|HomogeneousDistributedMultivariatePolynomial| . HDMP)
(|HyperellipticFiniteDivisor| . HELLFDIV)
(|IncrementingMaps| . INCRMAPS)
(|IndexedBits| . IBITS)
(|IndexedDirectProductAbelianGroup| . IDPAG)
(|IndexedDirectProductAbelianMonoid| . IDPAM)
(|IndexedDirectProductObject| . IDPO)
(|IndexedDirectProductOrderedAbelianMonoid| . IDPOAM)
(|IndexedDirectProductOrderedAbelianMonoidSup| . IDPOAMS)
(|IndexedExponents| . INDE)
(|IndexedFlexibleArray| . IFARRAY)
(|IndexedList| . ILIST)
(|IndexedMatrix| . IMATRIX)
(|IndexedOneDimensionalArray| . IARRAY1)
(|IndexedString| . ISTRING)
(|IndexedTwoDimensionalArray| . IARRAY2)
(|IndexedVector| . IVECTOR)
(|InnerAlgFactor| . IALGFACT)
(|InnerAlgebraicNumber| . IAN)
(|InnerCommonDenominator| . ICDEN)
(|InnerFiniteField| . IFF)
(|InnerFreeAbelianMonoid| . IFAMON)
(|InnerIndexedTwoDimensionalArray| . IIARRAY2)
(|InnerMatrixLinearAlgebraFunctions| . IMATLIN)
(|InnerMatrixQuotientFieldFunctions| . IMATQF)
(|InnerModularGcd| . INMODGCD)
(|InnerMultFact| . INNMFAC)
(|InnerNormalBasisFieldFunctions| . INBFF)
(|InnerNumericEigenPackage| . INEP)
(|InnerNumericFloatSolvePackage| . INFSP)
(|InnerPAdicInteger| . IPADIC)
(|InnerPolySign| . INPSIGN)
(|InnerPolySum| . ISUMP)
(|InnerPrimeField| . IPF)
(|InnerSparseUnivariatePowerSeries| . ISUPS)
(|InnerTable| . INTABL)
(|InnerTaylorSeries| . ITAYLOR)
(|InnerTrigonometricManipulations| . ITRIGMNP)
(|InputForm| . INFORM)
(|InputFormFunctions1| . INFORM1)
(|IntegerBits| . INTBIT)
(|IntegerFactorizationPackage| . INTFACT)
(|IntegerMod| . ZMOD)
(|IntegerSolveLinearPolynomialEquation| . INTSLPE)

```

```

(|IntegralBasisPolynomialTools| . IBPTOOLS)
(|IntegralBasisTools| . IBATool)
(|IntegrationResult| . IR)
(|IntegrationTools| . INTTOOLS)
(|InternalPrintPackage| . IPRNTPK)
(|InternalRationalUnivariateRepresentationPackage| . IRURPK)
(|IrredPolyOverFiniteField| . IRREDFFX)
(|Kernel| . KERNEL)
(|Kovacic| . KOVACIC)
(|LaurentPolynomial| . LAUPOL)
(|LeadingCoefDetermination| . LEADCDET)
(|LexTriangularPackage| . LEXTRIPK)
(|LieExponentials| . LEXP)
(|LiePolynomial| . LPOLY)
(|LinearDependence| . LINDEP)
(|LinearOrdinaryDifferentialOperatorFactorizer| . LODOF)
(|LinearOrdinaryDifferentialOperator1| . LOD01)
(|LinearOrdinaryDifferentialOperator2| . LOD02)
(|LinearOrdinaryDifferentialOperatorsOps| . LOD0OPS)
(|LinearPolynomialEquationByFractions| . LPEFRAC)
(|LinGroebnerPackage| . LGROBP)
(|LiouvillianFunction| . LF)
(|ListMonoidOps| . LMOPS)
(|ListMultiDictionary| . LMDICT)
(|LocalAlgebra| . LA)
(|Localize| . LO)
(|LyndonWord| . LWORD)
(|Magma| . MAGMA)
(|MakeBinaryCompiledFunction| . MKBCFUNC)
(|MakeCachableSet| . MKCHSET)
(|MakeUnaryCompiledFunction| . MKUCFUNC)
(|MappingPackageInternalHacks1| . MAPHACK1)
(|MappingPackageInternalHacks2| . MAPHACK2)
(|MappingPackageInternalHacks3| . MAPHACK3)
(|MeshCreationRoutinesForThreeDimensions| . MESH)
(|ModMonic| . MODMON)
(|ModularField| . MODFIELD)
(|ModularHermitianRowReduction| . MHROWRED)
(|ModularRing| . MODRING)
(|ModuleMonomial| . MODMONOM)
(|MoebiusTransform| . MOEBIUS)
(|MonoidRing| . MRING)
(|MonomialExtensionTools| . MONOTOOL)
(|MPolyCatPolyFactorizer| . MPCPF)
(|MPolyCatFunctions3| . MPC3)
(|MRationalFactorize| . MRATFAC)
(|MultipleMap| . MMAP)
(|MultivariateLifting| . MLIFT)
(|MultivariateSquareFree| . MULTSQFR)
(|HomogeneousDirectProduct| . HDP)

```



```

(|NewSparseMultivariatePolynomial| . NSMP)
(|NewSparseUnivariatePolynomial| . NSUP)
(|NewSparseUnivariatePolynomialFunctions2| . NSUP2)
(|NonCommutativeOperatorDivision| . NCODIV)
(|NewtonInterpolation| . NEWTON)
(|None| . NONE)
(|NonLinearFirstOrderODESolver| . NODE1)
(|NonLinearSolvePackage| . NLINSOL)
(|NormRetractPackage| . NORMRETR)
(|NPCoef| . NPCOEF)
(|NumberFormats| . NUMFMT)
(|NumberFieldIntegralBasis| . NFINTBAS)
(|NumericTubePlot| . NUMTUBE)
(|ODEIntegration| . ODEINT)
(|ODETools| . ODETOOLS)
(|Operator| . OP)
(|OppositeMonogenicLinearOperator| . OML0)
(|OrderedDirectProduct| . ODP)
(|OrderedFreeMonoid| . OFMONOID)
(|OrderedVariableList| . OVAR)
(|OrderingFunctions| . ORDFUNS)
(|OrderlyDifferentialPolynomial| . ODPOL)
(|OrderlyDifferentialVariable| . ODVAR)
(|OrdinaryWeightedPolynomials| . OWP)
(|OutputForm| . OUTFORM)
(|PadeApproximants| . PADE)
(|PAdicInteger| . PADIC)
(|PAdicRational| . PADICRAT)
(|PAdicRationalConstructor| . PADICRC)
(|PAdicWildFunctionFieldIntegralBasis| . PWFFINTB)
(|ParadoxicalCombinatorsForStreams| . YSTREAM)
(|ParametricLinearEquations| . PLEQN)
(|PartialFractionPackage| . PFRPAC)
(|Partition| . PRITION)
(|Pattern| . PATTERN)
(|PatternFunctions1| . PATTERN1)
(|PatternMatchFunctionSpace| . PMFS)
(|PatternMatchIntegerNumberSystem| . PMINS)
(|PatternMatchIntegration| . INTPM)
(|PatternMatchKernel| . PMKERNEL)
(|PatternMatchListAggregate| . PMLSAGG)
(|PatternMatchListResult| . PATLRES)
(|PatternMatchPolynomialCategory| . PMPLCAT)
(|PatternMatchPushDown| . PMDOWN)
(|PatternMatchQuotientFieldCategory| . PMQFCAT)
(|PatternMatchResult| . PATRES)
(|PatternMatchSymbol| . PMSYM)
(|PatternMatchTools| . PMTOOLS)
(|PlaneAlgebraicCurvePlot| . ACPLLOT)
(|Plot| . PLOT)

```

```

(|PlotFunctions1| . PLOT1)
(|PlotTools| . PLOTTOOL)
(|Plot3D| . PLOT3D)
(|PoincareBirkhoffWittLyndonBasis| . PBWLB)
(|Point| . POINT)
(|PointsOfFiniteOrder| . PFO)
(|PointsOfFiniteOrderRational| . PFOQ)
(|PointsOfFiniteOrderTools| . PFOTOOLS)
(|PointPackage| . PTPACK)
(|PolToPol| . POLTOPOL)
(|PolynomialCategoryLifting| . POLYLIFT)
(|PolynomialCategoryQuotientFunctions| . POLYCATQ)
(|PolynomialFactorizationByRecursion| . PFBR)
(|PolynomialFactorizationByRecursionUnivariate| . PFBRU)
(|PolynomialGcdPackage| . PGCD)
(|PolynomialInterpolation| . PINTERP)
(|PolynomialInterpolationAlgorithms| . PINTERPA)
(|PolynomialNumberTheoryFunctions| . PNTHEORY)
(|PolynomialRing| . PR)
(|PolynomialRoots| . POLYROOT)
(|PolynomialSetUtilitiesPackage| . PSETPK)
(|PolynomialSolveByFormulas| . SOLVEFOR)
(|PolynomialSquareFree| . PSQFR)
(|PrecomputedAssociatedEquations| . PREASSOC)
(|PrimitiveArray| . PRIMARR)
(|PrimitiveElement| . PRIMELT)
(|PrimitiveRatDE| . ODEPRIM)
(|PrimitiveRatRicDE| . ODEPRRIC)
(|Product| . PRODUCT)
(|PseudoRemainderSequence| . PRS)
(|PseudoLinearNormalForm| . PSEUDLIN)
(|PureAlgebraicIntegration| . INTPAF)
(|PureAlgebraicLODE| . ODEPAL)
(|PushVariables| . PUSHVAR)
(|QuasiAlgebraicSet| . QALGSET)
(|QuasiAlgebraicSet2| . QALGSET2)
(|RadicalFunctionField| . RADFF)
(|RandomDistributions| . RDIST)
(|RandomFloatDistributions| . RFDIST)
(|RandomIntegerDistributions| . RIDIST)
(|RationalFactorize| . RATFACT)
(|RationalIntegration| . INTRAT)
(|RationalInterpolation| . RINTERP)
(|RationalLODE| . ODERAT)
(|RationalRicDE| . ODERTRIC)
(|RationalUnivariateRepresentationPackage| . RURPK)
(|RealSolvePackage| . REALSOLV)
(|RectangularMatrix| . RMATRIX)
(|ReducedDivisor| . RDIV)
(|ReduceLODE| . ODERED)

```

```

(|ReductionOfOrder| . REDORDER)
(|Reference| . REF)
(|RepeatedDoubling| . REPDB)
(|RepeatedSquaring| . REPSQ)
(|ResidueRing| . RESRING)
(|RetractSolvePackage| . RETSOL)
(|RuleCalled| . RULECOLD)
(|SetOfMIntegersInOneToN| . SETMN)
(|SExpression| . SEX)
(|SExpressionOf| . SEXOF)
(|SequentialDifferentialPolynomial| . SDPOL)
(|SequentialDifferentialVariable| . SDVAR)
(|SimpleAlgebraicExtension| . SAE)
(|SingletonAsOrderedSet| . SAOS)
(|SortedCache| . SCACHE)
(|SortPackage| . SORTPAK)
(|SparseMultivariatePolynomial| . SMP)
(|SparseMultivariateTaylorSeries| . SMTS)
(|SparseTable| . STBL)
(|SparseUnivariatePolynomial| . SUP)
(|SparseUnivariateSkewPolynomial| . ORESUP)
(|SparseUnivariateLaurentSeries| . SULS)
(|SparseUnivariatePuisseuxSeries| . SUPXS)
(|SparseUnivariateTaylorSeries| . SUTS)
(|SplitHomogeneousDirectProduct| . SHDP)
(|SplittingNode| . SPLNODE)
(|SplittingTree| . SPLTREE)
(|SquareMatrix| . SQMATRIX)
(|Stack| . STACK)
(|StorageEfficientMatrixOperations| . MATSTOR)
(|StreamInfiniteProduct| . STINPROD)
(|StreamTaylorSeriesOperations| . STTAYLOR)
(|StreamTranscendentalFunctions| . STTF)
(|StreamTranscendentalFunctionsNonCommutative| . STTFNC)
(|StringTable| . STRTBL)
(|SubResultantPackage| . SUBRESP)
(|SubSpace| . SUBSPACE)
(|SubSpaceComponentProperty| . COMPPROP)
(|SuchThat| . SUCH)
(|SupFractionFactorizer| . SUPFRACF)
(|SymmetricFunctions| . SYMFUNC)
(|SymmetricPolynomial| . SYMPOLY)
(|SystemODESolver| . ODESYS)
(|Table| . TABLE)
(|TableauxBumpers| . TABLBUMP)
(|TabulatedComputationPackage| . TBCMPPK)
(|TangentExpansions| . TANEXP)
(|ToolsForSign| . TOOLSIGN)
(|TranscendentalHermiteIntegration| . INTHERTR)
(|TranscendentalIntegration| . INTTR)

```

```

(|TranscendentalRischDE| . RDETR)
(|TranscendentalRischDESystem| . RDETRS)
(|TransSolvePackageService| . SOLVESER)
(|TriangularMatrixOperations| . TRIMAT)
(|TubePlot| . TUBE)
(|TubePlotTools| . TUBETOOL)
(|Tuple| . TUPLE)
(|TwoDimensionalArray| . ARRAY2)
(|TwoDimensionalPlotClipping| . CLIP)
(|TwoDimensionalViewport| . VIEW2D)
(|TwoFactorize| . TWOFACT)
(|UnivariateFactorize| . UNIFACT)
(|UnivariateLaurentSeries| . ULS)
(|UnivariateLaurentSeriesConstructor| . ULSCONS)
(|UnivariatePolynomialDecompositionPackage| . UPDECOMP)
(|UnivariatePolynomialDivisionPackage| . UPDIVP)
(|UnivariatePolynomialSquareFree| . UPSQFREE)
(|UnivariatePuisseuxSeries| . UPXS)
(|UnivariatePuisseuxSeriesConstructor| . UPXSCONS)
(|UnivariatePuisseuxSeriesWithExponentialSingularity| . UPXSSING)
(|UnivariateSkewPolynomial| . OREUP)
(|UnivariateSkewPolynomialCategoryOps| . OREPCTO)
(|UnivariateTaylorSeries| . UTS)
(|UnivariateTaylorSeriesODESolver| . UTSODE)
(|UserDefinedPartialOrdering| . UDPO)
(|UTSodeTools| . UTSODETL)
(|Variable| . VARIABLE)
(|ViewportPackage| . VIEW)
(|WeierstrassPreparation| . WEIER)
(|WeightedPolynomials| . WP)
(|WildFunctionFieldIntegralBasis| . WFFINTBS)
(|XDistributedPolynomial| . XDPOLY)
(|XExponentialPackage| . XEXPPKG)
(|XPBWPolynomial| . XPBWPOLY)
(|XPolynomial| . XPOLY)
(|XPolynomialRing| . XPR)
(|XRecursivePolynomial| . XRPOLY))
(|defaults|
  (|AbelianGroup&| . ABELGRP-)
  (|AbelianMonoid&| . ABELMON-)
  (|AbelianMonoidRing&| . AMR-)
  (|AbelianSemiGroup&| . ABELSG-)
  (|Aggregate&| . AGG-)
  (|Algebra&| . ALGEBRA-)
  (|AlgebraicallyClosedField&| . ACF-)
  (|AlgebraicallyClosedFunctionSpace&| . ACFS-)
  (|ArcTrigonometricFunctionCategory&| . ATRIG-)
  (|BagAggregate&| . BGAGG-)
  (|BasicType&| . BASTYPE-)
  (|BinaryRecursiveAggregate&| . BRAGG-)

```

```

(|BinaryTreeCategory&| . BTCAT-)
(|BitAggregate&| . BTAGG-)
(|Collection&| . CLAGG-)
(|ComplexCategory&| . COMPCAT-)
(|Dictionary&| . DIAGG-)
(|DictionaryOperations&| . DIOPS-)
(|DifferentialExtension&| . DIFEXT-)
(|DifferentialPolynomialCategory&| . DPOLCAT-)
(|DifferentialRing&| . DIFRING-)
(|DifferentialVariableCategory&| . DVARCAT-)
(|DirectProductCategory&| . DIRPCAT-)
(|DivisionRing&| . DIVRING-)
(|ElementaryFunctionCategory&| . ELEMFUN-)
(|EltableAggregate&| . ELTAGG-)
(|EuclideanDomain&| . EUCDOM-)
(|Evalable&| . EVALAB-)
(|ExpressionSpace&| . ES-)
(|ExtensibleLinearAggregate&| . ELAGG-)
(|ExtensionField&| . XF-)
(|Field&| . FIELD-)
(|FieldOfPrimeCharacteristic&| . FPC-)
(|FiniteAbelianMonoidRing&| . FAMR-)
(|FiniteAlgebraicExtensionField&| . FAXF-)
(|FiniteDivisorCategory&| . FDIVCAT-)
(|FiniteFieldCategory&| . FFIELDC-)
(|FiniteLinearAggregate&| . FLAGG-)
(|FiniteSetAggregate&| . FSAGG-)
(|FiniteRankAlgebra&| . FINRALG-)
(|FiniteRankNonAssociativeAlgebra&| . FINAALG-)
(|FloatingPointSystem&| . FPS-)
(|FramedAlgebra&| . FRAMALG-)
(|FramedNonAssociativeAlgebra&| . FRNAALG-)
(|FullyEvalableOver&| . FEVALAB-)
(|FullyLinearlyExplicitRingOver&| . FLINEXP-)
(|FullyRetractableTo&| . FRETRCT-)
(|FunctionFieldCategory&| . FFCAT-)
(|FunctionSpace&| . FS-)
(|GcdDomain&| . GCDDOM-)
(|GradedAlgebra&| . GRALG-)
(|GradedModule&| . GRMOD-)
(|Group&| . GROUP-)
(|HomogeneousAggregate&| . HOAGG-)
(|HyperbolicFunctionCategory&| . HYPCAT-)
(|IndexedAggregate&| . IXAGG-)
(|InnerEvalable&| . IEVALAB-)
(|IntegerNumberSystem&| . INS-)
(|IntegralDomain&| . INTDOM-)
(|KeyedDictionary&| . KDAGG-)
(|LazyStreamAggregate&| . LZSTAGG-)
(|LeftAlgebra&| . LALG-)

```

```

(|LieAlgebra&| . LIECAT-)
(|LinearAggregate&| . LNAGG-)
(|ListAggregate&| . LSAGG-)
(|Logic&| . LOGIC-)
(|LinearOrdinaryDifferentialOperatorCategory&| . LODOCAT-)
(|MatrixCategory&| . MATCAT-)
(|Module&| . MODULE-)
(|Monad&| . MONAD-)
(|MonadWithUnit&| . MONADWU-)
(|Monoid&| . MONOID-)
(|MonogenicAlgebra&| . MONOGEN-)
(|NonAssociativeAlgebra&| . NAALG-)
(|NonAssociativeRing&| . NASRING-)
(|NonAssociativeRng&| . NARNG-)
(|OctonionCategory&| . OC-)
(|OneDimensionalArrayAggregate&| . A1AGG-)
(|OrderedRing&| . ORDRING-)
(|OrderedSet&| . ORDSET-)
(|PartialDifferentialRing&| . PDRING-)
(|PolynomialCategory&| . POLYCAT-)
(|PolynomialFactorizationExplicit&| . PFECAT-)
(|PolynomialSetCategory&| . PSETCAT-)
(|PowerSeriesCategory&| . PSCAT-)
(|QuaternionCategory&| . QUATCAT-)
(|QuotientFieldCategory&| . QFCAT-)
(|RadicalCategory&| . RADCAT-)
(|RealClosedField&| . RCFIELD-)
(|RealNumberSystem&| . RNS-)
(|RealRootCharacterizationCategory&| . RRCC-)
(|RectangularMatrixCategory&| . RMATCAT-)
(|RecursiveAggregate&| . RCAGG-)
(|RecursivePolynomialCategory&| . RPOLCAT-)
(|RegularTriangularSetCategory&| . RSETCAT-)
(|RetractableTo&| . RETRACT-)
(|Ring&| . RING-)
(|SemiGroup&| . SGROUP-)
(|SetAggregate&| . SETAGG-)
(|SetCategory&| . SETCAT-)
(|SquareMatrixCategory&| . SMATCAT-)
(|StreamAggregate&| . STAGG-)
(|StringAggregate&| . SRAGG-)
(|TableAggregate&| . TBAGG-)
(|TranscendentalFunctionCategory&| . TRANFUN-)
(|TriangularSetCategory&| . TSETCAT-)
(|TrigonometricFunctionCategory&| . TRIGCAT-)
(|TwoDimensionalArrayCategory&| . ARR2CAT-)
(|UnaryRecursiveAggregate&| . URAGG-)
(|UniqueFactorizationDomain&| . UFD-)
(|UnivariateLaurentSeriesConstructorCategory&| . ULSCCAT-)
(|UnivariatePolynomialCategory&| . UPOLYC-)

```

```
(|UnivariatePowerSeriesCategory&| . UPSCAT-)
(|UnivariatePuisseuxSeriesConstructorCategory&| . UPXSCCA-)
(|UnivariateSkewPolynomialCategory&| . OREPCAT-)
(|UnivariateTaylorSeriesCategory&| . UTSCAT-)
(|VectorCategory&| . VECTCAT-)
(|VectorSpace&| . VSPACE-)))))
```

44.18.2 defvar \$localExposureDataDefault

— initvars —

```
(defvar |$localExposureDataDefault|
  (vector
    ;;These groups will be exposed
    (list '|basic| '|categories| '|naglink| '|anna|)
    ;;These constructors will be explicitly exposed
    (list )
    ;;These constructors will be explicitly hidden
    (list )))
```

44.18.3 defvar \$localExposureData

— initvars —

```
(defvar |$localExposureData| (copy-seq |$localExposureDataDefault|))
```

44.19 Functions

44.19.1 The top level set expose command handler

```
[displayExposedGroups p678]
[sayMSG p333]
[displayExposedConstructors p678]
[displayHiddenConstructors p679]
```

```
[sayKeyedMsg p331]
[namestring p996]
[pathname p998]
[pairp p??]
[qcar p??]
[qcdr p??]
[selectOptionLC p459]
[setExposeAdd p671]
[setExposeDrop p675]
[setExpose p670]
```

— **defun setExpose** —

```
(defun |setExpose| (arg)
  "The top level set expose command handler"
  (let (fnargs fn)
    (cond
      ((eq arg '|%initialize%|))
      ((eq arg '|%display%|) "...")
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|displayExposedGroups|)
       (|sayMSG| " ")
       (|displayExposedConstructors|)
       (|sayMSG| " ")
       (|displayHiddenConstructors|)
       (|sayMSG| " ")))
    ((and (pairp arg)
          (progn (setq fn (qcar arg)) (setq fnargs (qcdr arg)) t)
          (setq fn (|selectOptionLC| fn '(|add| |drop|) nil))))
    (cond
      ((eq fn '|add|) (|setExposeAdd| fnargs))
      ((eq fn '|drop|) (|setExposeDrop| fnargs))
      (t nil)))
    (t (|setExpose| nil))))))
```

—————

44.19.2 The top level set expose add command handler

```
[centerAndHighlight p??]
[specialChar p936]
[displayExposedGroups p678]
[sayMSG p333]
[displayExposedConstructors p678]
[sayKeyedMsg p331]
[pairp p??]
```



```
[qcar p??]
[qcdr p??]
[selectOptionLC p459]
[setExposeAddGroup p672]
[setExposeAddConstr p674]
[setExposeAdd p671]
[$linelength p751]
```

— defun setExposeAdd —

```
(defun |setExposeAdd| (arg)
  "The top level set expose add command handler"
  (declare (special $linelength))
  (let (fnargs fn)
    (cond
      ((null arg)
        (|centerAndHighlight|
          '|The add Option| $linelength (|specialChar| '|hbar|))
        (|displayExposedGroups|)
        (|sayMSG| " ")
        (|displayExposedConstructors|)
        (|sayMSG| " ")
        (|sayKeyedMsg| 's2iz0049e nil))
      ((and (pairp arg)
        (progn (setq fn (qcar arg)) (setq fnargs (qcdr arg)) t)
        (setq fn (|selectOptionLC| fn '(|group| |constructor|) nil))))
        (cond
          ((eq fn '|group|) (|setExposeAddGroup| fnargs))
          ((eq fn '|constructor|) (|setExposeAddConstr| fnargs))
          (t nil)))
      (t (|setExposeAdd| nil))))))
```

—

44.19.3 Expose a group

Note that \$localExposureData is a vector of lists. It consists of [exposed groups,exposed constructors,hidden constructors] [object2String p??]

```
[pairp p??]
[qcar p??]
[setelt p??]
[displayExposedGroups p678]
[sayMSG p333]
[displayExposedConstructors p678]
[displayHiddenConstructors p679]
[clearClams p??]
```

```
[getalist p??]
[sayKeyedMsg p331]
[member p1004]
[msort p??]
[centerAndHighlight p??]
[specialChar p936]
[namestring p996]
[pathname p998]
[sayAsManyPerLineAsPossible p??]
[$globalExposureGroupAlist p644]
[$localExposureData p670]
[$interpreterFrameName p??]
[$linelength p751]
```

— defun setExposeAddGroup —

```
(defun |setExposeAddGroup| (arg)
  "Expose a group"
  (declare (special |$globalExposureGroupAlist| |$localExposureData|
                  |$interpreterFrameName| $linelength))
  (if (null arg)
      (progn
        (|centerAndHighlight|
         '|The group Option| $linelength (|specialChar| '|hbar|))
        (|displayExposedGroups|)
        (|sayMSG| " ")
        (|sayAsManyPerLineAsPossible|
         (mapcar #'(lambda (x) (|object2String| (first x)))
                  |$globalExposureGroupAlist|)))
      (dolist (x arg)
        (when (pairp x) (setq x (qcar x)))
        (cond
         ((eq x '|all|)
          (setelt |$localExposureData| 0
                  (mapcar #'first |$globalExposureGroupAlist|))
          (setelt |$localExposureData| 1 nil)
          (setelt |$localExposureData| 2 nil)
          (|displayExposedGroups|)
          (|sayMSG| " ")
          (|displayExposedConstructors|)
          (|sayMSG| " ")
          (|displayHiddenConstructors|)
          (|clearClams|))
         ((null (getalist |$globalExposureGroupAlist| x))
          (|sayKeyedMsg| 's2iz0049h (cons x nil)))
         ((|member| x (elt |$localExposureData| 0))
          (|sayKeyedMsg| 's2iz0049i (list x |$interpreterFrameName|)))
         (t
          (setelt |$localExposureData| 0
```

```
(msort (cons x (elt |$localExposureData| 0)))
(|sayKeyedMsg| 's2iz0049r (list x |$interpreterFrameName|))
(|clearClams|))))))
```

44.19.4 The top level set expose add constructor handler

```
[unabbrev p??]
[paip p??]
[qcar p??]
[getdatabase p967]
[sayKeyedMsg p331]
[member p1004]
[setelt p??]
[delete p??]
[msort p??]
[clearClams p??]
[centerAndHighlight p??]
[specialChar p936]
[displayExposedConstructors p678]
[$linelength p751]
[$localExposureData p670]
[$interpreterFrameName p??]
```

— defun setExposeAddConstr —

```
(defun |setExposeAddConstr| (arg)
  "The top level set expose add constructor handler"
  (declare (special $linelength |$localExposureData| |$interpreterFrameName|))
  (if (null arg)
    (progn
      (|centerAndHighlight|
        '|The constructor Option| $linelength (|specialChar| '|hbar|))
      (|displayExposedConstructors|))
    (dolist (x arg)
      (setq x (|unabbrev| x))
      (when (paip x) (setq x (qcar x)))
      (cond
        ((null (getdatabase x 'constructorkind))
          (|sayKeyedMsg| 's2iz0049j (list x)))
        ((|member| x (elt |$localExposureData| 1))
          (|sayKeyedMsg| 's2iz0049k (list x |$interpreterFrameName| )))
        (t
          (when (|member| x (elt |$localExposureData| 2))
            (setelt |$localExposureData| 2
```

```
(|delete| x (elt |$localExposureData| 2)))
(setelt |$localExposureData| 1
 (msort (cons x (elt |$localExposureData| 1))))
(|clearClams|)
(|sayKeyedMsg| 's2iz0049p (list x |$interpreterFrameName| )))))))
```

44.19.5 The top level set expose drop handler

```
[centerAndHighlight p??]
[specialChar p936]
[displayHiddenConstructors p679]
[sayMSG p333]
[sayKeyedMsg p331]
[pairp p??]
[qcar p??]
[qcdr p??]
[selectOptionLC p459]
[setExposeDropGroup p676]
[setExposeDropConstr p677]
[setExposeDrop p675]
[$linelength p751]
```

— defun setExposeDrop —

```
(defun |setExposeDrop| (arg)
  "The top level set expose drop handler"
  (declare (special $linelength))
  (let (fnargs fn)
    (cond
      ((null arg)
        (|centerAndHighlight|
          '|The drop Option| $linelength (|specialChar| '|hbar|))
        (|displayHiddenConstructors|)
        (|sayMSG| " ")
        (|sayKeyedMsg| 's2iz0049f nil))
      ((and (pairp arg)
        (progn (setq fn (qcar arg)) (setq fnargs (qcdr arg)) t)
        (setq fn (|selectOptionLC| fn '|group| |constructor|) nil)))
        (cond
          ((eq fn '|group|) (|setExposeDropGroup| fnargs))
          ((eq fn '|constructor|) (|setExposeDropConstr| fnargs))
          (t nil)))
      (t (|setExposeDrop| nil))))))
```

44.19.6 The top level set expose drop group handler

```
[pairp p??]
[qcar p??]
[setelt p??]
[displayExposedGroups p678]
[sayMSG p333]
[displayExposedConstructors p678]
[displayHiddenConstructors p679]
[clearClams p??]
[member p1004]
[delete p??]
[sayKeyedMsg p331]
[getalist p??]
[centerAndHighlight p??]
[specialChar p936]
[$linelength p751]
[$localExposureData p670]
[$interpreterFrameName p??]
[$globalExposureGroupAlist p644]
```

— defun setExposeDropGroup —

```
(defun |setExposeDropGroup| (arg)
  "The top level set expose drop group handler"
  (declare (special $linelength |$localExposureData| |$interpreterFrameName|
                    |$globalExposureGroupAlist|))
  (if (null arg)
      (progn
        (|centerAndHighlight|
         '|The group Option| $linelength (|specialChar| 'hbar|))
        (|sayKeyedMsg| 's2iz00491 nil)
        (|sayMSG| " ")
        (|displayExposedGroups|))
      (dolist (x arg)
        (when (pairp x) (setq x (qcar x)))
        (cond
         ((eq x '|all|)
          (setelt |$localExposureData| 0 nil)
          (setelt |$localExposureData| 1 nil)
          (setelt |$localExposureData| 2 nil)
          (|displayExposedGroups|)
          (|sayMSG| " ")
          (|displayExposedConstructors|)
          (|sayMSG| " ")))
```

```

(|displayHiddenConstructors|)
(|clearClams|))
((|member| x (elt |$localExposureData| 0))
 (setelt |$localExposureData| 0
  (|delete| x (elt |$localExposureData| 0)))
(|clearClams|)
(|sayKeyedMsg| 's2iz0049s (list x |$interpreterFrameName| )))
((|getalist |$globalExposureGroupAlist| x)
 (|sayKeyedMsg| 's2iz0049i (list x |$interpreterFrameName| )))
(t (|sayKeyedMsg| 's2iz0049h (list x ))))))))

```

44.19.7 The top level set expose drop constructor handler

```

[unabbrev p??]
[pairp p??]
[qcar p??]
[getdatabase p967]
[sayKeyedMsg p331]
[member p1004]
[setelt p??]
[delete p??]
[msort p??]
[clearClams p??]
[centerAndHighlight p??]
[specialChar p936]
[sayMSG p333]
[displayExposedConstructors p678]
[displayHiddenConstructors p679]
[$linelength p751]
[$localExposureData p670]
[$interpreterFrameName p??]

```

— defun setExposeDropConstr —

```

(defun |setExposeDropConstr| (arg)
  "The top level set expose drop constructor handler"
  (declare (special $linelength |$localExposureData| |$interpreterFrameName|))
  (if (null arg)
    (progn
      (|centerAndHighlight|
        '|The constructor Option| $linelength (|specialChar| '|hbar|))
      (|sayKeyedMsg| 's2iz0049n nil)
      (|sayMSG| " ")
      (|displayExposedConstructors|)
    )
  )

```

```

(|sayMSG| " ")
(|displayHiddenConstructors|)
(dolist (x arg)
  (setq x (|unabbrev| x))
  (when (pairp x) (setq x (qcar x)))
  (cond
   ((null (getdatabase x 'constructorkind))
    (|sayKeyedMsg| 's2iz0049j (list x)))
   ((|member| x (elt |$localExposureData| 2))
    (|sayKeyedMsg| 's2iz0049o (list x |$interpreterFrameName|)))
   (t
    (when (|member| x (elt |$localExposureData| 1))
      (setelt |$localExposureData| 1
        (|delete| x (elt |$localExposureData| 1))))
      (setelt |$localExposureData| 2
        (msort (cons x (elt |$localExposureData| 2))))
      (|clearClams|)
      (|sayKeyedMsg| 's2iz0049q (list x |$interpreterFrameName|))))))

```

44.19.8 Display exposed groups

```

[sayKeyedMsg p331]
[centerAndHighlight p??]
[$interpreterFrameName p??]
[$localExposureData p670]

```

— defun displayExposedGroups —

```

(defun |displayExposedGroups| ()
  "Display exposed groups"
  (declare (special |$interpreterFrameName| |$localExposureData|))
  (|sayKeyedMsg| 's2iz0049a (list |$interpreterFrameName|))
  (if (null (elt |$localExposureData| 0))
      (|centerAndHighlight| "there are no exposed groups")
      (dolist (c (elt |$localExposureData| 0))
        (|centerAndHighlight| c))))

```

44.19.9 Display exposed constructors

```

[sayKeyedMsg p331]
[centerAndHighlight p??]

```

[`$localExposureData` p670]

— `defun displayExposedConstructors` —

```
(defun |displayExposedConstructors| ()
  "Display exposed constructors"
  (declare (special |$localExposureData|))
  (|sayKeyedMsg| 's2iz0049b nil)
  (if (null (elt |$localExposureData| 1))
      (|centerAndHighlight| "there are no explicitly exposed constructors")
      (dolist (c (elt |$localExposureData| 1))
        (|centerAndHighlight| c))))
```

—————

44.19.10 Display hidden constructors

[`sayKeyedMsg` p331]
 [`centerAndHighlight` p??]
 [`$localExposureData` p670]

— `defun displayHiddenConstructors` —

```
(defun |displayHiddenConstructors| ()
  "Display hidden constructors"
  (declare (special |$localExposureData|))
  (|sayKeyedMsg| 's2iz0049c nil)
  (if (null (elt |$localExposureData| 2))
      (|centerAndHighlight| "there are no explicitly hidden constructors")
      (dolist (c (elt |$localExposureData| 2))
        (|centerAndHighlight| c))))
```

—————

44.20 functions

Current Values of functions Variables

Variable	Description	Current Value
cache	number of function results to cache	0
compile	compile, don't just define function bodies	off
recurrence	specially compile recurrence relations	on

— functions —

```
(|functions|
  "some interpreter function options"
  |interpreter|
  TREE
  |novar|
  (
    \getchunk{functionscache}
    \getchunk{functionscompile}
    \getchunk{functionsrecurrence}
  ))
```

44.21 functions cache

----- The cache Option -----

Description: number of function results to cache

)set functions cache is used to tell AXIOM how many values computed by interpreter functions should be saved. This can save quite a bit of time in recursive functions, though one must consider that the cached values will take up (perhaps valuable) room in the workspace.

The value given after cache must either be the word all or a positive integer. This may be followed by any number of function names whose cache sizes you wish to so set. If no functions are given, the default cache size is set.

Examples:)set fun cache all
)set fun cache 10 f g Legendre

In general, functions will cache no returned values.

— functionscache —

```
(|cache|
  "number of function results to cache"
  |interpreter|
  FUNCTION
  |setFunctionsCache|
```

```
NIL
|htSetCache|)
```

44.22 Variables Used

44.22.1 defvar \$cacheAlist

— initvars —

```
(defvar |$cacheAlist| nil)
```

44.23 Functions

44.23.1 The top level set functions cache handler

```
\calls{setFunctionsCache}{object2String}
\calls{setFunctionsCache}{describeSetFunctionsCache}
\calls{setFunctionsCache}{sayAllCacheCounts}
\calls{setFunctionsCache}{nequal}
\calls{setFunctionsCache}{sayMessage}
\calls{setFunctionsCache}{bright}
\calls{setFunctionsCache}{terminateSystemCommand}
\calls{setFunctionsCache}{countCache}
\usesdollar{setFunctionsCache}{options}
\usesdollar{setFunctionsCache}{cacheCount}
\usesdollar{setFunctionsCache}{cacheAlist}
\begin{chunk}{defun setFunctionsCache}
(defun |setFunctionsCache| (arg)
  "The top level set functions cache handler"
  (let (|$options| n)
    (declare (special |$options| |$cacheCount| |$cacheAlist|))
    (cond
      ((eq arg '|%initialize%|)
       (setq |$cacheCount| 0)
       (setq |$cacheAlist| nil))
      ((eq arg '|%display%|)
       (if (null |$cacheAlist|)
           (|object2String| |$cacheCount|)
           "..."))
```

```

((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
  (|describeSetFunctionsCache|)
  (terpri)
  (|sayAllCacheCounts|))
(t
  (setq n (car arg))
  (cond
    ((and (nequal n '|all|) (or (null (integerp n)) (minusp n)))
      (|sayMessage|
        '("Your value of" ,@(|bright| n) "is invalid because ..."))
      (|describeSetFunctionsCache|)
      (|terminateSystemCommand|))
    (t
      (when (cdr arg) (list (cons '|vars| (cdr arg))))
      (|countCache| n))))))

\end{chunk}

\defunsec{countCache}{Display a particular cache count}
\calls{countCache}{pairp}
\calls{countCache}{qcdr}
\calls{countCache}{qcar}
\calls{countCache}{identp}
\calls{countCache}{sayKeyedMsg}
\calls{countCache}{insertAlist}
\calls{countCache}{internl}
\calls{countCache}{sayCacheCount}
\calls{countCache}{optionError}
\usesdollar{countCache}{options}
\usesdollar{countCache}{cacheAlist}
\usesdollar{countCache}{cacheCount}
\begin{chunk}{defun countCache}
(defun |countCache| (n)
  "Display a particular cache count"
  (let (tmp1 l cachecountname)
    (declare (special |$options| |$cacheAlist| |$cacheCount|))
    (cond
      (|$options|
        (cond
          ((and (pairp |$options|)
            (eq (qcdr |$options|) nil)
            (progn
              (setq tmp1 (qcar |$options|))
              (and (pairp tmp1)
                (eq (qcar tmp1) '|vars|)
                (progn (setq l (qcdr tmp1)) t))))
          (dolist (x l)
            (if (null (identp x))
              (|sayKeyedMsg| 's2if0007 (list x))
              (progn

```

```

        (setq |$cacheAlist| (|insertAlist| x n |$cacheAlist|))
        (setq cachecountname (internl x ";COUNT"))
        (set cachecountname n)
        (|sayCacheCount| x n))))
    (t (|optionError| (caar |$options|) nil))))
(t
  (|sayCacheCount| nil (setq |$cacheCount| n))))))

\end{chunk}

\defunsec{describeSetFunctionsCache}{Describe the set functions cache}
\calls{describeSetFunctionsCache}{sayBrightly}
\begin{chunk}{defun describeSetFunctionsCache}
(defun |describeSetFunctionsCache| ()
  "Describe the set functions cache"
  (|sayBrightly| (list
    '|%b| "set functions cache"
    '|%d| "is used to tell AXIOM how many"
    '|%l| " values computed by interpreter functions should be saved. This"
    '|%l| " can save quite a bit of time in recursive functions, though one"
    '|%l| " must consider that the cached values will take up (perhaps"
    '|%l| " valuable) room in the workspace."
    '|%l|
    '|%l| " The value given after"
    '|%b| "cache"
    '|%d| "must either be the word"
    '|%b| "all"
    '|%d| "or a positive integer."
    '|%l| " This may be followed by any number of function names whose cache"
    '|%l| " sizes you wish to so set. If no functions are given, the default"
    '|%l| " cache size is set."
    '|%l|
    '|%l| " Examples:"
    '|%l| "   )set fun cache all           )set fun cache 10 f g Legendre"))))

\end{chunk}

\defunsec{sayAllCacheCounts}{Display all cache counts}
\calls{sayAllCacheCounts}{sayCacheCount}
\calls{sayAllCacheCounts}{nequal}
\usesdollar{sayAllCacheCounts}{cacheCount}
\usesdollar{sayAllCacheCounts}{cacheAlist}
\begin{chunk}{defun sayAllCacheCounts}
(defun |sayAllCacheCounts| ()
  "Display all cache counts"
  (let (x n)
    (declare (special |$cacheCount| |$cacheAlist|))
    (|sayCacheCount| nil |$cacheCount|)
    (when |$cacheAlist|
      (do ((t0 |$cacheAlist| (cdr t0)) (t1 nil))

```

```

      ((or (atom t0)
            (progn (setq t1 (car t0)) nil)
            (progn
              (progn (setq x (car t1)) (setq n (cdr t1)) t1)
              nil))
            nil)
      (when (nequal n |$cacheCount|) (|sayCacheCount| x n))))))

\end{chunk}

\defunsec{sayCacheCount}{Describe the cache counts}
\calls{sayCacheCount}{bright}
\calls{sayCacheCount}{linearFormatName}
\calls{sayCacheCount}{sayBrightly}
\begin{chunk}{defun sayCacheCount}
(defun |sayCacheCount| (fn n)
  "Describe the cache counts"
  (let (prefix phrase)
    (setq prefix
      (cond
        (fn (cons '|function| (|bright| (|linearFormatName| fn))))
        ((eq1 n 0) (list '|interpreter functions|))
        (t (list '|In general, interpreter functions|))))
      (cond
        ((eq1 n 0)
          (cond
            (fn
              (|sayBrightly|
                '("   Caching for " ,prefix "is turned off")))
            (t
              (|sayBrightly| " In general, functions will cache no returned values."
                ))))
          (t
            (setq phrase
              (cond
                ((eq n '|all|) '(',@(|bright| '|all|) |values.|))
                ((eq1 n 1) (list '| only the last value.|))
                (t '(| the last| ,@(|bright| n) |values.|))))
              (|sayBrightly|
                '("   " ,@prefix "will cache" ,@phrase))))))

\end{chunk}

\section{functions compile}
\begin{verbatim}
----- The compile Option -----

Description: compile, don't just define function bodies

The compile option may be followed by any one of the following:

```

```
-> on
    off
```

The current setting is indicated.

44.23.2 defvar \$compileDontDefineFunctions

— initvars —

```
(defvar |$compileDontDefineFunctions| t
  "compile, don't just define function bodies")
```

— functionscompile —

```
(|compile|
  "compile, don't just define function bodies"
 |interpreter|
 LITERALS
 |$compileDontDefineFunctions|
 (|on| |off|)
 |on|)
```

44.24 functions recurrence

----- The recurrence Option -----

Description: specially compile recurrence relations

The recurrence option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.24.1 defvar \$compileRecurrence

— initvars —

```
(defvar |$compileRecurrence| t "specially compile recurrence relations")
```

— functionsrecurrence —

```
(|recurrence|
 "specially compile recurrence relations"
 |interpreter|
 LITERALS
 |$compileRecurrence|
 (|on| |off|)
 |on|)
```

44.25 fortran

Current Values of fortran Variables

Variable	Description	Current Value
ints2floats	where sensible, coerce integers to reals	on
fortindent	the number of characters indented	6
fortlength	the number of characters on a line	72
typedecs	print type and dimension lines	on
defaulttype	default generic type for FORTRAN object	REAL
precision	precision of generated FORTRAN objects	double
intrinsic	whether to use INTRINSIC FORTRAN functions	off
explength	character limit for FORTRAN expressions	1320
segment	split long FORTRAN expressions	on
optlevel	FORTRAN optimisation level	0
startindex	starting index for FORTRAN arrays	1
calling	options for external FORTRAN calls	...

Variables with current values of ... have further sub-options.
For example, issue)set calling to see what the options are for
calling.

For more information, issue)help set .

— fortran —

```
(|fortran|
  "view and set options for FORTRAN output"
  |interpreter|
  TREE
  |novar|
  (
\getchunk{fortranints2floats}
\getchunk{fortranfortindent}
\getchunk{fortranfortlength}
\getchunk{fortrantypedecs}
\getchunk{fortrandefaultttype}
\getchunk{fortranprecision}
\getchunk{fortranintrinsic}
\getchunk{fortranexplength}
\getchunk{fortransegment}
\getchunk{fortranoptlevel}
\getchunk{fortranstartindex}
\getchunk{fortrancalling}
  ))
```

44.25.1 ints2floats

----- The ints2floats Option -----

Description: where sensible, coerce integers to reals

The ints2floats option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.25.2 defvar \$fortInts2Floats

— initvars —

```
(defvar |$fortInts2Floats| t "where sensible, coerce integers to reals")
```

— **fortranints2floats** —

```
(|ints2floats|
  "where sensible, coerce integers to reals"
  |interpreter|
  LITERALS
  |$fortInts2Floats|
  (|on| |off|)
  |on|)
```

44.25.3 fortindent

----- The fortindent Option -----

Description: the number of characters indented

The fortindent option may be followed by an integer in the range 0 to inclusive. The current setting is 6

44.25.4 defvar \$fortIndent— **initvars** —

```
(defvar |$fortIndent| 6 "the number of characters indented")
```

— **fortranfortindent** —

```
(|fortindent|
  "the number of characters indented"
  |interpreter|
  INTEGER
  |$fortIndent|
  (0 NIL)
  6)
```

44.25.5 forlength

----- The forlength Option -----

Description: the number of characters on a line

The forlength option may be followed by an integer in the range 1 to inclusive. The current setting is 72

44.25.6 defvar \$fortLength

— initvars —

```
(defvar |$fortLength| 72 "the number of characters on a line")
```

— fortranfortlength —

```
(|fortlength|
  "the number of characters on a line"
  |interpreter|
  INTEGER
  |$fortLength|
  (1 NIL)
  72)
```

44.25.7 typedecs

----- The typedecs Option -----

Description: print type and dimension lines

The typedecs option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.25.8 defvar \$printFortranDecs

— initvars —

```
(defvar |$printFortranDecs| t "print type and dimension lines")
```

—————

— fortrantypedecs —

```
(|typedecs|
 "print type and dimension lines"
 |interpreter|
 LITERALS
 |$printFortranDecs|
 (|on| |off|)
 |on|)
```

—————

44.25.9 defaulttype

----- The defaulttype Option -----

Description: default generic type for FORTRAN object

The defaulttype option may be followed by any one of the following:

```
-> REAL
    INTEGER
    COMPLEX
    LOGICAL
    CHARACTER
```

The current setting is indicated.

44.25.10 defvar \$defaultFortranType

— initvars —

```
(defvar |$defaultFortranType| 'real "default generic type for FORTRAN object")
```

— **fortrandefaulttype** —

```
(|defaulttype|
 "default generic type for FORTRAN object"
 |interpreter|
 LITERALS
 |$defaultFortranType|
 (REAL INTEGER COMPLEX LOGICAL CHARACTER)
 REAL)
```

44.25.11 precision

----- The precision Option -----

Description: precision of generated FORTRAN objects

The precision option may be followed by any one of the following:

```
single
-> double
```

The current setting is indicated.

44.25.12 defvar \$fortranPrecision

— **initvars** —

```
(defvar |$fortranPrecision| 'double| "precision of generated FORTRAN objects")
```

— **fortranprecision** —

```
(|precision|
 "precision of generated FORTRAN objects"
 |interpreter|
 LITERALS
 |$fortranPrecision|
```

```
(|single| |double|)
|double|)
```

44.25.13 intrinsic

----- The intrinsic Option -----

Description: whether to use INTRINSIC FORTRAN functions

The intrinsic option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.25.14 defvar \$useIntrinsicFunctions

— initvars —

```
(defvar |$useIntrinsicFunctions| nil
  "whether to use INTRINSIC FORTRAN functions")
```

— fortranintrinsic —

```
(|intrinsic|
  "whether to use INTRINSIC FORTRAN functions"
|interpreter|
LITERALS
|$useIntrinsicFunctions|
(|on| |off|)
|off|)
```

44.25.15 explength

----- The explength Option -----

Description: character limit for FORTRAN expressions

The `explength` option may be followed by an integer in the range 0 to inclusive. The current setting is 1320

44.25.16 `defvar $maximumFortranExpressionLength`

— `initvars` —

```
(defvar |$maximumFortranExpressionLength| 1320
  "character limit for FORTRAN expressions")
```

— `fortranexplength` —

```
(|explength|
  "character limit for FORTRAN expressions"
  |interpreter|
  INTEGER
  |$maximumFortranExpressionLength|
  (0 NIL)
  1320)
```

44.25.17 `segment`

----- The segment Option -----

Description: split long FORTRAN expressions

The `segment` option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.25.18 defvar \$fortranSegment

— initvars —

```
(defvar |$fortranSegment| t "split long FORTRAN expressions")
```

—————

— fortransegment —

```
(|segment|
 "split long FORTRAN expressions"
 |interpreter|
 LITERALS
 |$fortranSegment|
 (|on| |off|)
 |on|)
```

—————

44.25.19 optlevel

----- The optlevel Option -----

Description: FORTRAN optimisation level

The optlevel option may be followed by an integer in the range 0 to 2 inclusive. The current setting is 0

44.25.20 defvar \$fortranOptimizationLevel

— initvars —

```
(defvar |$fortranOptimizationLevel| 0 "FORTRAN optimisation level")
```

—————

— fortranoptlevel —

```
(|optlevel|
 "FORTRAN optimisation level")
```

```
|interpreter|
INTEGER
|$fortranOptimizationLevel|
(0 2)
0)
```

44.25.21 startindex

----- The startindex Option -----

Description: starting index for FORTRAN arrays

The startindex option may be followed by an integer in the range 0 to 1 inclusive. The current setting is 1

44.25.22 defvar \$fortranArrayStartingIndex

— initvars —

```
(defvar |$fortranArrayStartingIndex| 1 "starting index for FORTRAN arrays")
```

— fortranstartindex —

```
(|startindex|
"starting index for FORTRAN arrays"
|interpreter|
INTEGER
|$fortranArrayStartingIndex|
(0 1)
1)
```

44.25.23 calling

Current Values of calling Variables

Variable	Description	Current Value

```

tempfile      set location of temporary data files      /tmp/
directory     set location of generated FORTRAN files    ./
linker        linker arguments (e.g. libraries to search) -lxf

```

— fortrancalling —

```

(|calling|
"options for external FORTRAN calls"
|interpreter|
TREE
|novar|
(
\getchunk{callingtempfile}
\getchunk{callingdirectory}
\getchunk{callinglinker}
)
)

```

tempfile

----- The tempfile Option -----

Description: set location of temporary data files

)set fortran calling tempfile is used to tell AXIOM where to place intermediate FORTRAN data files . This must be the name of a valid existing directory to which you have permission to write (including the final slash).

Syntax:

```
)set fortran calling tempfile DIRECTORYNAME
```

The current setting is /tmp/

44.25.24 defvar \$fortranTmpDir

— initvars —

```
(defvar |$fortranTmpDir| "/tmp/" "set location of temporary data files")
```

— callingtempfile —

```
(|tempfile|
  "set location of temporary data files"
  |interpreter|
  FUNCTION
  |setFortTmpDir|
  (("enter directory name for which you have write-permission"
    DIRECTORY
    |$fortranTmpDir|
    |chkDirectory|
    "/tmp/"))
  NIL)
```

44.25.25 The top level set fortran calling tempfile handler

```
[pname p1001]
[describeSetFortTmpDir p698]
[validateOutputDirectory p698]
[sayBrightly p??]
[bright p??]
[$fortranTmpDir p696]
```

— defun setFortTmpDir —

```
(defun |setFortTmpDir| (arg)
  "The top level set fortran calling tempfile handler"
  (let (mode)
    (declare (special |$fortranTmpDir|))
    (cond
      ((eq arg '|%initialize%|) (setq |$fortranTmpDir| "/tmp/"))
      ((eq arg '|%display%|)
        (if (stringp |$fortranTmpDir|)
          |$fortranTmpDir|
          (pname |$fortranTmpDir|)))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '??))
        (|describeSetFortTmpDir|))
      ((null (setq mode (|validateOutputDirectory| arg)))
        (|sayBrightly|
          '(" Sorry, but your argument(s)" ,@(|bright| arg)
            "is(are) not valid." |%l|))
        (|describeSetFortTmpDir|))
      (t (setq |$fortranTmpDir| mode))))))
```

44.25.26 Validate the output directory

— defun validateOutputDirectory —

```
(defun |validateOutputDirectory| (x)
  "Validate the output directory"
  (let ((dirname (car x)))
    (when (and (pathname-directory dirname) (null (probe-file dirname)))
      dirname)))
```

44.25.27 Describe the set fortran calling tempfile

```
[sayBrightly p??]
[$fortranTmpDir p696]
```

— defun describeSetFortTmpDir —

```
(defun |describeSetFortTmpDir| ()
  "Describe the set fortran calling tempfile"
  (declare (special |$fortranTmpDir|))
  (|sayBrightly| (list
    '|%b| ")set fortran calling tempfile"
    '|%d| " is used to tell AXIOM where"
    '|%l| " to place intermediate FORTRAN data files . This must be the "
    '|%l| " name of a valid existing directory to which you have permission "
    '|%l| " to write (including the final slash)."'
    '|%l|
    '|%l| " Syntax:"
    '|%l| " )set fortran calling tempfile DIRECTORYNAME"
    '|%l|
    '|%l| " The current setting is"
    '|%b| |$fortranTmpDir|
    '|%d|)))
```

directory

----- The directory Option -----

Description: set location of generated FORTRAN files

)set fortran calling directory is used to tell AXIOM where to place generated FORTRAN files. This must be the name of a valid existing directory to which you have permission to write (including the final slash).

Syntax:

```
)set fortran calling directory DIRECTORYNAME
```

The current setting is ./

44.25.28 defvar \$fortranDirectory

— initvars —

```
(defvar |$fortranDirectory| "./" "set location of generated FORTRAN files")
```

—————

— callingdirectory —

```
(|directory|
 "set location of generated FORTRAN files"
 |interpreter|
 FUNCTION
 |setFortDir|
 (("enter directory name for which you have write-permission"
  DIRECTORY
  |$fortranDirectory|
  |chkDirectory|
  "./")
  NIL)
```

—————

44.25.29 defun setFortDir

```
[pname p1001]
[describeSetFortDir p700]
[validateOutputDirectory p698]
[sayBrightly p??]
[bright p??]
```

[`$fortranDirectory` p699]

— **defun setFortDir** —

```
(defun |setFortDir| (arg)
  (declare (special |$fortranDirectory|))
  (let (mode)
    (COND
      ((eq arg '|%initialize%|) (setq |$fortranDirectory| ". /"))
      ((eq arg '|%display%|)
       (if (stringp |$fortranDirectory|)
           |$fortranDirectory|
           (pname |$fortranDirectory|)))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeSetFortDir|))
      ((null (setq mode (|validateOutputDirectory| arg)))
       (|sayBrightly|
        (" Sorry, but your argument(s)" ,@(|bright| arg)
         "is(are) not valid." |%l|))
       (|describeSetFortDir|))
      (t (setq |$fortranDirectory| mode))))))
```

—————

44.25.30 defun describeSetFortDir

[`sayBrightly` p??]

[`$fortranDirectory` p699]

— **defun describeSetFortDir** —

```
(defun |describeSetFortDir| ()
  (declare (special |$fortranDirectory|))
  (|sayBrightly| (list
    '|%b| ")set fortran calling directory"
    '|%d| " is used to tell AXIOM where"
    '|%l| " to place generated FORTRAN files. This must be the name "'
    '|%l| " of a valid existing directory to which you have permission "'
    '|%l| " to write (including the final slash)."'
    '|%l|
    '|%l| " Syntax:"
    '|%l| " )set fortran calling directory DIRECTORYNAME"'
    '|%l|
    '|%l| " The current setting is"
    '|%b| |$fortranDirectory|
    '|%d|)))
```

linker

----- The linker Option -----

Description: linker arguments (e.g. libraries to search)

)set fortran calling linkerargs is used to pass arguments to the linker when using mkFort to create functions which call Fortran code. For example, it might give a list of libraries to be searched, and their locations.

The string is passed verbatim, so must be the correct syntax for the particular linker being used.

Example:)set fortran calling linker "-lxlif"

The current setting is -lxlif

44.25.31 defvar \$fortranLibraries

— initvars —

```
(defvar |$fortranLibraries| "-lxlif"
  "linker arguments (e.g. libraries to search)")
```

— callinglinker —

```
(|linker|
  "linker arguments (e.g. libraries to search)"
  |interpreter|
  FUNCTION
  |setLinkerArgs|
  (("enter linker arguments "
    STRING
    |$fortranLibraries|
    |chkDirectory|
    "-lxlif"))
  NIL
  )
```

44.25.32 defun setLinkerArgs

```
[object2String p??]
[describeSetLinkerArgs p702]
[$fortranLibraries p701]
```

— defun setLinkerArgs —

```
(defun |setLinkerArgs| (arg)
  (declare (special |$fortranLibraries|))
  (cond
    ((eq arg '|%initialize%|) (setq |$fortranLibraries| "-lxlif"))
    ((eq arg '|%display%|) (|object2String| |$fortranLibraries|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetLinkerArgs|))
    ((and (listp arg) (stringp (car arg)))
     (setq |$fortranLibraries| (car arg)))
    (t (|describeSetLinkerArgs|))))
```

44.25.33 defun describeSetLinkerArgs

```
[sayBrightly p??]
[$fortranLibraries p701]
```

— defun describeSetLinkerArgs —

```
(defun |describeSetLinkerArgs| ()
  (declare (special |$fortranLibraries|))
  (|sayBrightly| (list
    '|%b| " )set fortran calling linkerargs"
    '|%d| " is used to pass arguments to the linker"
    '|%l| " when using "
    '|%b| "mkFort"
    '|%d| " to create functions which call Fortran code."
    '|%l| " For example, it might give a list of libraries to be searched,"
    '|%l| " and their locations."
    '|%l| " The string is passed verbatim, so must be the correct syntax for"
    '|%l| " the particular linker being used."
    '|%l|
    '|%l| " Example: )set fortran calling linker \"-lxlif\""'
    '|%l|
    '|%l| " The current setting is"
    '|%b| |$fortranLibraries|
    '|%d|)))
```

44.26 kernel

Current Values of kernel Variables

Variable	Description	Current Value
warn	warn when re-definition is attempted	off
protect	prevent re-definition of kernel functions	off

— kernel —

```
(|kernel|
  "library functions built into the kernel for efficiency"
  |interpreter|
  TREE
  |novar|
  (
    \getchunk{kernelwarn}
    \getchunk{kernelprotect}
  )
)
```

44.26.1 kernelwarn

----- The warn Option -----

Description: warn when re-definition is attempted

Some AXIOM library functions are compiled into the kernel for efficiency reasons. To prevent them being re-defined when loaded from a library they are specially protected. If a user wishes to know when an attempt is made to re-define such a function, he or she should issue the command:

```
)set kernel warn on
```

To restore the default behaviour, he or she should issue the command:

```
)set kernel warn off
```

— kernelwarn —


```
(|warn|
  "warn when re-definition is attempted"
  |interpreter|
  FUNCTION
  |protectedSymbolsWarning|
  NIL
  |htSetKernelWarn|)
```

44.26.2 defun protectedSymbolsWarning

[protected-symbol-warn p??]
 [describeProtectedSymbolsWarning p704]
 [translateYesNo2TrueFalse p632]

— defun protectedSymbolsWarning —

```
(defun |protectedSymbolsWarning| (arg)
  (let (v)
    (cond
      ((eq arg '|%initialize%|) (protected-symbol-warn nil))
      ((eq arg '|%display%|)
       (setq v (protected-symbol-warn t))
       (protected-symbol-warn v)
       (if v "on" "off")))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeProtectedSymbolsWarning|))
      (t (protected-symbol-warn (|translateYesNo2TrueFalse| (car arg)))))))
```

44.26.3 defun describeProtectedSymbolsWarning

[sayBrightly p??]

— defun describeProtectedSymbolsWarning —

```
(defun |describeProtectedSymbolsWarning| ()
  (|sayBrightly| (list
    "Some AXIOM library functions are compiled into the kernel for efficiency"
    '|%l| "reasons. To prevent them being re-defined when loaded from a library"
    '|%l|
    "they are specially protected. If a user wishes to know when an attempt"
    '|%l|
    "is made to re-define such a function, he or she should issue the command:"
```

```
'|%1| "          )set kernel warn on"
'|%1| "To restore the default behaviour, he or she should issue the command:"
'|%1| "          )set kernel warn off")))
```

44.26.4 kernelprotect

----- The protect Option -----

Description: prevent re-definition of kernel functions

Some AXIOM library functions are compiled into the kernel for efficiency reasons. To prevent them being re-defined when loaded from a library they are specially protected. If a user wishes to re-define these functions, he or she should issue the command:

```
)set kernel protect off
```

To restore the default behaviour, he or she should issue the command:

```
)set kernel protect on
```

— kernelprotect —

```
(|protect|
 "prevent re-definition of kernel functions"
 |interpreter|
 FUNCTION
 |protectSymbols|
 NIL
 |htSetKernelProtect|)
```

44.26.5 defun protectSymbols

```
[protect-symbols p??]
[describeProtectSymbols p706]
[translateYesNo2TrueFalse p632]
```

— defun protectSymbols —

```
(defun |protectSymbols| (arg)
  (let (v)
    (cond
      ((eq arg '|%initialize%|) (protect-symbols t))
```

```

(eq arg '|%display%|)
(setq v (protect-symbols t))
(protect-symbols v)
(if v "on" "off"))
((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
(|describeProtectSymbols|))
(t (protect-symbols (|translateYesNo2TrueFalse| (car arg))))))

```

44.26.6 defun describeProtectSymbols

[sayBrightly p??]

— defun describeProtectSymbols —

```

(defun |describeProtectSymbols| ()
  (|sayBrightly| (list
    "Some AXIOM library functions are compiled into the kernel for efficiency"
    '|%l|
    "reasons. To prevent them being re-defined when loaded from a library"
    '|%l| "they are specially protected. If a user wishes to re-define these"
    '|%l| "functions, he or she should issue the command:"
    '|%l| "          )set kernel protect off"
    '|%l|
    "To restore the default behaviour, he or she should issue the command:"
    '|%l| "          )set kernel protect on"))))

```

44.27 hyperdoc

Current Values of hyperdoc Variables

Variable	Description	Current Value
fullscreen	use full screen for this facility	off
mathwidth	screen width for history output	120

— hyperdoc —

```
(|hyperdoc|
```

```

"options in using HyperDoc"
|interpreter|
TREE
|novar|
(
\getchunk{hyperdocfullscreen}
\getchunk{hyperdocmathwidth}
))

```

44.27.1 fullscreen

----- The fullscreen Option -----

Description: use full screen for this facility

The fullscreen option may be followed by any one of the following:

```

on
-> off

```

The current setting is indicated.

44.27.2 defvar \$fullScreenSysVars

— initvars —

```
(defvar |$fullScreenSysVars| nil "use full screen for this facility")
```

— hyperdocfullscreen —

```

(|fullscreen|
"use full screen for this facility"
|interpreter|
LITERALS
|$fullScreenSysVars|
(|on| |off|)
|off|)

```

44.27.3 mathwidth

----- The mathwidth Option -----

Description: screen width for history output

The mathwidth option may be followed by an integer in the range 0 to inclusive. The current setting is 120

44.27.4 defvar \$historyDisplayWidth

— initvars —

```
(defvar |$historyDisplayWidth| 120 "screen width for history output")
```

— hyperdocmathwidth —

```
(|mathwidth|
 "screen width for history output"
 |interpreter|
 INTEGER
 |$historyDisplayWidth|
 (0 NIL)
 120)
```

44.28 help

Current Values of help Variables

Variable	Description	Current Value
fullscreen	use fullscreen facility, if possible	on

— help —

```
(|help|
 "view and set some help options")
```

```
|interpreter|
TREE
|novar|
(
\getchunk{helpfullscreen}
))
```

44.28.1 fullscreen

----- The fullscreen Option -----

Description: use fullscreen facility, if possible

The fullscreen option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.28.2 defvar \$useFullScreenHelp

— initvars —

```
(defvar |$useFullScreenHelp| t "use fullscreen facility, if possible")
```

— helpfullscreen —

```
(|fullscreen|
 "use fullscreen facility, if possible"
 |interpreter|
 LITERALS
 |$useFullScreenHelp|
 (|on| |off|)
 |on|)
```

44.29 history

----- The history Option -----

Description: save workspace values in a history file

The history option may be followed by any one of the following:

-> on
 off

The current setting is indicated.

44.29.1 defvar \$HiFiAccess

— initvars —

```
(defvar |$HiFiAccess| t "save workspace values in a history file")
```

— history —

```
(|history|
 "save workspace values in a history file"
 |interpreter|
 LITERALS
 |$HiFiAccess|
 (|on| |off|)
 |on|)
```

44.30 messages

Current Values of messages Variables

Variable	Description	Current Value

autoload	print file auto-load messages	off
bottomup	display bottom up modemap selection	off
coercion	display datatype coercion messages	off

dropmap	display old map defn when replaced	off
expose	warning for unexposed functions	off
file	print msgs also to SPADMSG LISTING	off
frame	display messages about frames	off
highlighting	use highlighting in system messages	off
instant	present instantiation summary	off
insteach	present instantiation info	off
interponly	say when function code is interpreted	on
number	display message number with message	off
prompt	set type of input prompt to display	step
selection	display function selection msgs	off
set	show)set setting after assignment	off
startup	display messages on start-up	off
summary	print statistics after computation	off
testing	print system testing header	off
time	print timings after computation	off
type	print type after computation	on
void	print Void value when it occurs	off
any	print the internal type of objects of domain Any	on
naglink	show NAGLink messages	on

— messages —

```
(|messages|
  "show messages for various system features"
  |interpreter|
  TREE
  |novar|
  (
    \getchunk{messagesany}
    \getchunk{messagesautoload}
    \getchunk{messagesbottomup}
    \getchunk{messagescoercion}
    \getchunk{messagesdropmap}
    \getchunk{messagesexpose}
    \getchunk{messagesfile}
    \getchunk{messagesframe}
    \getchunk{messageshighlighting}
    \getchunk{messagesinstant}
    \getchunk{messagesinsteach}
    \getchunk{messagesinterponly}
    \getchunk{messagesnaglink}
    \getchunk{messagesnumber}
    \getchunk{messagesprompt}
    \getchunk{messagesselection}
    \getchunk{messagesset}
    \getchunk{messagesstartup}
    \getchunk{messagessummary}
```



```

\getchunk{messagestesting}
\getchunk{messagestime}
\getchunk{messagestype}
\getchunk{messagesvoid}
))

```

44.30.1 any

----- The any Option -----

Description: print the internal type of objects of domain Any

The any option may be followed by any one of the following:

```

-> on
    off

```

The current setting is indicated.

44.30.2 defvar \$printAnyIfTrue

— initvars —

```

(defvar |$printAnyIfTrue| t
  "print the internal type of objects of domain Any")

```

— messagesany —

```

(|any|
  "print the internal type of objects of domain Any"
  |interpreter|
  LITERALS
  |$printAnyIfTrue|
  (|on| |off|)
  |on|)

```

44.30.3 autoload

----- The autoload Option -----

Description: print file auto-load messages

44.30.4 defvar \$printLoadMsgs

— initvars —

```
(defvar |$printLoadMsgs| nil "print file auto-load messages")
```

—————

— messagesautoload —

```
(|autoload|
 "print file auto-load messages"
 |interpreter|
 LITERALS
 |$printLoadMsgs|
 (|on| |off|)
 |on|)
```

—————

44.30.5 bottomup

----- The bottomup Option -----

Description: display bottom up modemap selection

The bottomup option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.6 defvar \$reportBottomUpFlag

— initvars —

```
(defvar |$reportBottomUpFlag| nil "display bottom up modemap selection")
```

—————

— messagesbottomup —

```
(|bottomup|
 "display bottom up modemap selection"
 |development|
 LITERALS
 |$reportBottomUpFlag|
 (|on| |off|)
 |off|)
```

—————

44.30.7 coercion

----- The coercion Option -----

Description: display datatype coercion messages

The coercion option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.8 defvar \$reportCoerceIfTrue

— initvars —

```
(defvar |$reportCoerceIfTrue| nil "display datatype coercion messages")
```

—————

— messagescoercion —

```
(|coercion|
 "display datatype coercion messages"
 |development|
 LITERALS
 |$reportCoerceIfTrue|
 (|on| |off|)
 |off|)
```

44.30.9 dropmap

----- The dropmap Option -----

Description: display old map defn when replaced

The dropmap option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.10 defvar \$displayDroppedMap

— initvars —

```
(defvar |$displayDroppedMap| nil "display old map defn when replaced")
```

— messagesdropmap —

```
(|dropmap|
 "display old map defn when replaced"
 |interpreter|
 LITERALS
 |$displayDroppedMap|
 (|on| |off|)
```

```
|off|)
```

44.30.11 expose

----- The expose Option -----

Description: warning for unexposed functions

The expose option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.12 defvar \$giveExposureWarning

— initvars —

```
(defvar |$giveExposureWarning| nil "warning for unexposed functions")
```

— messagesexpose —

```
(|expose|
 "warning for unexposed functions"
 |interpreter|
 LITERALS
 |$giveExposureWarning|
 (|on| |off|)
 |off|)
```

44.30.13 file

----- The file Option -----

Description: print msgs also to SPADMSG LISTING

The file option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.14 defvar \$printMsgsToFile

— initvars —

```
(defvar |$printMsgsToFile| nil "print msgs also to SPADMSG LISTING")
```

— messagesfile —

```
(|file|
 "print msgs also to SPADMSG LISTING"
 |development|
 LITERALS
 |$printMsgsToFile|
 (|on| |off|)
 |off|)
```

44.30.15 frame

----- The frame Option -----

Description: display messages about frames

The frame option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.16 defvar \$frameMessages

— initvars —

```
(defvar |$frameMessages| nil "display messages about frames")
```

—————

— messagesframe —

```
(|frame|
 "display messages about frames"
 |interpreter|
 LITERALS
 |$frameMessages|
 (|on| |off|)
 |off|)
```

—————

44.30.17 highlighting

----- The highlighting Option -----

Description: use highlighting in system messages

The highlighting option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.18 defvar \$highlightAllowed

— initvars —

```
(defvar |$highlightAllowed| nil "use highlighting in system messages")
```

—————

— messageshighlighting —

```
(|highlighting|
 "use highlighting in system messages"
 |interpreter|
 LITERALS
 |$highlightAllowed|
 (|on| |off|)
 |off|)
```

44.30.19 instant

----- The instant Option -----

Description: present instantiation summary

The instant option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.20 defvar \$reportInstantiations

— initvars —

```
(defvar |$reportInstantiations| nil "present instantiation summary")
```

— messagesinstant —

```
(|instant|
 "present instantiation summary"
 |development|
 LITERALS
 |$reportInstantiations|
 (|on| |off|)
```



```
|off|)
```

44.30.21 insteach

----- The insteach Option -----

Description: present instantiation info

The insteach option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.22 defvar \$reportEachInstantiation—

— initvars —

```
(defvar |$reportEachInstantiation| nil "present instantiation info")
```

— messagesinsteach —

```
(|insteach|
 "present instantiation info"
 |development|
 LITERALS
 |$reportEachInstantiation|
 (|on| |off|)
 |off|)
```

44.30.23 interponly

----- The interponly Option -----

Description: say when function code is interpreted

The interponly option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.30.24 defvar \$reportInterpOnly

— initvars —

```
(defvar |$reportInterpOnly| t "say when function code is interpreted")
```

—————

— messagesinterponly —

```
(|interponly|
 "say when function code is interpreted"
 |interpreter|
 LITERALS
 |$reportInterpOnly|
 (|on| |off|)
 |on|)
```

—————

44.30.25 naglink

----- The naglink Option -----

Description: show NAGLink messages

The naglink option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.30.26 defvar \$nagMessages

— initvars —

```
(defvar |$nagMessages| t "show NAGLink messages")
```

—————

— messagesnaglink —

```
(|naglink|
 "show NAGLink messages"
 |interpreter|
 LITERALS
 |$nagMessages|
 (|on| |off|)
 |on|)
```

—————

44.30.27 number

----- The number Option -----

Description: display message number with message

The number option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.28 defvar \$displayMsgNumber

— initvars —

```
(defvar |$displayMsgNumber| nil "display message number with message")
```

— messagesnumber —

```
(|number|
 "display message number with message"
 |interpreter|
 LITERALS
 |$displayMsgNumber|
 (|on| |off|)
 |off|)
```

44.30.29 prompt

----- The prompt Option -----

Description: set type of input prompt to display

The prompt option may be followed by any one of the following:

```
none
frame
plain
-> step
verbose
```

The current setting is indicated.

44.30.30 defvar \$inputPromptType

— initvars —

```
(defvar |$inputPromptType| '|step| "set type of input prompt to display")
```

— messagesprompt —

```
(|prompt|
  "set type of input prompt to display"
  |interpreter|
  LITERALS
  |$inputPromptType|
  (|none| |frame| |plain| |step| |verbose|)
  |step|)
```

44.30.31 selection

----- The selection Option -----

Description: display function selection msgs

The selection option may be followed by any one of the following:

```
    on
-> off
```

The current setting is indicated.

TPDHERE: This is a duplicate of)set mes bot on because both use the \$reportBottomUpFlag flag

— messageselection —

```
(|selection|
  "display function selection msgs"
  |interpreter|
  LITERALS
  |$reportBottomUpFlag|
  (|on| |off|)
  |off|)
```

44.30.32 set

----- The set Option -----

Description: show)set setting after assignment

The set option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.33 defvar \$displaySetValue

— initvars —

```
(defvar |$displaySetValue| nil "show )set setting after assignment")
```

— messageset —

```
(|set|
 "show )set setting after assignment"
 |interpreter|
 LITERALS
 |$displaySetValue|
 (|on| |off|)
 |off|)
```

44.30.34 startup

----- The startup Option -----

Description: display messages on start-up

The startup option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.35 defvar \$displayStartMsgs

— initvars —

```
(defvar |$displayStartMsgs| t "display messages on start-up")
```

—————

— messagesstartup —

```
(|startup|
 "display messages on start-up"
 |interpreter|
 LITERALS
 |$displayStartMsgs|
 (|on| |off|)
 |on|)
```

—————

44.30.36 summary

----- The summary Option -----

Description: print statistics after computation

The summary option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.30.37 defvar \$printStatsSummaryIfTrue

— initvars —

```
(defvar |$printStatsSummaryIfTrue| nil
 "print statistics after computation")
```

— messagesummary —

```
(|summary|
  "print statistics after computation"
  |interpreter|
  LITERALS
  |$printStatsSummaryIfTrue|
  (|on| |off|)
  |off|)
```

44.30.38 testing

----- The testing Option -----

Description: print system testing header

The testing option may be followed by any one of the following:

```
    on
-> off
```

The current setting is indicated.

44.30.39 defvar \$testingSystem

— initvars —

```
(defvar |$testingSystem| nil "print system testing header")
```

— messagestesting —

```
(|testing|
  "print system testing header"
  |development|
  LITERALS)
```



```
|$testingSystem|
(|on| |off|)
|off|)
```

44.30.40 time

----- The time Option -----

Description: print timings after computation

The time option may be followed by any one of the following:

```
on
-> off
long
```

The current setting is indicated.

44.30.41 defvar \$printTimeIfTrue

— initvars —

```
(defvar |$printTimeIfTrue| nil "print timings after computation")
```

— message time —

```
(|time|
 "print timings after computation"
 |interpreter|
 LITERALS
 |$printTimeIfTrue|
 (|on| |off| |long|)
 |off|)
```

44.30.42 type

----- The type Option -----

Description: print type after computation

The type option may be followed by any one of the following:

```
-> on
    off
```

The current setting is indicated.

44.30.43 defvar \$printTypeIfTrue

— initvars —

```
(defvar |$printTypeIfTrue| t "print type after computation")
```

— messagestype —

```
(|type|
 "print type after computation"
 |interpreter|
 LITERALS
 |$printTypeIfTrue|
 (|on| |off|)
 |on|)
```

44.30.44 void

----- The void Option -----

Description: print Void value when it occurs

The void option may be followed by any one of the following:

```
    on
-> off
```

The current setting is indicated.

44.30.45 defvar \$printVoidIfTrue

— initvars —

```
(defvar |$printVoidIfTrue| nil "print Void value when it occurs")
```

—————

— messagesvoid —

```
(|void|
 "print Void value when it occurs"
 |interpreter|
 LITERALS
 |$printVoidIfTrue|
 (|on| |off|)
 |off|)
```

—————

44.31 naglink

Current Values of naglink Variables

Variable	Description	Current Value
host	internet address of host for NAGLink	localhost
persistence	number of (fortran) functions to remember	1
messages	show NAGLink messages	on
double	enforce DOUBLE PRECISION ASPs	on

— naglink —

```
(|naglink|
 "options for NAGLink"
 |interpreter|
 TREE)
```

```
|novar|
(
\getchunk{naglinkhost}
\getchunk{naglinkpersistence}
\getchunk{naglinkmessages}
\getchunk{naglinkdouble}
))
```

44.31.1 host

----- The host Option -----

Description: internet address of host for NAGLink

)set naglink host is used to tell AXIOM which host to contact for a NAGLink request. An Internet address should be supplied. The host specified must be running the NAGLink daemon.

The current setting is localhost

44.31.2 defvar \$nagHost

— initvars —

```
(defvar |$nagHost| "localhost" "internet address of host for NAGLink")
```

— naglinkhost —

```
(|host|
 "internet address of host for NAGLink"
|interpreter|
FUNCTION
|setNagHost|
(("enter host name"
 DIRECTORY
 |$nagHost|
 |chkDirectory|
 "localhost"))
NIL)
```

44.31.3 defun setNagHost

```
[object2String p??]
[describeSetNagHost p732]
[$nagHost p731]
```

— defun setNagHost —

```
(defun |setNagHost| (arg)
  (declare (special |$nagHost|))
  (cond
    ((eq arg '|%initialize%|) (setq |$nagHost| "localhost"))
    ((eq arg '|%display%|) (|object2String| |$nagHost|))
    ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
     (|describeSetNagHost|))
    (t (setq |$nagHost| (|object2String| arg)))))
```

44.31.4 defun describeSetNagHost

```
[sayBrightly p??]
[$nagHost p731]
```

— defun describeSetNagHost —

```
(defun |describeSetNagHost| ()
  (declare (special |$nagHost|))
  (|sayBrightly| (list
    '|%b| ")set naglink host"
    '|%d| "is used to tell AXIOM which host to contact for"
    '|%l| " a NAGLink request. An Internet address should be supplied. The host"
    '|%l| " specified must be running the NAGLink daemon."
    '|%l|
    '|%l| " The current setting is"
    '|%b| |$nagHost|
    '|%d|)))
```

44.31.5 persistence

----- The persistence Option -----

Description: number of (fortran) functions to remember

)set naglink persistence is used to tell the nagd daemon how many ASP source and object files to keep around in case you reuse them. This helps to avoid needless recompilations. The number specified should be a non-negative integer.

The current setting is 1

44.31.6 defvar \$fortPersistence

— initvars —

```
(defvar |$fortPersistence| 1 "number of (fortran) functions to remember")
```

—————

— naglinkpersistence —

```
(|persistence|
  "number of (fortran) functions to remember"
  |interpreter|
  FUNCTION
  |setFortPers|
  (("Requested remote storage (for asps):"
    INTEGER
    |$fortPersistence|
    (0 NIL)
    10))
  NIL)
```

—————

44.31.7 defun setFortPers

```
[describeFortPersistence p734]
[sayMessage p??]
[bright p??]
[terminateSystemCommand p432]
[$fortPersistence p733]
```

— defun setFortPers —

```
(defun |setFortPers| (arg)
  (let (n)
    (declare (special |$fortPersistence|))
    (cond
      ((eq arg '|%initialize%|) (setq |$fortPersistence| 1))
      ((eq arg '|%display%|) |$fortPersistence|)
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeFortPersistence|))
      (t
       (setq n (car arg))
       (cond
         ((or (null (integerp n)) (minusp n))
          (|sayMessage|
           '("Your value of" ,@(|bright| n) "is invalid because ..."))
          (|describeFortPersistence|)
          (|terminateSystemCommand|))
         (t (setq |$fortPersistence| (car arg))))))))
```

44.31.8 defun describeFortPersistence

```
[sayBrightly p??]
[$fortPersistence p733]
```

— defun describeFortPersistence —

```
(defun |describeFortPersistence| ()
  (declare (special |$fortPersistence|))
  (|sayBrightly| (list
    '|%b| "set naglink persistence"
    '|%d| "is used to tell the "
    '|%b| '|nagd|
    '|%d| '| daemon how many ASP|
    '|%l|
    " source and object files to keep around in case you reuse them. This helps"
    '|%l| " to avoid needless recompilations. The number specified should be a "
    '|%l| " non-negative integer."
    '|%l|
    '|%l| " The current setting is"
    '|%b| |$fortPersistence|
    '|%d|)))
```

44.31.9 messages

----- The messages Option -----

Description: show NAGLink messages

The messages option may be followed by any one of the following:

-> on
 off

The current setting is indicated.

TPDHERE: this is the same as)set nag mes on

— **naglinkmessages** —

```
(|messages|
  "show NAGLink messages"
  |interpreter|
  LITERALS
  |$nagMessages|
  (|on| |off|)
  |on|)
```

—————

44.31.10 double

----- The double Option -----

Description: enforce DOUBLE PRECISION ASPs

The double option may be followed by any one of the following:

-> on
 off

The current setting is indicated.

44.31.11 defvar \$nagEnforceDouble

— **initvars** —


```
(defvar |$nagEnforceDouble| t "enforce DOUBLE PRECISION ASPs")
```

— **naglinkdouble** —

```
(|double|
 "enforce DOUBLE PRECISION ASPs"
 |interpreter|
 LITERALS
 |$nagEnforceDouble|
 (|on| |off|)
 |on|)
```

44.32 output

The result of the **)set output** command is:

Variable	Description	Current Value
abbreviate	abbreviate type names	off
algebra	display output in algebraic form	On:CONSOLE
characters	choose special output character set	plain
fortran	create output in FORTRAN format	Off:CONSOLE
fraction	how fractions are formatted	vertical
html	create output in HTML style	Off:CONSOLE
length	line length of output displays	77
mathml	create output in MathML style	Off:CONSOLE
openmath	create output in OpenMath style	Off:CONSOLE
script	display output in SCRIPT formula format	Off:CONSOLE
scripts	show subscripts,... linearly	off
showeditor	view output of)show in editor	off
tex	create output in TeX style	Off:CONSOLE

Since the output option has a bunch of sub-options each suboption is defined within the output structure.

— **output** —

```
(|output|
 "view and set some output options"
 |interpreter|
 TREE
 |novar|)
```

```
(
\getchunk{outputabbreviate}
\getchunk{outputalgebra}
\getchunk{outputcharacters}
\getchunk{outputfortran}
\getchunk{outputfraction}
\getchunk{outputhtml}
\getchunk{outputlength}
\getchunk{outputmathml}
\getchunk{outputopenmath}
\getchunk{outputscript}
\getchunk{outputscripts}
\getchunk{outputshoweditor}
\getchunk{outputtex}
))
```

44.32.1 abbreviate

----- The abbreviate Option -----

Description: abbreviate type names

The abbreviate option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.32.2 defvar \$abbreviateTypes

— initvars —

```
(defvar |$abbreviateTypes| nil "abbreviate type names")
```

— outputabbreviate —

```
(|abbreviate|
"abbreviate type names"
```

```
|interpreter|
LITERALS
|$abbreviateTypes|
(|on| |off|)
|off|)
```

44.32.3 algebra

----- The algebra Option -----

Description: display output in algebraic form

)set output algebra is used to tell AXIOM to turn algebra-style output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Syntax:)set output algebra <arg>

where arg can be one of

on	turn algebra printing on (default state)
off	turn algebra printing off
console	send algebra output to screen (default state)
fp<.fe>	send algebra output to file with file prefix fp and file extension .fe. If not given, .fe defaults to .spout.

If you wish to send the output to a file, you may need to issue this command twice: once with on and once with the file name. For example, to send algebra output to the file polymer.spout, issue the two commands

```
)set output algebra on
)set output algebra polymer
```

The output is placed in the directory from which you invoked AXIOM or the one you set with the)cd system command. The current setting is: On:CONSOLE

44.32.4 defvar \$algebraFormat

— initvars —

```
(defvar |$algebraFormat| t "display output in algebraic form")
```

44.32.5 defvar \$algebraOutputFile

— initvars —

```
(defvar |$algebraOutputFile| "CONSOLE"
  "where algebra printing goes (enter {\em console} or a pathname)?")
```

— outputalgebra —

```
(|algebra|
  "display output in algebraic form"
  |interpreter|
  FUNCTION
  |setOutputAlgebra|
  (("display output in algebraic form"
    LITERALS
    |$algebraFormat|
    (|off| |on|)
    |on|)
  (break $algebraFormat)
  ("where algebra printing goes (enter {\em console} or a pathname)?"
  FILENAME
  |$algebraOutputFile|
  |chkOutputFileName|
  "console"))
NIL)
```

44.32.6 defvar \$algebraOutputStream

— initvars —

```
(defvar |$algebraOutputStream| *standard-output*)
```

44.32.7 defun setOutputAlgebra

```
[defiostream p938]
[concat p1003]
[describeSetOutputAlgebra p742]
[pairp p??]
[qcdr p??]
[qcar p??]
[member p1004]
[upcase p??]
[sayKeyedMsg p331]
[shut p938]
[pathnameType p996]
[pathnameDirectory p998]
[pathnameName p996]
[$filep p??]
[make-outstream p937]
[object2String p??]
[$algebraOutputStream p739]
[$algebraOutputFile p739]
[$filep p??]
[$algebraFormat p738]
```

— defun setOutputAlgebra —

```
(defun |setOutputAlgebra| (arg)
  (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$algebraOutputStream| |$algebraOutputFile| $filep
      |$algebraFormat|))
    (cond
      ((eq arg '|%initialize%|)
        (setq |$algebraOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
        (setq |$algebraOutputFile| "CONSOLE")
        (setq |$algebraFormat| t))
      ((eq arg '|%display%|)
        (if |$algebraFormat|
          (setq label "On:")
          (setq label "Off:"))
        (concat label |$algebraOutputFile|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
        (|describeSetOutputAlgebra|))
      (t
        (cond
          ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t)
            (|member| fn '(y n ye yes no o on of off console
```

```

      |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
    'ok|)
  (t (setq arg (list fn 'spout|)))
(cond
  ((and (pairp arg)
        (eq (qcdr arg) nil)
        (progn (setq fn (qcar arg)) t))
   (cond
    ((|member| (upcase fn) '(y n ye o of))
     (|sayKeyedMsg| 's2iv0002 '(|algebra| |algebra|)))
    ((|member| (upcase fn) '(no off)) (setq |$algebraFormat| nil))
    ((|member| (upcase fn) '(yes on)) (setq |$algebraFormat| t))
    ((eq (upcase fn) 'console)
     (shut |$algebraOutputStream|)
     (setq |$algebraOutputStream|
           (defiostream '((mode . output) (device . console)) 255 0))
     (setq |$algebraOutputFile| "CONSOLE"))))
  ((or
    (and (pairp arg)
         (progn
          (setq fn (qcar arg))
          (setq tmp1 (qcdr arg))
          (and (pairp tmp1)
               (eq (qcdr tmp1) nil)
               (progn (setq ft (qcar tmp1)) t))))
    (and (pairp arg)
         (progn (setq fn (qcar arg))
                  (setq tmp1 (qcdr arg))
                  (and (pairp tmp1)
                       (progn (setq ft (qcar tmp1))
                              (setq tmp2 (qcdr tmp1))
                              (and (pairp tmp2)
                                   (eq (qcdr tmp2) nil)
                                   (progn
                                    (setq fm (qcar tmp2))
                                    t))))))))
    (when (setq ptype (|pathnameType| fn))
      (setq fn (concat (|pathnameDirectory| fn) (|pathnameName| fn)))
      (setq ft ptype))
    (unless fm (setq fm 'a))
    (setq filename ($filep fn ft fm))
    (cond
     ((null filename)
      (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
     ((setq teststream (make-outstream filename 255 0))
      (shut |$algebraOutputStream|)
      (setq |$algebraOutputStream| teststream)
      (setq |$algebraOutputFile| (|object2String| filename))
      (|sayKeyedMsg| 's2iv0004 (list "Algebra" |$algebraOutputFile|))
      (t (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))))

```

```
(t
  (|sayKeyedMsg| 's2iv0005 nil)
  (|describeSetOutputAlgebra|))))))
```

44.32.8 defun describeSetOutputAlgebra

```
[sayBrightly p??]
[setOutputAlgebra p740]
```

— defun describeSetOutputAlgebra —

```
(defun |describeSetOutputAlgebra| ()
  (|sayBrightly| (list
    '|%b| ")set output algebra"
    '|%d| "is used to tell AXIOM to turn algebra-style output"
    '|%l| "printing on and off, and where to place the output. By default, the"
    '|%l| "destination for the output is the screen but printing is turned off."
    '|%l|
    '|%l| "Syntax: )set output algebra <arg>"
    '|%l| "      where arg can be one of"
    '|%l| "  on          turn algebra printing on (default state)"
    '|%l| "  off         turn algebra printing off"
    '|%l| "  console      send algebra output to screen (default state)"
    '|%l| "  fp<.fe>      send algebra output to file with file prefix fp"
    '|%l|
    "                        and file extension .fe. If not given, .fe defaults to .spout."
    '|%l|
    '|%l|
    "If you wish to send the output to a file, you may need to issue this command"
    '|%l| "twice: once with"
    '|%b| "on"
    '|%d| "and once with the file name. For example, to send"
    '|%l| "algebra output to the file"
    '|%b| "polymer.spout,"
    '|%d| "issue the two commands"
    '|%l|
    '|%l| " )set output algebra on"
    '|%l| " )set output algebra polymer"
    '|%l|
    '|%l| "The output is placed in the directory from which you invoked AXIOM or"
    '|%l| "the one you set with the )cd system command."
    '|%l| "The current setting is: "
    '|%b| (|setOutputAlgebra| '|%display%|)
    '|%d|)))
```

44.32.9 characters

----- The characters Option -----

Description: choose special output character set

The characters option may be followed by any one of the following:

default
-> plain

The current setting is indicated. This option determines the special characters used for algebraic output. This is what the current choice of special characters looks like:

ulc is shown as +	urc is shown as +
llc is shown as +	lrc is shown as +
vbar is shown as	hbar is shown as -
quad is shown as ?	lbrk is shown as [
rbrk is shown as]	lbrc is shown as {
rbrc is shown as }	ttee is shown as +
btee is shown as +	rtee is shown as +
ltee is shown as +	ctee is shown as +
bslash is shown as \	

— outputcharacters —

```
(|characters|
 "choose special output character set"
 |interpreter|
 FUNCTION
 |setOutputCharacters|
 NIL
 |htSetOutputCharacters|)
```

44.32.10 defun setOutputCharacters

```
[sayMessage p??]
[bright p??]
```



```

[sayBrightly p??]
[concat p1003]
[pname p1001]
[specialChar p936]
[sayAsManyPerLineAsPossible p??]
[pairp p??]
[qcdr p??]
[qcar p??]
[downcase p??]
[setOutputCharacters p743]
[$specialCharacters p935]
[$plainRTspecialCharacters p934]
[$RTspecialCharacters p934]
[$specialCharacterAlist p935]

```

— **defun setOutputCharacters** —

```

(defun |setOutputCharacters| (arg)
  (let (current char s l fn)
    (declare (special |$specialCharacters| |$plainRTspecialCharacters|
                      |$RTspecialCharacters| |$specialCharacterAlist|))
    (if (eq arg '|%initialize%|)
        (setq |$specialCharacters| |$plainRTspecialCharacters|)
        (progn
          (setq current
                (cond
                 ((eq |$specialCharacters| |$RTspecialCharacters|) "default")
                 ((eq |$specialCharacters| |$plainRTspecialCharacters|) "plain")
                 (t "unknown")))
            (cond
             ((eq arg '|%display%|) current)
             ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
              (|sayMessage|
               '(" The" ,@( |bright| "characters")
                 "option may be followed by any one of the following:"))
              (dolist (name '("default" "plain"))
                (if (string= (string current) name)
                    (|sayBrightly| '(" ->" ,@( |bright| name)))
                    (|sayBrightly| (list " " name))))
              (terpri)
              (|sayBrightly|
               " The current setting is indicated within the list. This option determines ")
              (|sayBrightly|
               " the special characters used for algebraic output. This is what the")
              (|sayBrightly|
               " current choice of special characters looks like:")
              (do ((t1 |$specialCharacterAlist| (CDR t1)) (t2 nil))
                  ((or (atom t1)

```

```

      (progn (setq t2 (car t1)) nil)
      (progn (progn (setq char (car t2)) t2) nil)) nil)
  (setq s
    (concat " " (pname char) " is shown as "
      (pname (|specialChar| char))))
  (setq l (cons s l)))
  (|sayAsManyPerLineAsPossible| (reverse l)))
  ((and (pairp arg)
    (eq (qcdr arg) NIL)
    (progn (setq fn (qcar arg)) t)
    (setq fn (downcase fn))))
  (cond
    ((eq fn '|default|)
      (setq |$specialCharacters| |$RTspecialCharacters|))
    ((eq fn '|plain|)
      (setq |$specialCharacters| |$plainRTspecialCharacters|))
    (t (|setOutputCharacters| nil))))
  (t (|setOutputCharacters| nil))))))

```

44.32.11 fortran

----- The fortran Option -----

Description: create output in FORTRAN format

)set output fortran is used to tell AXIOM to turn FORTRAN-style output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Also See:)set fortran

Syntax:)set output fortran <arg>

where arg can be one of

on	turn FORTRAN printing on
off	turn FORTRAN printing off (default state)
console	send FORTRAN output to screen (default state)
fp<.fe>	send FORTRAN output to file with file prefix fp and file extension .fe. If not given, .fe defaults to .sfort.

If you wish to send the output to a file, you must issue this command twice: once with on and once with the file name. For example, to send FORTRAN output to the file polymer.sfort, issue the two commands

```

)set output fortran on
)set output fortran polymer

```

The output is placed in the directory from which you invoked AXIOM or the one you set with the)cd system command.
The current setting is: Off:CONSOLE

44.32.12 defvar \$fortranFormat

— initvars —

```
(defvar |$fortranFormat| nil "create output in FORTRAN format")
```

—————

44.32.13 defvar \$fortranOutputFile

— initvars —

```
(defvar |$fortranOutputFile| "CONSOLE"
  "where FORTRAN output goes (enter {\em console} or a a pathname)")
```

—————

— outputfortran —

```

(|fortran|
 "create output in FORTRAN format"
 |interpreter|
 FUNCTION
 |setOutputFortran|
 (("create output in FORTRAN format"
  LITERALS
  |$fortranFormat|
  (|off| |on|)
  |off|)
 (|break| |$fortranFormat|)
 ("where FORTRAN output goes (enter {\em console} or a a pathname)"
  FILENAME
  |$fortranOutputFile|
  |chkOutputFileName|
  "console"))
 NIL)

```

44.32.14 defun setOutputFortran

[defiostream p938]
 [concat p1003]
 [describeSetOutputFortran p749]
 [upcase p??]
 [pairp p??]
 [qcdr p??]
 [qcar p??]
 [member p1004]
 [sayKeyedMsg p331]
 [shut p938]
 [pathnameType p996]
 [pathnameDirectory p998]
 [pathnameName p996]
 [\$filep p??]
 [makeStream p939]
 [object2String p??]
 [\$fortranOutputStream p??]
 [\$fortranOutputFile p746]
 [\$filep p??]
 [\$fortranFormat p746]

— defun setOutputFortran —

```
(defun |setOutputFortran| (arg)
  (let (label APPEND quiet tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$fortranOutputStream| |$fortranOutputFile| $filep
      |$fortranFormat|))
    (cond
      ((eq arg '|%initialize%|)
        (setq |$fortranOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
        (setq |$fortranOutputFile| "CONSOLE")
        (setq |$fortranFormat| nil))
      ((eq arg '|%display%|)
        (if |$fortranFormat|
          (setq label "On:")
          (setq label "Off:"))
        (concat label |$fortranOutputFile|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
        (|describeSetOutputFortran|))
      (t
        (DO ()
          ((null (and (listp arg)
```

```

(|member| (upcase (car arg)) '(append quiet)))
nil)
(cond
  ((eq (upcase (car arg)) 'append) (setq append t))
  ((eq (upcase (car arg)) 'quiet) (setq quiet t))
  (t nil))
(setq arg (cdr arg)))
(cond
  ((and (pairp arg)
        (eq (qcdr arg) nil)
        (progn (setq fn (qcar arg)) t)
        (|member| fn '(Y N YE YES NO O ON OF OFF CONSOLE
                      |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
   '|ok|)
   (t (setq arg (list fn '|sfort|))))
(cond
  ((and (pairp arg) (eq (qcdr arg) nil) (progn (setq fn (qcar arg)) t))
   (cond
    ((|member| (upcase fn) '(y n ye o of))
     (|sayKeyedMsg| 's2iv0002 '(fortran |fortran|)))
    ((|member| (upcase fn) '(no off)) (setq |$fortranFormat| nil))
    ((|member| (upcase fn) '(yes on)) (setq |$fortranFormat| t))
    ((eq (upcase fn) 'console)
     (shut |$fortranOutputStream|)
     (setq |$fortranOutputStream|
           (defiostream '((mode . output) (device . console)) 255 0))
     (setq |$fortranOutputFile| "CONSOLE"))))
   (or
    (and (pairp arg)
         (progn
          (setq fn (qcar arg))
          (setq tmp1 (qcdr arg))
          (and (pairp tmp1)
               (eq (qcdr tmp1) nil)
               (progn (setq ft (qcar tmp1)) t))))
         (and (pairp arg)
              (progn
               (setq fn (qcar arg))
               (setq tmp1 (qcdr arg))
               (and (pairp tmp1)
                    (progn
                     (setq ft (qcar tmp1))
                     (setq tmp2 (qcdr tmp1))
                     (and (pairp tmp2)
                          (eq (qcdr tmp2) nil)
                          (progn (setq fm (qcar tmp2)) t))))))))
    (when (setq ptype (|pathnameType| fn))
      (setq fn (concat (|pathnameDirectory| fn) (|pathnameName| fn)))
      (setq ft ptype))
    (unless fm (setq fm 'a))

```

```

(setq filename ($filep fn ft fm))
(cond
  ((null filename)
   (|sayKeyedMsg| 'S2IV0003 (list fn ft fm)))
  ((setq teststream (|makeStream| append filename 255 0))
   (SHUT |$fortranOutputStream|)
   (setq |$fortranOutputStream| teststream)
   (setq |$fortranOutputFile| (|object2String| filename))
   (unless quiet
    (|sayKeyedMsg| 'S2IV0004 (list 'fortran |$fortranOutputFile|))))
  ((null quiet)
   (|sayKeyedMsg| 'S2IV0003 (list fn ft fm)))
  (t nil)))
(t
 (unless quiet (|sayKeyedMsg| 'S2IV0005 nil))
 (|describeSetOutputFortran|))))))

```

44.32.15 defun describeSetOutputFortran

```

[|sayBrightly| p??]
[|setOutputFortran| p747]

```

— defun describeSetOutputFortran —

```

(defun |describeSetOutputFortran| ()
  (|sayBrightly| (list
    '|%b| " )set output fortran"
    '|%d| "is used to tell AXIOM to turn FORTRAN-style output"
    '|%l| "printing on and off, and where to place the output. By default, the"
    '|%l| "destination for the output is the screen but printing is turned off."
    '|%l|
    '|%l| "Also See: )set fortran"
    '|%l|
    '|%l| "Syntax: )set output fortran <arg>"
    '|%l| " where arg can be one of"
    '|%l| " on turn FORTRAN printing on"
    '|%l| " off turn FORTRAN printing off (default state)"
    '|%l| " console send FORTRAN output to screen (default state)"
    '|%l|
    " fp<.fe> send FORTRAN output to file with file prefix fp and file"
    '|%l| " extension .fe. If not given, .fe defaults to .sfort."
    '|%l|
    '|%l| "If you wish to send the output to a file, you must issue this command"
    '|%l| "twice: once with"
    '|%b| "on"
  ))

```

```
'|%d| "and once with the file name. For example, to send"
'|%l| "FORTRAN output to the file"
'|%b| "polymer.sfort,"
'|%d| "issue the two commands"
'|%l|
'|%l| " )set output fortran on"
'|%l| " )set output fortran polymer"
'|%l|
'|%l| "The output is placed in the directory from which you invoked AXIOM or"
'|%l| "the one you set with the )cd system command."
'|%l| "The current setting is: "
'|%b| (|setOutputFortran| '|%display%|)
'|%d|))
```

44.32.16 fraction

----- The fraction Option -----

Description: how fractions are formatted

The fraction option may be followed by any one of the following:

```
-> vertical
    horizontal
```

The current setting is indicated.

44.32.17 defvar \$fractionDisplayType

— initvars —

```
(defvar |$fractionDisplayType| '|vertical| "how fractions are formatted")
```

— outputfraction —

```
(|fraction|
 "how fractions are formatted"
 |interpreter|
 LITERALS
```

```
|$fractionDisplayType|
(|vertical| |horizontal|)
|vertical|)
```

44.32.18 length

----- The length Option -----

Description: line length of output displays

The length option may be followed by an integer in the range 10 to 245 inclusive. The current setting is 77

44.32.19 defvar \$margin

— initvars —

```
(defvar $margin 3)
```

44.32.20 defvar \$linelength

— initvars —

```
(defvar $linelength 77 "line length of output displays")
```

— outputlength —

```
(|length|
 "line length of output displays"
 |interpreter|
 INTEGER
 $LINELENGTH
 (10 245)
```


77)

44.32.21 mathml

----- The mathml Option -----

Description: create output in MathML style

)set output mathml is used to tell AXIOM to turn MathML-style output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Syntax:)set output mathml <arg>
 where arg can be one of
 on turn MathML printing on
 off turn MathML printing off (default state)
 console send MathML output to screen (default state)
 fp<.fe> send MathML output to file with file prefix fp
 and file extension .fe. If not given,
 .fe defaults to .smml.

If you wish to send the output to a file, you must issue this command twice: once with on and once with the file name. For example, to send MathML output to the file polymer.smml, issue the two commands

```
)set output mathml on
)set output mathml polymer
```

The output is placed in the directory from which you invoked AXIOM or the one you set with the)cd system command. The current setting is: Off:CONSOLE

44.32.22 defvar \$mathmlFormat

— initvars —

```
(defvar |$mathmlFormat| nil "create output in MathML format")
```

44.32.23 defvar \$mathmlOutputFile

— initvars —

```
(defvar |$mathmlOutputFile| "CONSOLE"
  "where MathML output goes (enter {\em console} or a pathname)")
```

—————

— outputmathml —

```
(|mathml|
  "create output in MathML style"
  |interpreter|
  FUNCTION
  |setOutputMathml|
  (("create output in MathML format"
    LITERALS
    |$mathmlFormat|
    (|off| |on|)
    |off|)
  (|break| |$mathmlFormat|)
  ("where MathML output goes (enter {\em console} or a pathname)"
  FILENAME
  |$mathmlOutputFile|
  |chkOutputFileName|
  "console"))
  NIL)
```

—————

44.32.24 defun setOutputMathml

```
[defiostream p938]
[concat p1003]
[describeSetOutputMathml p755]
[pairp p??]
[qcdr p??]
[qcar p??]
[member p1004]
[upcase p??]
[sayKeyedMsg p331]
[shut p938]
[pathnameType p996]
```

```
[pathnameDirectory p998]
[pathnameName p996]
[$filep p??]
[make-outstream p937]
[object2String p??]
[$mathmlOutputStream p??]
[$mathmlOutputFile p753]
[$mathmlFormat p752]
[$filep p??]
```

— defun setOutputMathml —

```
(defun |setOutputMathml| (arg)
  (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$mathmlOutputStream| |$mathmlOutputFile| |$mathmlFormat|
                      $filep))
    (cond
      ((eq arg '|%initialize%|)
       (setq |$mathmlOutputStream|
             (defiostream '((mode . output) (device . console)) 255 0))
       (setq |$mathmlOutputFile| "CONSOLE")
       (setq |$mathmlFormat| nil))
      ((eq arg '|%display%|)
       (if |$mathmlFormat|
          (setq label "On:")
          (setq label "Off:"))
       (concat label |$mathmlOutputFile|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeSetOutputMathml|))
      (t
       (cond
         ((and (pairp arg)
              (eq (qcdr arg) nil)
              (progn (setq fn (qcar arg)) t)
              (|member| fn '(y n ye yes no o on of off console
                           |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
          '|ok|)
         (t (setq arg (list fn '|smml|))))
       (cond
         ((and (pairp arg)
              (eq (qcdr arg) nil)
              (progn (setq fn (qcar arg)) t))
          (cond
            ((|member| (upcase fn) '(y n ye o of))
             (|sayKeyedMsg| 's2iv0002 '(|MathML| |mathml|)))
            ((|member| (upcase fn) '(no off)) (setq |$mathmlFormat| nil))
            ((|member| (upcase fn) '(yes on)) (setq |$mathmlFormat| t))
            ((eq (upcase fn) 'console)
             (shut |$mathmlOutputStream|))
```

```

(setq |$mathmlOutputStream|
  (defiostream '((mode . output) (device . console)) 255 0))
(setq |$mathmlOutputFile| "CONSOLE"))))
((or
  (and (pairp arg)
    (progn
      (setq fn (qcar arg))
      (setq tmp1 (qcdr arg))
      (and (pairp tmp1)
        (eq (qcdr tmp1) nil)
        (progn (setq ft (qcar tmp1)) t))))
    (and (pairp arg)
      (progn (setq fn (qcar arg))
        (setq tmp1 (qcdr arg))
        (and (pairp tmp1)
          (progn
            (setq ft (qcar tmp1))
            (setq tmp2 (qcdr tmp1))
            (and (pairp tmp2)
              (eq (qcdr tmp2) nil)
              (progn
                (setq fm (qcar tmp2))
                t))))))))
    (when (setq ptype (|pathnameType| fn))
      (setq fn
        (concat (|pathnameDirectory| fn) (|pathnameName| fn)))
      (setq ft ptype))
    (unless fm (setq fm 'a))
    (setq filename ($filep fn ft fm))
    (cond
      ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
      ((setq teststream (make-outstream filename 255 0))
        (shut |$mathmlOutputStream|)
        (setq |$mathmlOutputStream| teststream)
        (setq |$mathmlOutputFile| (|object2String| filename))
        (|sayKeyedMsg| 's2iv0004 (list "MathML" |$mathmlOutputFile|)))
      (t (|sayKeyedMsg| 's2iv0003 (list fn ft fm))))))
(t
  (|sayKeyedMsg| 's2iv0005 nil)
  (|describeSetOutputMathml|))))))

```

44.32.25 defun describeSetOutputMathml

```

[sayBrightly p??]
[setOutputMathml p753]

```

— defun describeSetOutputMathml —

```
(defun |describeSetOutputMathml| ()
  (|sayBrightly| (LIST
    '|%b| ")set output mathml"
    '|%d| "is used to tell AXIOM to turn MathML-style output"
    '|%l| "printing on and off, and where to place the output. By default, the"
    '|%l| "destination for the output is the screen but printing is turned off."
    '|%l|
    '|%l| "Syntax:  )set output mathml <arg>"
    '|%l| "   where arg can be one of"
    '|%l| "   on           turn MathML printing on"
    '|%l| "   off          turn MathML printing off (default state)"
    '|%l| "   console      send MathML output to screen (default state)"
    '|%l| "   fp<.fe>       send MathML output to file with file prefix fp and file"
    '|%l| "                   extension .fe. If not given, .fe defaults to .stex."
    '|%l|
    '|%l| "If you wish to send the output to a file, you must issue this command"
    '|%l| "twice: once with"
    '|%b| "on"
    '|%d| "and once with the file name. For example, to send"
    '|%l| "MathML output to the file"
    '|%b| "polymer.smml,"
    '|%d| "issue the two commands"
    '|%l|
    '|%l| "   )set output mathml on"
    '|%l| "   )set output mathml polymer"
    '|%l|
    '|%l| "The output is placed in the directory from which you invoked AXIOM or"
    '|%l| "the one you set with the )cd system command."
    '|%l| "The current setting is: "
    '|%b| (|setOutputMathml| '|%display%|)
    '|%d|)))
```

44.32.26 html

----- The html Option -----

Description: create output in html style

)set output html is used to tell AXIOM to turn html-style output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Syntax: `)set output html <arg>`
 where `arg` can be one of

<code>on</code>	turn html printing on
<code>off</code>	turn html printing off (default state)
<code>console</code>	send html output to screen (default state)
<code>fp<.fe></code>	send html output to file with file prefix <code>fp</code> and file extension <code>.fe</code> . If not given, <code>.fe</code> defaults to <code>.html</code> .

If you wish to send the output to a file, you must issue this command twice: once with `on` and once with the file name. For example, to send MathML output to the file `polymer.html`, issue the two commands

```
)set output html on
)set output html polymer
```

The output is placed in the directory from which you invoked Axiom or the one you set with the `)cd` system command. The current setting is: `Off:CONSOLE`

44.32.27 defvar \$htmlFormat

— initvars —

```
(defvar |$htmlFormat| nil "create output in HTML format")
```

—————

44.32.28 defvar \$htmlOutputFile

— initvars —

```
(defvar |$htmlOutputFile| "CONSOLE"
  "where HTML output goes (enter {\em console} or a pathname)")
```

—————

— outputhtml —

```
(|html|
  "create output in HTML style"
  |interpreter|
```

```

FUNCTION
|setOutputHtml|
(("create output in HTML format"
  LITERALS
  |$htmlFormat|
  (|off| |on|)
  |off|)
(|break| |$htmlFormat|)
("where HTML output goes (enter {\em console} or a pathname)"
  FILENAME
  |$htmlOutputFile|
  |chkOutputFileName|
  "console"))
NIL)

```

44.32.29 defun setOutputHtml

```

[defiostream p938]
[concat p1003]
[describeSetOutputHtml p760]
[pairp p??]
[qcdr p??]
[qcar p??]
[member p1004]
[upcase p??]
[sayKeyedMsg p331]
[shut p938]
[pathnameType p996]
[pathnameDirectory p998]
[pathnameName p996]
[$filep p??]
[make-outstream p937]
[object2String p??]
[$htmlOutputStream p??]
[$htmlOutputFile p757]
[$htmlFormat p757]
[$filep p??]

```

— defun setOutputHtml —

```

(defun |setOutputHtml| (arg)
  (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$htmlOutputStream| |$htmlOutputFile| |$htmlFormat|
                      $filep)))

```

```

(cond
  ((eq arg '|%initialize%|)
    (setq |$htmlOutputStream|
      (defiostream '((mode . output) (device . console)) 255 0))
    (setq |$htmlOutputFile| "CONSOLE")
    (setq |$htmlFormat| nil))
  ((eq arg '|%display%|)
    (if |$htmlFormat|
      (setq label "On:")
      (setq label "Off:"))
    (concat label |$htmlOutputFile|))
  ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
    (describeSetOutputHtml)))
(t
  (cond
    ((and (pairp arg)
      (eq (qcdr arg) nil)
      (progn (setq fn (qcar arg)) t)
      (|member| fn '(y n ye yes no o on of off console
        |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|))))
      'ok|)
    (t (setq arg (list fn '|smml|)))))
  (cond
    ((and (pairp arg)
      (eq (qcdr arg) nil)
      (progn (setq fn (qcar arg)) t))
      (cond
        ((|member| (upcase fn) '(y n ye o of))
          (|sayKeyedMsg| 's2iv0002 '(|HTML| |html|)))
        ((|member| (upcase fn) '(no off)) (setq |$htmlFormat| nil))
        ((|member| (upcase fn) '(yes on)) (setq |$htmlFormat| t))
        ((eq (upcase fn) 'console)
          (shut |$htmlOutputStream|)
          (setq |$htmlOutputStream|
            (defiostream '((mode . output) (device . console)) 255 0))
          (setq |$htmlOutputFile| "CONSOLE"))))
      ((or
        (and (pairp arg)
          (progn
            (setq fn (qcar arg))
            (setq tmp1 (qcdr arg))
            (and (pairp tmp1)
              (eq (qcdr tmp1) nil)
              (progn (setq ft (qcar tmp1)) t))))
          (and (pairp arg)
            (progn (setq fn (qcar arg))
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                (progn
                  (setq ft (qcar tmp1))

```



```

        (setq tmp2 (qcdr tmp1))
        (and (pairp tmp2)
              (eq (qcdr tmp2) nil)
              (progn
                (setq fm (qcar tmp2))
                t))))))
(when (setq ptype (|pathnameType| fn))
  (setq fn
    (concat (|pathnameDirectory| fn) (|pathnameName| fn)))
  (setq ft ptype))
(unless fm (setq fm 'a))
(setq filename ($filep fn ft fm))
(cond
  ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
  ((setq teststream (make-outstream filename 255 0))
   (shut |$htmlOutputStream|)
   (setq |$htmlOutputStream| teststream)
   (setq |$htmlOutputFile| (|object2String| filename))
   (|sayKeyedMsg| 's2iv0004 (list "HTML" |$htmlOutputFile|)))
  (t (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))))
(t
  (|sayKeyedMsg| 's2iv0005 nil)
  (|describeSetOutputHtml|))))))

```

44.32.30 defun describeSetOutputHtml

```

[sayBrightly p??]
[setOutputHtml p758]

```

— defun describeSetOutputHtml —

```

(defun |describeSetOutputHtml| ()
  (|sayBrightly| (LIST
    '|%b| ")set output html"
    '|%d| "is used to tell AXIOM to turn HTML-style output"
    '|%l| "printing on and off, and where to place the output. By default, the"
    '|%l| "destination for the output is the screen but printing is turned off."
    '|%l|
    '|%l| "Syntax:   )set output html <arg>"
    '|%l| "   where arg can be one of"
    '|%l| "   on       turn HTML printing on"
    '|%l| "   off      turn HTML printing off (default state)"
    '|%l| "   console  send HTML output to screen (default state)"
    '|%l| "   fp<.fe>  send HTML output to file with file prefix fp and file"
    '|%l| "           extension .fe. If not given, .fe defaults to .stex."
  ))

```

```
'|%l|
'|%l| "If you wish to send the output to a file, you must issue this command"
'|%l| "twice: once with"
'|%b| "on"
'|%d| "and once with the file name. For example, to send"
'|%l| "HTML output to the file"
'|%b| "polymer.smml,"
'|%d| "issue the two commands"
'|%l|
'|%l| " )set output html on"
'|%l| " )set output html polymer"
'|%l|
'|%l| "The output is placed in the directory from which you invoked AXIOM or"
'|%l| "the one you set with the )cd system command."
'|%l| "The current setting is: "
'|%b| (|setOutputHtml| '|%display%|)
'|%d|)))
```

44.32.31 openmath

----- The openmath Option -----

Description: create output in OpenMath style

`)set output tex` is used to tell AXIOM to turn OpenMath output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Syntax: `)set output tex <arg>`

where `arg` can be one of

<code>on</code>	turn OpenMath printing on
<code>off</code>	turn OpenMath printing off (default state)
<code>console</code>	send OpenMath output to screen (default state)
<code>fp<.fe></code>	send OpenMath output to file with file prefix <code>fp</code> and file extension <code>.fe</code> . If not given, <code>.fe</code> defaults to <code>.sopen</code> .

If you wish to send the output to a file, you must issue this command twice: once with `on` and once with the file name. For example, to send OpenMath output to the file `polymer.sopen`, issue the two commands

```
)set output openmath on
)set output openmath polymer
```

The output is placed in the directory from which you invoked AXIOM or the one you set with the)cd system command.
The current setting is: Off:CONSOLE

44.32.32 defvar \$OpenMathFormat

— initvars —

```
(defvar |$OpenMathFormat| nil "create output in OpenMath format")
```

—————

44.32.33 defvar \$OpenMathOutputFile

— initvars —

```
(defvar |$OpenMathOutputFile| "CONSOLE"
  "where TeX output goes (enter {\em console} or a pathname)")
```

—————

— outputopenmath —

```
(|openmath|
  "create output in OpenMath style"
  |interpreter|
  FUNCTION
  |setOutputOpenMath|
  (("create output in OpenMath format"
    LITERALS
    |$OpenMathFormat|
    (|off| |on|)
    |off|)
  (|break| |$OpenMathFormat|)
  ("where TeX output goes (enter {\em console} or a pathname)"
  FILENAME
  |$OpenMathOutputFile|
  |chkOutputFileName|
  "console"))
NIL)
```

—————

44.32.34 defun setOutputOpenMath

```

[defiostream p938]
[concat p1003]
[describeSetOutputOpenMath p765]
[pairp p??]
[qcdr p??]
[qcar p??]
[member p1004]
[upcase p??]
[sayKeyedMsg p331]
[shut p938]
[pathnameType p996]
[pathnameDirectory p998]
[pathnameName p996]
[$filep p??]
[make-outstream p937]
[object2String p??]
[$openMathOutputStream p??]
[$openMathFormat p762]
[$filep p??]
[$openMathOutputFile p762]

```

— **defun setOutputOpenMath** —

```

(defun |setOutputOpenMath| (arg)
  (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$openMathOutputStream| |$openMathFormat| $filep
                      |$openMathOutputFile|))
    (cond
      ((eq arg '|%initialize%|)
       (setq |$openMathOutputStream|
              (defiostream '((mode . output) (device . console)) 255 0))
       (setq |$openMathOutputFile| "CONSOLE")
       (setq |$openMathFormat| NIL))
      ((eq arg '|%display%|)
       (if |$openMathFormat|
           (setq label "On:")
           (setq label "Off:"))
       (concat label |$openMathOutputFile|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeSetOutputOpenMath|))
      (t
       (cond
         ((and (pairp arg)
                (eq (qcdr arg) nil)
                (progn (setq fn (qcar arg)) t)
                (|member| fn '(y n ye yes no o on of off console)

```

```

        |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
      '|ok|)
    (t (setq arg (list fn '|som|))))
  (cond
    ((and (pairp arg)
          (eq (qcdr arg) nil)
          (progn (setq fn (qcar arg)) t))
     (cond
       ((|member| (upcase fn) '(y n ye o of))
        (|sayKeyedMsg| 's2iv0002 '(|OpenMath| |openmath|)))
       ((|member| (upcase fn) '(no off)) (setq |$openMathFormat| nil))
       ((|member| (upcase fn) '(yes on)) (setq |$openMathFormat| t))
       ((eq (upcase fn) 'console)
        (shut |$openMathOutputStream|)
        (setq |$openMathOutputStream|
              (defiostream '((mode . output) (device . console)) 255 0))
        (setq |$openMathOutputFile| "CONSOLE"))))
    ((or
      (and (pairp arg)
            (progn (setq fn (qcar arg))
                    (setq tmp1 (qcdr arg))
                    (and (pairp tmp1)
                         (eq (qcdr tmp1) nil)
                         (progn (setq ft (qcar tmp1)) t))))
      (and (pairp arg)
            (progn
              (setq fn (qcar arg))
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                    (progn (setq ft (qcar tmp1))
                          (setq tmp2 (qcdr tmp1))
                          (and (pairp tmp2)
                               (eq (qcdr tmp2) nil)
                               (progn (setq fm (qcar tmp2)) t)))))))
      (when (setq ptype (|pathnameType| fn))
        (setq fn (concat (|pathnameDirectory| fn) (|pathnameName| fn)))
        (setq ft ptype))
      (unless fm (setq fm 'a))
      (setq filename ($filep fn ft fm))
      (cond
        ((null filename)
         (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
        ((setq teststream (make-outstream filename 255 0))
         (shut |$openMathOutputStream|)
         (setq |$openMathOutputStream| teststream)
         (setq |$openMathOutputFile| (|object2String| filename))
         (|sayKeyedMsg| 's2iv0004 (list "OpenMath" |$openMathOutputFile|)))
        (t
         (|sayKeyedMsg| 's2iv0003 (list fn ft fm))))
    (t

```

```
(|sayKeyedMsg| 's2iv0005 nil)
(|describeSetOutputOpenMath|))))))
```

44.32.35 defun describeSetOutputOpenMath

```
[sayBrightly p??]
[setOutputOpenMath p763]
```

— defun describeSetOutputOpenMath —

```
(defun |describeSetOutputOpenMath| ()
  (|sayBrightly| (list
    '|%b| ")set output openmath"
    '|%d| "is used to tell AXIOM to turn OpenMath output"
    '|%l| "printing on and off, and where to place the output. By default, the"
    '|%l| "destination for the output is the screen but printing is turned off."
    '|%l|
    '|%l| "Syntax:  )set output openmath <arg>"
    '|%l| "      where arg can be one of"
    '|%l| "      on          turn OpenMath printing on"
    '|%l| "      off         turn OpenMath printing off (default state)"
    '|%l| "      console     send OpenMath output to screen (default state)"
    '|%l|
    "      fp<.fe>      send OpenMath output to file with file prefix fp and file"
    '|%l| "                      extension .fe. If not given, .fe defaults to .som."
    '|%l|
    '|%l| "If you wish to send the output to a file, you must issue this command"
    '|%l| "twice: once with"
    '|%b| "on"
    '|%d| "and once with the file name. For example, to send"
    '|%l| "OpenMath output to the file"
    '|%b| "polymer.som,"
    '|%d| "issue the two commands"
    '|%l|
    '|%l| "      )set output openmath on"
    '|%l| "      )set output openmath polymer"
    '|%l|
    '|%l| "The output is placed in the directory from which you invoked AXIOM or"
    '|%l| "the one you set with the )cd system command."
    '|%l| "The current setting is: "
    '|%b| (|setOutputOpenMath| '|%display%|)
    '|%d|)))
```

44.32.36 script

----- The script Option -----

Description: display output in SCRIPT formula format

)set output script is used to tell AXIOM to turn IBM Script formula-style output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Syntax:)set output script <arg>
 where arg can be one of
 on turn IBM Script formula printing on
 off turn IBM Script formula printing off
 (default state)
 console send IBM Script formula output to screen
 (default state)
 fp<.fe> send IBM Script formula output to file with file
 prefix fp and file extension .fe. If not given,
 .fe defaults to .sform.

If you wish to send the output to a file, you must issue this command twice: once with on and once with the file name. For example, to send IBM Script formula output to the file polymer.sform, issue the two commands

```
)set output script on
)set output script polymer
```

The output is placed in the directory from which you invoked AXIOM or the one you set with the)cd system command. The current setting is: Off:CONSOLE

44.32.37 defvar \$formulaFormat

— initvars —

```
(defvar |$formulaFormat| nil "display output in SCRIPT format")
```

44.32.38 defvar \$formulaOutputFile

— initvars —

```
(defvar |$formulaOutputFile| "CONSOLE"
  "where script output goes (enter {\em console} or a a pathname)")
```

— **outputscript** —

```
(|script|
  "display output in SCRIPT formula format"
  |interpreter|
  FUNCTION
  |setOutputFormula|
  (("display output in SCRIPT format"
    LITERALS
    |$formulaFormat|
    (|off| |on|)
    |off|)
  (|break| |$formulaFormat|)
  ("where script output goes (enter {\em console} or a a pathname)"
  FILENAME
  |$formulaOutputFile|
  |chkOutputFileName|
  "console"))
NIL)
```

44.32.39 defun setOutputFormula

```
[defiostream p938]
[concat p1003]
[describeSetOutputFormula p769]
[pairp p??]
[qcdr p??]
[qcar p??]
[member p1004]
[upcase p??]
[sayKeyedMsg p331]
[shut p938]
[pathnameType p996]
[pathnameDirectory p998]
[pathnameName p996]
[$filep p??]
[make-outstream p937]
[object2String p??]
```



```
[$formulaOutputStream p??]
[$formulaOutputFile p766]
[$filep p??]
[$formulaFormat p766]
```

— defun setOutputFormula —

```
(defun |setOutputFormula| (arg)
  (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$formulaOutputStream| |$formulaOutputFile| $filep
      |$formulaFormat|)))
    (cond
      ((eq arg '|%initialize%|)
        (setq |$formulaOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
        (setq |$formulaOutputFile| "CONSOLE")
        (setq |$formulaFormat| nil))
      ((eq arg '|%display%|)
        (if |$formulaFormat|
          (setq label "On:")
          (setq label "Off:"))
        (concat label |$formulaOutputFile|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
        (|describeSetOutputFormula|))
      (t
        (cond
          ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t)
            (|member| fn '(y n ye yes no o on of off console
              |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
            '|ok|)
          (t (setq arg (list fn '|sform|))))
        (cond
          ((and (pairp arg)
            (eq (qcdr arg) nil)
            (progn (setq fn (qcar arg)) t))
            (cond
              ((|member| (upcase fn) '(y n ye o of))
                (|sayKeyedMsg| 's2iv0002 '(|script| |script|)))
              ((|member| (upcase fn) '(no off)) (setq |$formulaFormat| nil))
              ((|member| (upcase fn) '(yes on)) (setq |$formulaFormat| t))
              ((eq (upcase fn) 'console)
                (SHUT |$formulaOutputStream|)
                (setq |$formulaOutputStream|
                  (defiostream '((mode . output) (device . console)) 255 0))
                (setq |$formulaOutputFile| "CONSOLE"))))
            (or
              (and (pairp arg)
```

```

      (progn (setq fn (qcar arg))
              (setq tmp1 (qcdr arg))
              (and (pairp tmp1)
                    (eq (qcdr tmp1) nil)
                    (progn (setq ft (qcar tmp1)) t))))
    (and (pairp arg)
          (progn (setq fn (qcar arg))
                  (setq tmp1 (qcdr arg))
                  (and (pairp tmp1)
                        (progn (setq ft (qcar tmp1))
                                (setq tmp2 (qcdr tmp1))
                                (and (pairp tmp2)
                                      (eq (qcdr tmp2) nil)
                                      (progn
                                         (setq fm (qcar tmp2)) t)))))))
    (if (setq ptype (|pathnameType| fn))
        (setq fn (concat (|pathnameDirectory| fn) (|pathnameName| fn))
            (setq ft ptype))
        (unless fm (setq fm 'a))
        (setq filename ($filep fn ft fm))
        (cond
         ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
         ((setq teststream (make-outstream filename 255 0))
          (shut |$formulaOutputStream|)
          (setq |$formulaOutputStream| teststream)
          (setq |$formulaOutputFile| (|object2String| filename))
          (|sayKeyedMsg| 's2iv0004
                        (list "IBM Script formula" |$formulaOutputFile| )))
         (t
          (|sayKeyedMsg| 's2iv0003 (list fn ft fm))))))
  (t
   (|sayKeyedMsg| 's2iv0005 nil)
   (|describeSetOutputFormula|))))))

```

44.32.40 defun describeSetOutputFormula

```

[|sayBrightly| p??]
[|setOutputFormula| p767]

```

— defun describeSetOutputFormula —

```

(defun |describeSetOutputFormula| ()
  (|sayBrightly| (list
    '|%b| "set output script"
    '|%d| "is used to tell AXIOM to turn IBM Script formula-style"

```

```

'|%l|
"output printing on and off, and where to place the output. By default, the"
'|%l| "destination for the output is the screen but printing is turned off."
'|%l|
'|%l| "Syntax:  )set output script <arg>"
'|%l| "      where arg can be one of"
'|%l| "  on          turn IBM Script formula printing on"
'|%l| "  off         turn IBM Script formula printing off (default state)"
'|%l| "  console     send IBM Script formula output to screen (default state)"
'|%l|
"  fp<.fe>      send IBM Script formula output to file with file prefix fp"
'|%l|
"              and file extension .fe. If not given, .fe defaults to .sform."
'|%l|
'|%l| "If you wish to send the output to a file, you must issue this command"
'|%l| "twice: once with"
'|%b| "on"
'|%d| "and once with the file name. For example, to send"
'|%l| "IBM Script formula output to the file"
'|%b| "polymer.sform,"
'|%d| "issue the two commands"
'|%l|
'|%l| "  )set output script on"
'|%l| "  )set output script polymer"
'|%l|
'|%l| "The output is placed in the directory from which you invoked AXIOM or"
'|%l| "the one you set with the )cd system command."
'|%l| "The current setting is: "
'|%b| (|setOutputFormula| '|%display%|)
'|%d|)))

```

44.32.41 scripts

----- The scripts Option -----

Description: show subscripts,... linearly

The scripts option may be followed by any one of the following:

```

yes
no

```

The current setting is indicated.

44.32.42 defvar \$linearFormatScripts

— initvars —

```
(defvar |$linearFormatScripts| nil "show subscripts,... linearly")
```

—————

— outputscripts —

```
(|scripts|
 "show subscripts,... linearly"
 |interpreter|
 LITERALS
 |$linearFormatScripts|
 (|on| |off|)
 |off|)
```

—————

44.32.43 showeditor

----- The showeditor Option -----

Description: view output of)show in editor

The showeditor option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.32.44 defvar \$useEditorForShowOutput

— initvars —

```
(defvar |$useEditorForShowOutput| nil "view output of )show in editor")
```

—————

— outputshoweditor —

```
(|showeditor|
"view output of )show in editor"
|interpreter|
LITERALS
|$useEditorForShowOutput|
(|on| |off|)
|off|)
```

44.32.45 tex

----- The tex Option -----

Description: create output in TeX style

)set output tex is used to tell AXIOM to turn TeX-style output printing on and off, and where to place the output. By default, the destination for the output is the screen but printing is turned off.

Syntax:)set output tex <arg>
 where arg can be one of

on	turn TeX printing on
off	turn TeX printing off (default state)
console	send TeX output to screen (default state)
fp<.fe>	send TeX output to file with file prefix fp and file extension .fe. If not given, .fe defaults to .stex.

If you wish to send the output to a file, you must issue this command twice: once with on and once with the file name. For example, to send TeX output to the file polymer.stex, issue the two commands

```
)set output tex on
)set output tex polymer
```

The output is placed in the directory from which you invoked AXIOM or the one you set with the)cd system command. The current setting is: Off:CONSOLE

44.32.46 defvar \$texFormat

— initvars —

```
(defvar |$texFormat| nil "create output in TeX format")
```

44.32.47 defvar \$texOutputFile

— initvars —

```
(defvar |$texOutputFile| "CONSOLE"
  "where TeX output goes (enter {\em console} or a pathname)")
```

— outputtex —

```
(|tex|
  "create output in TeX style"
  |interpreter|
  FUNCTION
  |setOutputTex|
  (("create output in TeX format"
    LITERALS
    |$texFormat|
    (|off| |on|)
    |off|)
   (|break| |$texFormat|)
   ("where TeX output goes (enter {\em console} or a pathname)"
    FILENAME
    |$texOutputFile|
    |chkOutputFileName|
    "console"))
  NIL)
```

44.32.48 defun setOutputTex

```
[defiostream p938]
[concat p1003]
```

```

[describeSetOutputTex p776]
[pairp p??]
[qcdr p??]
[qcar p??]
[member p1004]
[upcase p??]
[sayKeyedMsg p331]
[shut p938]
[pathnameType p996]
[pathnameDirectory p998]
[pathnameName p996]
[$filep p??]
[make-outstream p937]
[object2String p??]
[$texOutputStream p??]
[$texOutputFile p773]
[$texFormat p773]
[$filep p??]

```

— defun setOutputTex —

```

(defun |setOutputTex| (arg)
  (let (label tmp1 tmp2 ptype fn ft fm filename teststream)
    (declare (special |$texOutputStream| |$texOutputFile| |$texFormat| $filep))
    (cond
      ((eq arg '|%initialize%|)
       (setq |$texOutputStream|
             (defiostream '((mode . output) (device . console)) 255 0))
       (setq |$texOutputFile| "CONSOLE")
       (setq |$texFormat| nil))
      ((eq arg '|%display%|)
       (if |$texFormat|
          (setq label "On:")
          (setq label "Off:"))
       (concat label |$texOutputFile|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeSetOutputTex|))
      (t
       (cond
         ((and (pairp arg)
              (eq (qcdr arg) nil)
              (progn (setq fn (qcar arg)) t)
              (|member| fn '(y n ye yes no o on of off console
                           |y| |n| |ye| |yes| |no| |o| |on| |of| |off| |console|)))
          '|ok|)
         (t (setq arg (list fn '|stex| nil))))
       (cond
         ((and (pairp arg)

```

```

      (eq (qcdr arg) nil)
      (progn (setq fn (qcar arg)) t))
    (cond
      ((|member| (upcase fn) '(y n ye o of))
        (|sayKeyedMsg| 's2iv0002 '(|TeX| |tex|)))
      ((|member| (upcase fn) '(no off)) (setq |$texFormat| nil))
      ((|member| (upcase fn) '(yes on)) (setq |$texFormat| t))
      ((eq (upcase fn) 'console)
        (shut |$texOutputStream|)
        (setq |$texOutputStream|
          (defiostream '((mode . output) (device . console)) 255 0))
        (setq |$texOutputFile| "CONSOLE"))))
    ((or
      (and (pairp arg)
        (progn (setq fn (qcar arg))
          (setq tmp1 (qcdr arg))
          (and (pairp tmp1)
            (eq (qcdr tmp1) nil)
            (progn (setq ft (qcar tmp1)) t))))
      (and (pairp arg)
        (progn (setq fn (qcar arg))
          (setq tmp1 (qcdr arg))
          (and (pairp tmp1)
            (progn (setq ft (qcar tmp1))
              (setq tmp2 (qcdr tmp1))
              (and (pairp tmp2)
                (eq (qcdr tmp2) nil)
                (progn (setq fm (qcar tmp2)) t)))))))
      (when (setq ptype (|pathnameType| fn))
        (setq fn (concat (|pathnameDirectory| fn) (|pathnameName| fn)))
        (setq ft ptype))
      (unless fm (setq fm 'A))
      (setq filename ($file fn ft fm))
      (cond
        ((null filename) (|sayKeyedMsg| 's2iv0003 (list fn ft fm)))
        ((setq teststream (make-outstream filename 255 0))
          (shut |$texOutputStream|)
          (setq |$texOutputStream| teststream)
          (setq |$texOutputFile| (|object2String| filename))
          (|sayKeyedMsg| 's2iv0004 (list "TeX" |$texOutputFile|)))
        (t (|sayKeyedMsg| 'S2IV0003 (list fn ft fm)))))
    (t
      (|sayKeyedMsg| 's2iv0005 nil)
      (|describeSetOutputTex|))))))

```

44.32.49 defun describeSetOutputTex

```
[sayBrightly p??]
[setOutputTex p773]
```

— defun describeSetOutputTex —

```
(defun |describeSetOutputTex| ()
  (|sayBrightly| (list
    '|%b| ")set output tex"
    '|%d| "is used to tell AXIOM to turn TeX-style output"
    '|%l| "printing on and off, and where to place the output. By default, the"
    '|%l| "destination for the output is the screen but printing is turned off."
    '|%l|
    '|%l| "Syntax: )set output tex <arg>"
    '|%l| " where arg can be one of"
    '|%l| " on turn TeX printing on"
    '|%l| " off turn TeX printing off (default state)"
    '|%l| " console send TeX output to screen (default state)"
    '|%l| " fp<.fe> send TeX output to file with file prefix fp and file"
    '|%l| " extension .fe. If not given, .fe defaults to .stex."
    '|%l|
    '|%l| "If you wish to send the output to a file, you must issue this command"
    '|%l| "twice: once with"
    '|%b| "on"
    '|%d| "and once with the file name. For example, to send"
    '|%l| "TeX output to the file"
    '|%b| "polymer.stex,"
    '|%d| "issue the two commands"
    '|%l|
    '|%l| " )set output tex on"
    '|%l| " )set output tex polymer"
    '|%l|
    '|%l| "The output is placed in the directory from which you invoked AXIOM or"
    '|%l| "the one you set with the )cd system command."
    '|%l| "The current setting is: "
    '|%b| (|setOutputTex| '|%display%|)
    '|%d|)))
```

44.33 quit

----- The quit Option -----

Description: protected or unprotected quit

The quit option may be followed by any one of the following:

```
protected
-> unprotected
```

The current setting is indicated.

44.33.1 defvar \$quitCommandType

— initvars —

```
(defvar |$quitCommandType| ' |protected| "protected or unprotected quit")
```

—————

— quit —

```
(|quit|
 "protected or unprotected quit"
 |interpreter|
 LITERALS
 |$quitCommandType|
 (|protected| |unprotected|)
 |protected|)
```

—————

44.34 streams

Current Values of streams Variables

Variable	Description	Current Value
calculate	specify number of elements to calculate	10
showall	display all stream elements computed	off

— streams —

```
(|streams|
 "set some options for working with streams"
```

```

|interpreter|
TREE
|novar|
(
\getchunk{streamscalculat}
\getchunk{streamsshowall}
))

```

44.34.1 calculate

----- The calculate Option -----

Description: specify number of elements to calculate

)set streams calculate is used to tell AXIOM how many elements of a stream to calculate when a computation uses the stream. The value given after calculate must either be the word all or a positive integer.

The current setting is 10 .

44.34.2 defvar \$streamCount

— initvars —

```

(defvar |$streamCount| 10
  "number of initial stream elements you want calculated")

```

— streamscalculat —

```

(|calculat|
  "specify number of elements to calculate"
  |interpreter|
  FUNCTION
  |setStreamsCalculate|
  (("number of initial stream elements you want calculated"
    INTEGER
    |$streamCount|
    (0 NIL)
    10))

```

NIL)

44.34.3 defun setStreamsCalculate

```
[object2String p??]
[describeSetStreamsCalculate p779]
[nequal p??]
[sayMessage p??]
[bright p??]
[terminateSystemCommand p432]
[$streamCount p778]
```

— defun setStreamsCalculate —

```
(defun |setStreamsCalculate| (arg)
  (let (n)
    (declare (special |$streamCount|))
    (cond
      ((eq arg '|%initialize%|) (setq |$streamCount| 10))
      ((eq arg '|%display%|) (|object2String| |$streamCount|))
      ((or (null arg) (eq arg '|%describe%|) (eq (car arg) '?))
       (|describeSetStreamsCalculate|))
      (t
       (setq n (car arg))
       (cond
         ((and (nequal n '|all|) (or (null (integerp n)) (minusp n)))
          (|sayMessage|
           '("Your value of" ,@(|bright| n) "is invalid because ..."))
          (|describeSetStreamsCalculate|)
          (|terminateSystemCommand|))
         (t (setq |$streamCount| n)))))))
```

44.34.4 defun describeSetStreamsCalculate

```
[sayKeyedMsg p331]
[$streamCount p778]
```

— defun describeSetStreamsCalculate —

```
(defun |describeSetStreamsCalculate| ()
```

```
(declare (special |$streamCount|))
(|sayKeyedMsg| 's2iv0001 (list |$streamCount|)))
```

44.34.5 showall

----- The showall Option -----

Description: display all stream elements computed

The showall option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.34.6 defvar \$streamsShowAll

— initvars —

```
(defvar |$streamsShowAll| nil "display all stream elements computed")
```

— streamsshowall —

```
(|showall|
 "display all stream elements computed"
 |interpreter|
 LITERALS
 |$streamsShowAll|
 (|on| |off|)
 |off|)
```

44.35 system

Current Values of system Variables

Variable	Description	Current Value

functioncode	show gen. LISP for functions when compiled	off
optimization	show optimized LISP code	off
prettyprint	prettyprint BOOT func's as they compile	off

— system —

```
(|system|
  "set some system development variables"
  |development|
  TREE
  |novar|
  (
    \getchunk{systemfunctioncode}
    \getchunk{systemoptimization}
    \getchunk{systemprettyprint}
  ))
```

44.35.1 functioncode

----- The functioncode Option -----

Description: show gen. LISP for functions when compiled

The functioncode option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.35.2 defvar \$reportCompilation

— initvars —

```
(defvar |$reportCompilation| nil "show gen. LISP for functions when compiled")
```

— systemfunctioncode —

```
(|functioncode|
 "show gen. LISP for functions when compiled"
 |development|
 LITERALS
 |$reportCompilation|
 (|on| |off|)
 |off|)
```

44.35.3 optimization

----- The optimization Option -----

Description: show optimized LISP code

The optimization option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.35.4 defvar \$reportOptimization

— initvars —

```
(defvar |$reportOptimization| nil "show optimized LISP code")
```

— systemoptimization —

```
(|optimization|
 "show optimized LISP code"
 |development|
 LITERALS)
```

```
|$reportOptimization|
(|on| |off|)
|off|)
```

44.35.5 prettyprint

----- The prettyprint Option -----

Description: prettyprint BOOT func's as they compile

The prettyprint option may be followed by any one of the following:

```
on
-> off
```

The current setting is indicated.

44.35.6 defvar \$prettyprint

— initvars —

```
(defvar $prettyprint t "prettyprint BOOT func's as they compile")
```

— systemprettyprint —

```
(|prettyprint|
 "prettyprint BOOT func's as they compile"
 |development|
 LITERALS
 $prettyprint
 (|on| |off|)
 |on|)
```

44.36 userlevel

----- The userlevel Option -----

Description: operation access level of system user

The userlevel option may be followed by any one of the following:

```

    interpreter
    compiler
-> development

```

The current setting is indicated.

44.36.1 defvar \$UserLevel

— initvars —

```
(defvar |$UserLevel| ' |development| "operation access level of system user")
```

—————

— userlevel —

```

(|userlevel|
 "operation access level of system user"
 |interpreter|
 LITERALS
 |$UserLevel|
 (|interpreter| |compiler| |development|)
 |development|)

```

—————

— initvars —

```

(defvar |$setOptions| '(
 \getchunk{breakmode}
 \getchunk{compile}
 \getchunk{debug}
 \getchunk{expose}
 \getchunk{functions}

```

```

\getchunk{fortran}
\getchunk{kernel}
\getchunk{hyperdoc}
\getchunk{help}
\getchunk{history}
\getchunk{messages}
\getchunk{naglink}
\getchunk{output}
\getchunk{quit}
\getchunk{streams}
\getchunk{system}
\getchunk{userlevel}
))

```

44.36.2 defvar \$setOptionNames

— initvars —

```
(defvar |$setOptionNames| (mapcar #'car |$setOptions|))
```

— postvars —

```
(eval-when (eval load)
  (|initializeSetVariables| |$setOptions|))
```

44.37 Set code

44.37.1 defun set

```

[set1 p786]
|$setOptions p??]

```

— defun set —

```
(defun |set| (1)
```

```
(declare (special |$setOptions|))
(|set1| 1 |$setOptions|))
```

44.37.2 defun set1

This function will be called with the top level arguments to)set. For instance, given the command

```
)set break break
```

this function gets

```
(set1 (|break| |break|) ....)
```

and given the command

```
)set mes auto off
```

this function gets

```
(set1 (|mes| |auto| |off|) ....)
```

which, because “message” is a TREE, generates the recursive call:

```
(set1 (|auto| |off|) <the message subtree>)
```

The “autoload” subtree is a FUNCTION (printLoadMessages), which gets called with %describe% [displaySetVariableSettings p631]

```
[seq p??]
[exit p??]
[selectOption p459]
[downcase p??]
[lassoc p??]
[satisfiesUserLevel p431]
[sayKeyedMsg p331]
[poundsign p??]
[displaySetOptionInformation p629]
[kdr p??]
[sayMSG p333]
[sayMessage p??]
[bright p??]
[object2String p??]
[translateYesNo2TrueFalse p632]
```

```
[use-fast-links p??]
[literals p??]
[tree p??]
[set1 p786]
[$setOptionNames p785]
[$UserLevel p784]
[$displaySetValue p725]
```

— defun set1 —

```
(defun |set1| (l settree)
  (let (|$setOptionNames| arg setdata st setfunarg num upperlimit arg2)
    (declare (special |$setOptionNames| |$UserLevel| |$displaySetValue|))
    (cond
      ((null l) (|displaySetVariableSettings| settree '||))
      (t
       (setq |$setOptionNames|
              (do ((t1 settree (cdr t1)) t0 (x nil))
                  ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
              (seq
               (exit
                (setq t0 (cons (elt x 0) t0))))))
       (setq arg
              (|selectOption| (downcase (car l)) |$setOptionNames| '|optionError|))
       (setq setdata (cons arg (lassoc arg settree)))
       (cond
          ((null (|satisfiesUserLevel| (third setdata)))
           (|sayKeyedMsg| 's2iz0007 (list |$UserLevel| "set option" nil)))
          ((eq 1 (|#| l)) (|displaySetOptionInformation| arg setdata))
          (t
           (setq st (fourth setdata))
           (case (fourth setdata)
             (function
              (setq setfunarg
                     (if (eq (elt l 1) 'default)
                         '|%initialize%|
                         (kdr l)))
              (if (functionp (fifth setdata))
                  (funcall (fifth setdata) setfunarg)
                  (|sayMSG| (concatenate 'string "    Function not implemented. "
                                           (string (fifth setdata)))))
              (when |$displaySetValue|
                (|displaySetOptionInformation| arg setdata))
              NIL)
             (string
              (setq arg2 (elt l 1))
              (cond
                 ((eq arg2 'default) (set (fifth setdata) (seventh setdata)))
                 (arg2 (set (fifth setdata) arg2)))
```

```

(t nil))
(when (or |$displaySetValue| (null arg2))
  (|displaySetOptionInformation| arg setdata))
NIL)
(integer
(setq arg2
(progn
(setq num (elt 1 1))
(cond
((and (integerp num)
      (>= num (elt (sixth setdata) 0))
      (or (null (setq upperlimit (elt (sixth setdata) 1)))
          (<= num upperlimit)))
  num)
(t
(|selectOption|
 (elt 1 1)
 (cons '|default| (sixth setdata)) nil))))))
(cond
((eq arg2 'default) (set (fifth setdata) (seventh setdata)))
(arg2 (set (fifth setdata) arg2))
(t nil))
(cond
((or |$displaySetValue| (null arg2))
  (|displaySetOptionInformation| arg setdata)))
(cond
((null arg2)
  (|sayMessage|
   (" Your value" ,@(|bright| (|object2String| (elt 1 1)))
    "is not among the valid choices.")))
(t nil)))
(literals
(cond
((setq arg2
  (|selectOption| (elt 1 1)
  (cons '|default| (sixth setdata)) nil))
  (cond
    ((eq arg2 'default)
     (set (fifth setdata)
      (|translateYesNo2TrueFalse| (seventh setdata))))
    (t
     (cond ((eq arg2 '|nobreak|) (use-fast-links t)))
     (cond
      ((eq arg2 '|fastlinks|)
       (use-fast-links nil)
       (setq arg2 '|break|)))
      (set (fifth setdata) (|translateYesNo2TrueFalse| arg2))))))
(when (or |$displaySetValue| (null arg2))
  (|displaySetOptionInformation| arg setdata))
(cond

```

```

((null arg2)
 (|sayMessage|
  (cons " Your value"
    (append (|bright| (|object2String| (elt 1 1)))
      (cons "is not among the valid choices." nil)))))
 (t nil)))
(tree (|set1| (kdr 1) (sixth setdata)) nil)
(t
 (|sayMessage|
  ("Cannot handle set tree node type" ,@( |bright| st) |yet|))
 nil))))))

```

Chapter 45

)show help page Command

45.1 show help page man page

— show.help —

```
=====
A.22.  )show
=====
```

User Level Required: interpreter

Command Syntax:

-)show nameOrAbbrev
-)show nameOrAbbrev)operations
-)show nameOrAbbrev)attributes

Command Description:

This command displays information about AXIOM domain, package and category constructors. If no options are given, the)operations option is assumed. For example,

```
)show POLY
)show POLY )operations
)show Polynomial
)show Polynomial )operations
```

each display basic information about the Polynomial domain constructor and then provide a listing of operations. Since Polynomial requires a Ring (for example, Integer) as argument, the above commands all refer to a unspecified ring R. In the list of operations, \$ means Polynomial(R).

The basic information displayed includes the signature of the constructor (the name and arguments), the constructor abbreviation, the exposure status of the constructor, and the name of the library source file for the constructor.

If operation information about a specific domain is wanted, the full or abbreviated domain name may be used. For example,

```
)show POLY INT
)show POLY INT )operations
)show Polynomial Integer
)show Polynomial Integer )operations
```

are among the combinations that will display the operations exported by the domain Polynomial(Integer) (as opposed to the general domain constructor Polynomial). Attributes may be listed by using the)attributes option.

Also See:

- o)display
- o)set
- o)what

1

45.1.1 defun The)show command

[showSpad2Cmd p792]

— **defun show** —

```
(defun |show| (arg) (|showSpad2Cmd| arg))
```

45.1.2 defun The internal)show command

[member p1004]
 [helpSpad2Cmd p550]
 [sayKeyedMsg p331]
 [qcar p??]
 [reportOperations p793]
 [\$showOptions p??]

¹ “display” (29.2.1 p 513) “set” (44.37.1 p 785) “what” (52.1.2 p 903)

```
[$e p??]
[$env p??]
[$InteractiveFrame p??]
[$options p??]
```

— **defun showSpad2Cmd** —

```
(defun |showSpad2Cmd| (arg)
  (let (|$showOptions| |$e| |$env| constr)
    (declare (special |$showOptions| |$e| |$env| |$InteractiveFrame| |$options|))
    (if (equal arg (list nil))
      (|helpSpad2Cmd| '(|show|))
      (progn
        (setq |$showOptions| '(|attributes| |operations|))
        (unless |$options| (setq |$options| '(|operations|)))
        (setq |$e| |$InteractiveFrame|)
        (setq |$env| |$InteractiveFrame|)
        (cond
          ((and (pairp arg) (eq (qcdr arg) nil) (progn (setq constr (qcar arg)) t))
            (cond
              ((|member| constr '(|Union| |Record| |Mapping|))
                (cond
                  ((eq constr '|Record|)
                    (|sayKeyedMsg| 'S2IZ0044R
                      (list constr ")show Record(a: Integer, b: String)" )))
                  ((eq constr '|Mapping|) (|sayKeyedMsg| 'S2IZ0044M nil))
                  (t
                    (|sayKeyedMsg| 'S2IZ0045T
                      (list constr ")show Union(a: Integer, b: String)" )))
                    (|sayKeyedMsg| 'S2IZ0045U
                      (list constr ")show Union(Integer, String)" )))))
              ((and (pairp constr) (eq (qcar constr) '|Mapping|))
                (|sayKeyedMsg| 'S2IZ0044M nil))
              (t (|reportOperations| constr constr))))
          (t (|reportOperations| arg arg))))))
```

— — —

45.1.3 defun reportOperations

```
[sayBrightly p??]
[bright p??]
[sayKeyedMsg p331]
[qcar p??]
[isNameOfType p??]
[isDomainValuedVariable p??]
[reportOpsFromUnitDirectly0 p799]
```

```

[opOf p??]
[unabbrev p??]
[reportOpsFromLisplib0 p795]
[evaluateType p??]
[mkAtree p??]
[removeZeroOneDestructively p??]
[isType p??]
[$env p??]
[$eval p??]
[$genValue p53]
[$quadSymbol p??]
[$doNotAddEmptyModeIfTrue p??]

```

— defun reportOperations —

```

(defun |reportOperations| (oldArg u)
  (let (|$env| |$eval| |$genValue| |$doNotAddEmptyModeIfTrue|
        tmp1 v unitForm tree unitFormp)
    (declare (special |$env| |$eval| |$genValue| |$quadSymbol|
                      |$doNotAddEmptyModeIfTrue|))
    (setq |$env| (list (list nil)))
    (setq |$eval| t)
    (setq |$genValue| t)
    (when u
      (setq |$doNotAddEmptyModeIfTrue| t)
      (cond
        ((equal u |$quadSymbol|)
         (|sayBrightly|
          (cons " mode denotes" (append (|bright| "any") (list '|type|)))))
        ((eq u '|%)
         (|sayKeyedMsg| 'S2IZ0063 nil)
         (|sayKeyedMsg| 'S2IZ0064 nil)))
        ((and (null (and (pairp u) (eq (qcar u) '|Record|)))
              (null (and (pairp u) (eq (qcar u) '|Union|)))
              (null (|isNameOfType| u))
              (null (and (pairp u)
                        (eq (qcar u) '|typeOf|)
                        (progn
                          (setq tmp1 (qcdr u))
                          (and (pairp tmp1) (eq (qcdr tmp1) nil))))))
         (when (atom oldArg) (setq oldArg (list oldArg)))
         (|sayKeyedMsg| 'S2IZ0063 nil)
         (dolist (op oldArg)
          (|sayKeyedMsg| 'S2IZ0062 (list (|opOf| op)))))
        ((setq v (|isDomainValuedVariable| u)) (|reportOpsFromUnitDirectly0| v))
        (t
         (if (atom u)
              (setq unitForm (|opOf| (|unabbrev| u)))
              (setq unitForm (|unabbrev| u))))

```

```
(if (atom unitForm)
  (|reportOpsFromLisplib0| unitForm u)
  (progn
    (setq unitFormp (|evaluateType| unitForm))
    (setq tree (|mkAtree| (|removeZeroOneDestructively| unitForm)))
    (if (setq unitFormp (|isType| tree))
      (|reportOpsFromUnitDirectly0| unitFormp)
      (|sayKeyedMsg| 'S2IZ0041 (list unitForm))))))
```

45.1.4 defun reportOpsFromLisplib0

```
[reportOpsFromLisplib1 p795]
[reportOpsFromLisplib p796]
[$useEditorForShowOutput p771]
```

— defun reportOpsFromLisplib0 —

```
(defun |reportOpsFromLisplib0| (unitForm u)
  (declare (special |$useEditorForShowOutput|))
  (if |$useEditorForShowOutput|
    (|reportOpsFromLisplib1| unitForm u)
    (|reportOpsFromLisplib| unitForm u)))
```

45.1.5 defun reportOpsFromLisplib1

```
[pathname p998]
[erase p??]
[defiostream p938]
[sayShowWarning p802]
[reportOpsFromLisplib p796]
[shut p938]
[editFile p523]
[$sayBrightlyStream p??]
[$erase p??]
```

— defun reportOpsFromLisplib1 —

```
(defun |reportOpsFromLisplib1| (unitForm u)
  (let (|$sayBrightlyStream| showFile)
    (declare (special |$sayBrightlyStream| $erase))
```

```
(setq showFile (|pathname| (list 'show 'listing 'a)))
($erase showFile)
(setq |$sayBrightlyStream|
  (defiostream '((file ,showFile) (mode . output)) 255 0))
(|sayShowWarning|)
(|reportOpsFromLisplib| unitForm u)
(shut |$sayBrightlyStream|)
(|editFile| showFile)))
```

45.1.6 defun reportOpsFromLisplib

```
[constructor? p??]
[sayKeyedMsg p331]
[getConstructorSignature p??]
[kdr p??]
[getdatabase p967]
[eqsubstlist p??]
[nreverse0 p??]
[sayBrightly p??]
[concat p1003]
[bright p??]
[form2StringWithWhere p??]
[isExposedConstructor p??]
[strconc p??]
[namestring p996]
[selectOptionLC p459]
[dc1 p??]
[centerAndHighlight p??]
[specialChar p936]
[remdup p??]
[msort p??]
[form2String p??]
[say2PerLine p??]
[formatAttribute p??]
[displayOperationsFromLisplib p798]
[$linelength p751]
[$showOptions p??]
[$options p??]
[$FormalMapVariableList p??]
```

— defun reportOpsFromLisplib —

```
(defun |reportOpsFromLisplib| (op u)
```

```

(let (fn s typ nArgs argList functorForm argml tmp1 functorFormWithDecl
      verb sourceFile opt attList)
  (declare (special $linelength |$showOptions| |$options|
                    |$FormalMapVariableList|))
  (if (null (setq fn (|constructor?| op)))
      (|sayKeyedMsg| 'S2IZ0054 (list u))
      (progn
        (setq argml (when (setq s (|getConstructorSignature| op)) (kdr s)))
        (setq typ (getdatabase op 'constructorkind))
        (setq nArgs (|#| argml))
        (setq argList (kdr (getdatabase op 'constructorform)))
        (setq functorForm (cons op argList))
        (setq argml (eqsubstlist argList |$FormalMapVariableList| argml))
        (mapcar #'(lambda (a m) (push (list '(:| a m) tmp1)) argList argml))
        (setq functorFormWithDecl (cons op (nreverse0 tmp1)))
        (|sayBrightly|
         (|concat| (|bright| (|form2StringWithWhere| functorFormWithDecl))
                   " is a" (|bright| typ) "constructor"))
        (|sayBrightly|
         (cons " Abbreviation for"
               (append (|bright| op) (cons "is" (|bright| fn))))))
      (if (|isExposedConstructor| op)
          (setq verb "is")
          (setq verb "is not"))
      (|sayBrightly|
       (cons " This constructor"
             (append (|bright| verb) (list "exposed in this frame."))))
      (setq sourceFile (getdatabase op 'sourcefile))
      (|sayBrightly|
       (cons " Issue"
             (append (|bright| (strconc ")edit " (|namestring| sourceFile)))
                     (cons "to see algebra source code for"
                           (append (|bright| fn) (list '|%l|))))))
      (dolist (item |$options|)
        (setq opt (|selectOptionLC| (car item) |$showOptions| '|optionError|))
        (cond
          ((eq opt '|layout|) (|dc1| fn))
          ((eq opt '|views|)
           (|sayBrightly|
            (cons "To get" (append (|bright| "views")
                                   (list "you must give parameters of constructor")))))
          ((eq opt '|attributes|)
           (|centerAndHighlight| "Attributes" $linelength (|specialChar| '|hbar|))
           (|sayBrightly| ""))
          (setq attList
            (remdup
             (msort
              (mapcar #'(lambda (x) (caar x))
                      (reverse (getdatabase op 'attributes))))))
          (if (null attList)

```

```
(|sayBrightly|
  (|concat| ' |%b| (|form2String| functorForm)
    '|%d| '|has no attributes.| '|%l|))
  (|say2PerLine| (mapcar #'|formatAttribute| attList))))
((eq opt '|operations|)
  (|displayOperationsFromLisplib| functorForm))))))
```

45.1.7 defun displayOperationsFromLisplib

```
[getdatabase p967]
[centerAndHighlight p??]
[specialChar p936]
[reportOpsFromUnitDirectly p799]
[remdup p??]
[msort p??]
[eqsubstlist p??]
[formatOperationAlistEntry p??]
[say2PerLine p??]
[$FormalMapVariableList p??]
[$linelength p751]
```

— defun displayOperationsFromLisplib —

```
(defun |displayOperationsFromLisplib| (form)
  (let (name argl kind opList opl ops)
    (declare (special |$FormalMapVariableList| $linelength))
    (setq name (car form))
    (setq argl (cdr form))
    (setq kind (getdatabase name 'constructorkind))
    (|centerAndHighlight| "Operations" $linelength (|specialChar| '|hbar|))
    (setq opList (getdatabase name 'operationalist))
    (if (null opList)
      (|reportOpsFromUnitDirectly| form)
      (progn
        (setq opl
          (remdup (msort (eqsubstlist argl |$FormalMapVariableList| opList))))
        (setq ops nil)
        (dolist (x opl)
          (setq ops (append ops (|formatOperationAlistEntry| x))))
        (|say2PerLine| ops))))))
```

45.1.8 defun reportOpsFromUnitDirectly0

[reportOpsFromUnitDirectly1 p801]
 [reportOpsFromUnitDirectly p799]
 [\$useEditorForShowOutput p771]

— defun reportOpsFromUnitDirectly0 —

```
(defun |reportOpsFromUnitDirectly0| (D)
  (declare (special |$useEditorForShowOutput|))
  (if |$useEditorForShowOutput|
    (|reportOpsFromUnitDirectly1| D)
    (|reportOpsFromUnitDirectly| D)))
```

—————

45.1.9 defun reportOpsFromUnitDirectly

[member p1004]
 [qcar p??]
 [evalDomain p??]
 [getdatabase p967]
 [sayBrightly p??]
 [concat p1003]
 [formatOpType p??]
 [isExposedConstructor p??]
 [bright p??]
 [sayBrightly p??]
 [strconc p??]
 [namestring p996]
 [selectOptionLC p459]
 [centerAndHighlight p??]
 [specialChar p936]
 [remdup p??]
 [msort p??]
 [formatAttribute p??]
 [centerAndHighlight p??]
 [getl p??]
 [systemErrorHere p??]
 [nreverse0 p??]
 [getOplistForConstructorForm p??]
 [say2PerLine p??]
 [formatOperation p??]
 [\$commentedOps p??]
 [\$CategoryFrame p??]


```
[$linelength p751]
[$options p??]
[$showOptions p??]
```

— defun reportOpsFromUnitDirectly —

```
(defun |reportOpsFromUnitDirectly| (unitForm)
  (let (|$commentedOps| isRecordOrUnion unit top kind abb sourceFile verb opt
        attList constructorFunction tmp1 funlist a sigList tmp2)
    (declare (special |$commentedOps| |$CategoryFrame| $linelength |$options|
                      |$showOptions|))
    (setq isRecordOrUnion
          (and (pairp unitForm)
               (progn (setq a (qcar unitForm)) t)
               (|member| a '(|Record| |Union|))))
    (setq unit (|evalDomain| unitForm))
    (setq top (car unitForm))
    (setq kind (getdatabase top 'constructorkind))
    (|sayBrightly|
     (|concat| '|%b| (|formatOpType| unitForm) '|%d|
               "is a" '|%b| kind '|%d| "constructor."))
    (unless isRecordOrUnion
      (setq abb (getdatabase top 'abbreviation))
      (setq sourceFile (getdatabase top 'sourcefile))
      (|sayBrightly|
       (cons " Abbreviation for"
             (append (|bright| top) (cons "is" (|bright| abb))))))
      (if (|isExposedConstructor| top)
          (setq verb "is")
          (setq verb "is not"))
      (|sayBrightly|
       (cons " This constructor"
             (append (|bright| verb) (list "exposed in this frame." ))))
      (|sayBrightly|
       (cons " Issue"
             (append (|bright| (strconc ")edit " (|namestring| sourceFile)))
                     (cons "to see algebra source code for"
                           (append (|bright| abb) (list '|%l|))))))
    (dolist (item |$options|)
      (setq opt (|selectOptionLC| (car item) |$showOptions| '|optionError|))
      (cond
        ((eq opt '|attributes|)
         (|centerAndHighlight| "Attributes" $linelength (|specialChar| '|hbar|))
         (if isRecordOrUnion
             (|sayBrightly| " Records and Unions have no attributes.")
             (progn
              (|sayBrightly| "")
              (setq attList
                    (remdup
```

```

      (msort
        (mapcar #'(lambda (unit2) (car unit2)) (reverse (elt unit 2))))))
    (|say2PerLine|
      (mapcar #'|formatAttribute| attList))
    nil)))
  ((eq opt '|operations|)
    (setq |$commentedOps| 0)
;    --new form is (<op> <signature> <slotNumber> <condition> <kind>)
    (|centerAndHighlight| "Operations" $linelength (|specialChar| '|hbar|))
    (|sayBrightly| "")
    (cond
      (isRecordOrUnion
        (setq constructorFunction (get1 top '|makeFunctionList|))
        (unless constructorFunction
          (|systemErrorHere| "reportOpsFromUnitDirectly"))
        (setq tmp1
          (funcall constructorFunction '$ unitForm |$CategoryFrame|))
        (setq funlist (car tmp1))
        (setq sigList
          (remdup
            (msort
              (dolist (fun funlist (nreverse0 tmp2))
                (push '(((, (caar fun) ,(cadar fun)) t (,(caddar fun) 0 1)))
                  tmp2))))))
        (t
          (setq sigList
            (remdup (msort (|getOpListForConstructorForm| unitForm)))))
        (|say2PerLine|
          (mapcar #'(lambda (x) (|formatOperation| x unit)) sigList))
        (unless (= |$commentedOps| 0)
          (|sayBrightly|
            (list "Functions that are not yet implemented are preceded by"
              (|bright| "--"))))
          (|sayBrightly| ""))))
    nil))

```

45.1.10 defun reportOpsFromUnitDirectly1

```

[pathname p998]
[erase p??]
[defiostream p938]
[sayShowWarning p802]
[reportOpsFromUnitDirectly p799]
[shut p938]
[editFile p523]

```

```
[$sayBrightlyStream p??]
[$erase p??]
```

— **defun reportOpsFromUnitDirectly1** —

```
(defun |reportOpsFromUnitDirectly1| (D)
  (let (|$sayBrightlyStream| showFile)
    (declare (special |$sayBrightlyStream| $erase))
    (setq showFile (|pathname| (list 'show 'listing 'a)))
    ($erase showFile)
    (setq |$sayBrightlyStream|
      (defiostream '((file ,showFile) (mode . output)) 255 0))
    (|sayShowWarning|)
    (|reportOpsFromUnitDirectly1| D)
    (shut |$sayBrightlyStream|)
    (|editFile| showFile)))
```

45.1.11 **defun sayShowWarning**

```
[sayBrightly p??]
```

— **defun sayShowWarning** —

```
(defun |sayShowWarning| ()
  (|sayBrightly|
    "Warning: this is a temporary file and will be deleted the next")
  (|sayBrightly|
    "      time you use )show. Rename it and FILE if you wish to")
  (|sayBrightly| "      save the contents.")
  (|sayBrightly| ""))
```

Chapter 46

)spool help page Command

46.1 spool help page man page

— spool.help —

```
=====
A.23. )spool
=====
```

User Level Required: interpreter

Command Syntax:

-)spool [fileName]
-)spool

Command Description:

This command is used to save (spool) all AXIOM input and output into a file, called a spool file. You can only have one spool file active at a time. To start spool, issue this command with a filename. For example,

```
)spool integrate.out
```

To stop spooling, issue)spool with no filename.

If the filename is qualified with a directory, then the output will be placed in that directory. If no directory information is given, the spool file will be placed in the current directory. The current directory is the directory from which you started AXIOM or is the directory you specified using the)cd command.

Also See:

- o)cd

1

¹“cd” (?? p ??)

Chapter 47

)summary help page Command

47.1 summary help page man page

— summary.help —

```
)credits      : list the people who have contributed to Axiom

)help <command> gives more information
)quit        : exit AXIOM

)abbreviation : query, set and remove abbreviations for constructors
)cd           : set working directory
)clear        : remove declarations, definitions or values
)close        : throw away an interpreter client and workspace
)compile      : invoke constructor compiler
)display      : display Library operations and objects in your workspace
)edit         : edit a file
)frame        : manage interpreter workspaces
)history      : manage aspects of interactive session
)library      : introduce new constructors
)lisp         : evaluate a LISP expression
)read         : execute AXIOM commands from a file
)savesystem   : save LISP image to a file
)set          : view and set system variables
)show         : show constructor information
)spool        : log input and output to a file
)synonym      : define an abbreviation for system commands
)system       : issue shell commands
)trace        : trace execution of functions
)undo         : restore workspace to earlier state
)what         : search for various things by name
```

47.1.1 defun summary

```
[obey p??]  
[concat p1003]  
[getenviron p31]
```

— defun summary —

```
(defun |summary| (l)  
  (declare (ignore l))  
  (obey (concat "cat " (getenviron "AXIOM") "/doc/spadhelp/summary.help")))
```

Chapter 48

)synonym help page Command

48.1 synonym help page man page

— synonym.help —

```
=====
A.24. )synonym
=====
```

User Level Required: interpreter

Command Syntax:

-)synonym
-)synonym synonym fullCommand
-)what synonyms

Command Description:

This command is used to create short synonyms for system command expressions. For example, the following synonyms might simplify commands you often use.

)synonym save	history)save
)synonym restore	history)restore
)synonym mail	system mail
)synonym ls	system ls
)synonym fortran	set output fortran

Once defined, synonyms can be used in place of the longer command expressions. Thus

)fortran on

is the same as the longer

```
)set fortran output on
```

To list all defined synonyms, issue either of

```
)synonyms
)what synonyms
```

To list, say, all synonyms that contain the substring ‘‘ap’’, issue

```
)what synonyms ap
```

Also See:

- o)set
- o)what

1

48.1.1 defun The)synonym command

[synonymSpad2Cmd p808]

— defun synonym —

```
(defun |synonym| (&rest ignore)
  (declare (ignore ignore))
  (|synonymSpad2Cmd|))
```

48.1.2 defun The)synonym command implementation

```
[getSystemCommandLine p809]
[printSynonyms p454]
[processSynonymLine p811]
[putalist p??]
[terminateSystemCommand p432]
[$CommandSynonymAlist p458]
```

— defun synonymSpad2Cmd —

¹ “set” (44.37.1 p 785) “what” (52.1.2 p 903)

```
(defun |synonymSpad2Cmd| ()
  (let (line pair)
    (declare (special |$CommandSynonymAlist|))
    (setq line (|getSystemCommandLine|))
    (if (string= line "")
        (|printSynonyms| nil)
        (progn
          (setq pair (|processSynonymLine| line))
          (if |$CommandSynonymAlist|
              (putalist |$CommandSynonymAlist| (car pair) (cdr pair))
              (setq |$CommandSynonymAlist| (cons pair nil))))))
    (|terminateSystemCommand|)))
```

48.1.3 defun Return a sublist of applicable synonyms

The argument is a list of synonyms, and this returns a sublist of applicable synonyms at the current user level. [string2id-n p??]

[selectOptionLC p459]

[commandsForUserLevel p428]

[\$systemCommands p423]

[\$UserLevel p784]

— defun synonymsForUserLevel —

```
(defun |synonymsForUserLevel| (arg)
  (let (cmd nl)
    (declare (special |$systemCommands| |$UserLevel|))
    (if (eq |$UserLevel| '|development|)
        arg
        (dolist (syn (reverse arg))
          (setq cmd (string2id-n (cdr syn) 1))
          (when (|selectOptionLC| cmd (|commandsForUserLevel| |$systemCommands|) nil)
            (push syn nl))))
    nl))
```

48.1.4 defun Get the system command from the input line

[strpos p1002]

[substring p??]

[\$currentLine p??]

— defun `getSystemCommandLine` —

```
(defun |getSystemCommandLine| ()
  (let (p line)
    (declare (special |$currentLine|))
    (setq p (strpos ")" |$currentLine| 0 nil))
    (if p
      (setq line (substring |$currentLine| p nil))
      (setq line |$currentLine|))
    (string-left-trim '(#\space) line)))
```

—————

48.1.5 defun Remove system keyword

[`dropLeadingBlanks p??`]
 [`maxindex p??`]

— defun `processSynonymLine,removeKeyFromLine` —

```
(defun |processSynonymLine,removeKeyFromLine| (line)
  (prog (mx)
    (return
      (seq
        (setq line (|dropLeadingBlanks| line))
        (setq mx (maxindex line))
        (exit
          (do ((i 0 (1+ i)))
              ((> i mx) nil)
            (seq
              (exit
                (if (char= (elt line i) #\space)
                  (exit
                    (return
                     (do ((j (1+ i) (1+ j)))
                         ((> j mx) nil)
                       (seq
                        (exit
                          (if (char\= (elt line j) #\space)
                            (exit
                              (return
                               (substring line j nil))))))))))))))))))
```

—————

48.1.6 defun processSynonymLine

[processSynonymLine,removeKeyFromLine p810]

— defun processSynonymLine —

```
(defun |processSynonymLine| (line)
  (cons
    (string2id-n line 1)
    (|processSynonymLine,removeKeyFromLine| line)))
```

—————

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

Chapter 49

)system help page Command

49.1 system help page man page

— system.help —

```
=====
A.25. )system
=====
```

User Level Required: interpreter

Command Syntax:

-)system cmdExpression

Command Description:

This command may be used to issue commands to the operating system while remaining in AXIOM. The cmdExpression is passed to the operating system for execution.

To get an operating system shell, issue, for example,)system sh. When you enter the key combination, Ctrl-D (pressing and holding the Ctrl key and then pressing the D key) the shell will terminate and you will return to AXIOM. We do not recommend this way of creating a shell because Lisp may field some interrupts instead of the shell. If possible, use a shell running in another window.

If you execute programs that misbehave you may not be able to return to AXIOM. If this happens, you may have no other choice than to restart AXIOM and restore the environment via)history)restore, if possible.

Also See:

- o `)boot`
- o `)fin`
- o `)lisp`
- o `)pquit`
- o `)quit`

1

This command is in the list of `$noParseCommands` 18.1.3 which means that its arguments are passed verbatim. This will eventually result in a call to the function `handleNoParseCommands` 18.2.1

¹ “boot” (5.1.8 p 25) “fin” (31.1.1 p 526) “lisp” (?? p ??) “pquit” (40.2.1 p 612) “quit” (41.2.1 p 616)

Chapter 50

)trace help page Command

50.1 trace help page man page

— trace.help —

```
=====
A.26. )trace
=====
```

User Level Required: interpreter

Command Syntax:

-)trace
-)trace)off

-)trace function [options]
-)trace constructor [options]
-)trace domainOrPackage [options]

where options can be one or more of

-)after S-expression
-)before S-expression
-)break after
-)break before
-)cond S-expression
-)count
-)count n
-)depth n
-)local op1 [... opN]
-)nonquietly


```

- )nt
- )off
- )only listOfDataToDisplay
- )ops
- )ops op1 [... opN ]
- )restore
- )stats
- )stats reset
- )timer
- )varbreak
- )varbreak var1 [... varN ]
- )vars
- )vars var1 [... varN ]
- )within executingFunction

```

Command Description:

This command is used to trace the execution of functions that make up the AXIOM system, functions defined by users, and functions from the system library. Almost all options are available for each type of function but exceptions will be noted below.

To list all functions, constructors, domains and packages that are traced, simply issue

```
)trace
```

To untrace everything that is traced, issue

```
)trace )off
```

When a function is traced, the default system action is to display the arguments to the function and the return value when the function is exited. Note that if a function is left via an action such as a THROW, no return value will be displayed. Also, optimization of tail recursion may decrease the number of times a function is actually invoked and so may cause less trace information to be displayed. Other information can be displayed or collected when a function is traced and this is controlled by the various options. Most options will be of interest only to AXIOM system developers. If a domain or package is traced, the default action is to trace all functions exported.

Individual interpreter, lisp or boot functions can be traced by listing their names after)trace. Any options that are present must follow the functions to be traced.

```
)trace f
```

traces the function f. To untrace f, issue

```
)trace f )off
```

Note that if a function name contains a special character, it will be necessary to escape the character with an underscore

```
)trace _/D_,1
```

To trace all domains or packages that are or will be created from a particular constructor, give the constructor name or abbreviation after)trace.

```
)trace MATRIX
)trace List Integer
```

The first command traces all domains currently instantiated with Matrix. If additional domains are instantiated with this constructor (for example, if you have used Matrix(Integer) and Matrix(Float)), they will be automatically traced. The second command traces List(Integer). It is possible to trace individual functions in a domain or package. See the)ops option below.

The following are the general options for the)trace command.

```
)break after
  causes a Lisp break loop to be entered after exiting the traced function.
```

```
)break before
  causes a Lisp break loop to be entered before entering the traced
  function.
```

```
)break
  is the same as )break before.
```

```
)count
  causes the system to keep a count of the number of times the traced
  function is entered. The total can be displayed with )trace )stats and
  cleared with )trace )stats reset.
```

```
)count n
  causes information about the traced function to be displayed for the
  first n executions. After the nth execution, the function is untraced.
```

```
)depth n
  causes trace information to be shown for only n levels of recursion of
  the traced function. The command
```

```
)trace fib )depth 10
```

will cause the display of only 10 levels of trace information for the recursive execution of a user function fib.

`)math`
causes the function arguments and return value to be displayed in the AXIOM monospace two-dimensional math format.

`)nonquietly`
causes the display of additional messages when a function is traced.

`)nt`
This suppresses all normal trace information. This option is useful if the `)count` or `)timer` options are used and you are interested in the statistics but not the function calling information.

`)off`
causes untracing of all or specific functions. Without an argument, all functions, constructors, domains and packages are untraced. Otherwise, the given functions and other objects are untraced. To immediately retrace the untraced functions, issue `)trace)restore`.

`)only listOfDataToDisplay`
causes only specific trace information to be shown. The items are listed by using the following abbreviations:

<code>a</code>	display all arguments
<code>v</code>	display return value
<code>1</code>	display first argument
<code>2</code>	display second argument
<code>15</code>	display the 15th argument, and so on

`)restore`
causes the last untraced functions to be retraced. If additional options are present, they are added to those previously in effect.

`)stats`
causes the display of statistics collected by the use of the `)count` and `)timer` options.

`)stats reset`
resets to 0 the statistics collected by the use of the `)count` and `)timer` options.

`)timer`
causes the system to keep a count of execution times for the traced function. The total can be displayed with `)trace)stats` and cleared with `)trace)stats reset`.

`)varbreak var1 [... varN]`
causes a Lisp break loop to be entered after the assignment to any of the listed variables in the traced function.

`)vars`

causes the display of the value of any variable after it is assigned in the traced function. Note that library code must have been compiled (see description of command `)compile`) using the `)vartrace` option in order to support this option.

```
)vars var1 [... varN]
```

causes the display of the value of any of the specified variables after they are assigned in the traced function. Note that library code must have been compiled (see description of command `)compile`) using the `)vartrace` option in order to support this option.

```
)within executingFunction
```

causes the display of trace information only if the traced function is called when the given `executingFunction` is running.

The following are the options for tracing constructors, domains and packages.

```
)local [op1 [... opN] ]
```

causes local functions of the constructor to be traced. Note that to untrace an individual local function, you must use the fully qualified internal name, using the escape character `_` before the semicolon.

```
)trace FRAC )local
)trace FRAC_;cancelGcd )off
```

```
)ops op1 [... opN]
```

By default, all operations from a domain or package are traced when the domain or package is traced. This option allows you to specify that only particular operations should be traced. The command

```
)trace Integer )ops min max _+ _-
```

traces four operations from the domain `Integer`. Since `+` and `-` are special characters, it is necessary to escape them with an underscore.

Also See:

- o `)boot`
- o `)lisp`
- o `)ltrace`

1

50.1.1 The trace global variables

This decides when to give trace and untrace messages.

¹ “boot” (5.1.8 p 25) “lisp” (?? p ??) “ltrace” (39.1.1 p 610)

50.1.2 defvar \$traceNoisely

— initvars —

```
(defvar |$traceNoisely| nil)
```

—————

50.1.3 defvar \$reportSpadTrace

This reports the traced functions

— initvars —

```
(defvar |$reportSpadTrace| nil)
```

—————

50.1.4 defvar \$optionAlist

— initvars —

```
(defvar |$optionAlist| nil)
```

—————

50.1.5 defvar \$tracedMapSignatures

— initvars —

```
(defvar |$tracedMapSignatures| nil)
```

—————

50.1.6 defvar \$traceOptionList

— initvars —

```
(defvar |$traceOptionList|
  '(|after| |before| |break| |cond| |count| |depth| |local| |mathprint|
    |nonquietly| |nt| |of| |only| |ops| |restore| |timer| |varbreak|
    |vars| |within|))
```

50.1.7 defun trace

[traceSpad2Cmd p821]

— defun trace —

```
(defun |trace| (l)
  (|traceSpad2Cmd| l))
```

50.1.8 defun traceSpad2Cmd

```
[pairp p??]
[qcar p??]
[qcdr p??]
[getMapSubNames p843]
[trace1 p822]
[augmentTraceNames p846]
[traceReply p873]
[$mapSubNameAlist p??]
```

— defun traceSpad2Cmd —

```
(defun |traceSpad2Cmd| (l)
  (let (tmp1 l1)
    (declare (special |$mapSubNameAlist|))
    (cond
      ((and (pairp l)
        (eq (qcar l) '|Tuple|)
        (progn
          (setq tmp1 (qcdr l))
          (and (pairp tmp1)
            (eq (qcdr tmp1) nil)
            (progn
              (setq l1 (qcar tmp1))
              t))))))
```

```

    (setq l l1)))
  (setq |$mapSubNameAlist| (|getMapSubNames| 1))
  (|trace1| (|augmentTraceNames| 1))
  (|traceReply|)))

```

50.1.9 defun trace1

```

[hasOption p431]
[throwKeyedMsg p??]
[unabbrev p??]
[isFunctor p??]
[getTraceOption p828]
[untraceDomainLocalOps p854]
[qslessp p1008]
[poundsign p??]
[untrace p836]
[centerAndHighlight p??]
[ptimers p833]
[say p??]
[pcounters p834]
[selectOptionLC p459]
[resetSpacers p832]
[resetTimers p832]
[resetCounters p832]
[pairp p??]
[qcar p??]
[qcdr p??]
[vecp p??]
[sayKeyedMsg p331]
[devaluate p??]
[lassoc p??]
[trace1 p822]
[delete p??]
[?t p876]
[seq p??]
[exit p??]
[transTraceItem p837]
[addassoc p??]
[getTraceOptions p826]
[/trace,0 p??]
[saveMapSig p827]
[$traceNoisely p820]
[$options p??]

```

[`$lastUntraced` p??]

[`$optionAlist` p820]

— **defun** `trace1` —

```
(defun |trace1| (arg)
  (prog (|$traceNoisely| constructor ops lops temp1 opt a
        oldl newoptions domain tracelist optionlist domainlist
        oplist y varlist argument)
    (declare (special |$traceNoisely| |$options| |$lastUntraced|
                      |$optionAlist|))
    (return
     (seq
      (progn
       (setq |$traceNoisely| nil)
       (cond
        ((|hasOption| |$options| '|nonquietly|)
         (setq |$traceNoisely| t)))
       (cond
        ((|hasOption| |$options| '|off|)
         (cond
          (or (setq ops (|hasOption| |$options| '|ops|)
                  (setq lops (|hasOption| |$options| '|local|)))
              (cond
               ((null arg) (|throwKeyedMsg| 's2it0019 nil))
               (t
                (setq constructor
                      (|unabbrev|
                       (cond
                        ((atom arg) arg)
                        ((null (cdr arg))
                         (cond
                          ((atom (car arg)) (car arg))
                          (t (car (car arg))))))
                      (t nil))))
              (cond
               ((null (|isFunction| constructor))
                (|throwKeyedMsg| 's2it0020 nil))
               (t
                (cond (ops (setq ops (|getTraceOption| ops)) nil))
                (cond
                 (lops
                  (setq lops (cdr (|getTraceOption| lops))
                        (|untraceDomainLocalOps|)
                        (t nil))))))
              ((and (qslisp 1 (|#| |$options|)
                    (null (|hasOption| |$options| '|nonquietly|))
                    (|throwKeyedMsg| 's2it0021 nil))
               (t (|untrace| arg))))
              ((|hasOption| |$options| '|stats|)
```



```

(cond
  ((qslessp 1 (|#| |$options|))
   (|throwKeyedMsg| 's2it0001 (cons ")trace ... )stats" nil)))
(t
  (setq temp1 (car |$options|))
  (setq opt (cdr temp1))
  (cond
    ((null opt)
     (|centerAndHighlight| "Traced function execution times" 78 '-)
     (|ptimers|)
     (say " "))
     (|centerAndHighlight| "Traced function execution counts" 78 '-)
     (|pcounters|))
    (t
     (|selectOptionLC| (car opt) '(|reset|) '|optionError|)
     (|resetSpacers|)
     (|resetTimers|)
     (|resetCounters|)
     (|throwKeyedMsg| 's2it0002 nil))))))
((setq a (|hasOption| |$options| '|restore|))
 (unless (setq old1 |$lastUntraced|)
  (setq newoptions (|delete| a |$options|))
  (if (null arg)
   (|trace1| old1)
   (progn
    (dolist (x arg)
     (if (and (pairp x)
              (progn
               (setq domain (qcar x))
               (setq oplist (qcdr x))
               t)
          (vecp domain)))
      (|sayKeyedMsg| 's2it0003 (cons (|devaluate| domain) nil))
      (progn
       (setq |$options| (append newoptions (lassoc x |$optionAlist|)))
       (|trace1| (list x))))))))))
((null arg) nil)
((and (pairp arg) (eq (qcdr arg) nil) (eq (qcar arg) '?)) (|?t|))
(t
 (setq tracelist
  (or
   (prog (t1)
    (setq t1 nil)
    (return
     (do ((t2 arg (cdr t2)) (x nil))
        ((or (atom t2)
              (progn (setq x (car t2)) nil))
         (nreverse0 t1)))
    (seq
     (exit

```

```

        (setq t1 (cons (|transTraceItem| x) t1))))))
    (return nil)))
  (do ((t3 tracelist (cdr t3)) (x nil))
      ((or (atom t3) (progn (setq x (car t3)) nil)) nil)
    (seq
     (exit
      (setq |$optionAlist| (addassoc x |$options| |$optionAlist|))))
    (setq optionlist (|getTraceOptions| |$options|))
    (setq argument
     (cond
      ((setq domainlist (lassoc '|of| optionlist))
       (cond
        ((lassoc '|ops| optionlist)
         (|throwKeyedMsg| 's2it0004 nil))
        (t
         (setq oplist
          (cond
           (tracelist (list (cons '|ops| tracelist)))
           (t nil)))
          (setq varlist
           (cond
            ((setq y (lassoc '|vars| optionlist))
             (list (cons '|vars| y)))
            (t nil)))
           (append domainlist (append oplist varlist))))
         (optionlist (append tracelist optionlist))
         (t tracelist)))
      (|/TRACE,0|
       (prog (t4)
        (setq t4 nil)
        (return
         (do ((t5 argument (cdr t5)) (|funName| nil))
             ((or (atom t5)
                  (progn (setq |funName| (car t5)) nil))
              (nreverse0 t4))
          (seq
           (exit
            (setq t4 (cons |funName| t4))))))))
      (|saveMapSig|
       (prog (t6)
        (setq t6 nil)
        (return
         (do ((t7 argument (cdr t7)) (|funName| nil))
             ((or (atom t7)
                  (progn (setq |funName| (car t7)) nil))
              (nreverse0 t6))
          (seq
           (exit
            (setq t6 (cons |funName| t6))))))))))

```

50.1.10 defun getTraceOptions

```
[throwKeyedMsg p??]
[throwListOfKeyedMsgs p??]
[poundsign p??]
[seq p??]
[exit p??]
[getTraceOption p828]
[$traceErrorStack p??]
```

— defun getTraceOptions —

```
(defun |getTraceOptions| (|options|)
  (prog (|$traceErrorStack| optionlist temp1 key |parms|)
    (declare (special |$traceErrorStack|))
    (return
      (seq
        (progn
          (setq |$traceErrorStack| nil)
          (setq optionlist
            (prog (t0)
              (setq t0 nil)
              (return
                (do ((t1 |options| (cdr t1)) (x nil))
                  ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
                (seq
                  (exit
                    (setq t0 (cons (|getTraceOption| x) t0))))))))
          (cond
            (|$traceErrorStack|
              (cond
                ((null (cdr |$traceErrorStack|))
                  (setq temp1 (car |$traceErrorStack|))
                  (setq key (car temp1))
                  (setq |parms| (cadr temp1))
                  (|throwKeyedMsg| key (cons "" |parms|)))
                (t
                  (|throwListOfKeyedMsgs| 's2it0017
                    (cons (|#| |$traceErrorStack|) nil)
                    (nreverse |$traceErrorStack|))))
              (t optionlist))))))
```

50.1.11 defun saveMapSig

```
[rassoc p??]
[addassoc p??]
[getMapSig p827]
[$tracedMapSignatures p820]
[$mapSubNameAlist p??]
```

— **defun saveMapSig** —

```
(defun |saveMapSig| (funnames)
  (let (map)
    (declare (special |$tracedMapSignatures| |$mapSubNameAlist|))
    (dolist (name funnames)
      (when (setq map (|rassoc| name |$mapSubNameAlist|))
        (setq |$tracedMapSignatures|
              (addassoc name (|getMapSig| map name) |$tracedMapSignatures|))))))
```

—————

50.1.12 defun getMapSig

```
[get p??]
[boot-equal p??]
[$InteractiveFrame p??]
```

— **defun getMapSig** —

```
(defun |getMapSig| (mapname subname)
  (let (lmms sig)
    (declare (special |$InteractiveFrame|))
    (when (setq lmms (|get| mapname '|localModemap| |$InteractiveFrame|))
      (do ((t0 lmms (cdr t0)) (|mm| nil) (t1 nil sig))
          ((or (atom t0) (progn (setq |mm| (car t0)) nil) t1) nil)
        (when (boot-equal (cadr |mm|) subname) (setq sig (cdar |mm|))))
      sig)))
```

—————

50.1.13 defun getTraceOption,hn

```
[seq p??]
[exit p??]
[isDomainOrPackage p849]
```

[stackTraceOptionError p835]
 [domainToGenvar p835]

— **defun getTraceOption,hn** —

```
(defun |getTraceOption,hn| (x)
  (prog (g)
    (return
      (seq
        (if (and (atom x) (null (upper-case-p (elt (princ-to-string x) 0))))
          (exit
            (seq
              (if (|isDomainOrPackage| (eval x)) (exit x))
              (exit
                (|stackTraceOptionError|
                  (cons 's2it0013 (cons (cons x nil) nil)))))))
          (if (setq g (|domainToGenvar| x)) (exit g))
          (exit
            (|stackTraceOptionError| (cons 's2it0013 (cons (cons x nil) nil)))))))
```

—————

50.1.14 **defun getTraceOption**

[seq p??]
 [exit p??]
 [selectOptionLC p459]
 [identp p1003]
 [stackTraceOptionError p835]
 [concat p1003]
 [object2String p??]
 [transOnlyOption p834]
 [pairp p??]
 [qcdr p??]
 [qcar p??]
 [getTraceOption,hn p827]
 [isListOfIdentifiersOrStrings p843]
 [isListOfIdentifiers p842]
 [throwKeyedMsg p??]
 [\$traceOptionList p820]

— **defun getTraceOption** —

```
(defun |getTraceOption| (arg)
  (prog (l |opts| key a |n|)
    (declare (special |$traceOptionList|))
```

```

(return
  (seq
    (progn
      (setq key (car arg))
      (setq l (cdr arg))
      (setq key
        (|selectOptionLC| key |$traceOptionList| '|traceOptionError|))
      (setq arg (cons key l))
      (cond
        ((member key '(|nonquietly| |timer| |nt|)) arg)
        ((eq key '|break|)
          (cond
            ((null l) (cons '|break| (cons '|before| nil)))
            (t
              (setq |opts|
                (prog (t0)
                  (setq t0 nil)
                  (return
                    (do ((t1 l (cdr t1)) (y nil))
                      ((or (atom t1)
                          (progn (setq y (car t1)) nil))
                     (nreverse0 t0))
                  (seq
                    (exit
                     (setq t0
                       (cons
                        (|selectOptionLC| y '(|before| |after|) nil) t0))))))))
              (cond
                ((prog (t2)
                  (setq t2 t)
                  (return
                    (do ((t3 nil (null t2)) (t4 |opts| (cdr t4)) (y nil))
                      ((or t3 (atom t4) (progn (setq y (car t4)) nil)) t2)
                    (seq
                     (exit
                      (setq t2 (and t2 (identp y))))))))
                  (cons '|break| |opts|)))
                (t
                  (|stackTraceOptionError| (cons 's2it0008 (cons nil nil)))))))
          ((eq key '|restore|)
            (cond
              ((null l) arg)
              (t
                (|stackTraceOptionError|
                  (cons 's2it0009
                    (cons (cons (concat "") (|object2String| key)) nil) nil))))))
          ((eq key '|only|) (cons '|only| (|transOnlyOption| l)))
          ((eq key '|within|)
            (cond
              ((and (pairp l)

```

```

      (eq (qcdr l) nil)
      (progn (setq a (qcar l)) t)
      (identp a))
    arg)
  (t
   (|stackTraceOptionError|
    (cons 's2it0010 (cons (cons ")within" nil) nil))))))
((member key '(|cond| |before| |after|))
 (setq key
  (cond
   ((eq key '|cond|) '|when|)
   (t key)))
 (cond
  ((and (pairp l)
        (eq (qcdr l) nil)
        (progn (setq a (qcar l)) t))
   (cons key l))
  (t
   (|stackTraceOptionError|
    (cons 's2it0011
      (cons
       (cons (concat ")")
        (|object2String| key)) nil) nil))))))
((eq key '|depth|)
 (cond
  ((and (pairp l)
        (eq (qcdr l) nil)
        (progn (setq |n| (qcar l)) t)
        (integerp |n|))
   arg)
  (t
   (|stackTraceOptionError|
    (cons 's2it0012 (cons (cons ")depth" nil) nil))))))
((eq key '|count|)
 (cond
  ((or (null l)
       (and (pairp l)
            (eq (qcdr l) nil)
            (progn (setq |n| (qcar l)) t)
            (integerp |n|))
   arg)
  (t
   (|stackTraceOptionError|
    (cons 's2it0012 (cons (cons ")count" nil) nil))))))
((eq key '|of|)
 (cons '|of|
  (prog (t5)
   (setq t5 nil)
   (return
    (do ((t6 l (cdr t6)) (y nil))

```

```

      ((or (atom t6) (progn (setq y (car t6)) nil)) (nreverse0 t5))
    (seq
      (exit
        (setq t5 (cons (|getTraceOption,hn| y) t5)))))))))
  ((member key '(|local| ops |vars|))
    (cond
      ((or (null l)
        (and (pairp l) (eq (qcdr l) nil) (eq (qcar l) '|all|)))
        (cons key '|all|))
      ((|isListOfIdentifiersOrStrings| l) arg)
      (t
        (|stackTraceOptionError|
          (cons 's2it0015
            (cons
              (cons (concat "") (|object2String| key)) nil) nil))))))
    ((eq key '|varbreak|)
      (cond
        ((or (null l)
          (and (pairp l) (eq (qcdr l) nil) (eq (qcar l) '|all|)))
          (cons '|varbreak| '|all|))
        ((|isListOfIdentifiers| l) arg)
        (t
          (|stackTraceOptionError|
            (cons 's2it0016
              (cons
                (cons (concat "") (|object2String| key)) nil) nil))))))
    ((eq key '|mathprint|)
      (cond
        ((null l) arg)
        (t
          (|stackTraceOptionError|
            (cons 's2it0009
              (cons
                (cons (concat "") (|object2String| key)) nil) nil))))))
    (key (|throwKeyedMsg| 's2it0005 (cons key nil)))))))))

```

50.1.15 defun traceOptionError

[stackTraceOptionError p835]
 [commandAmbiguityError p432]

— defun traceOptionError —

```

(defun |traceOptionError| (opt keys)
  (if (null keys)

```



```
(|stackTraceOptionError| (cons 's2it0007 (cons (cons opt nil) nil)))
(|commandAmbiguityError| '|trace option| opt keys)))
```

50.1.16 defun resetTimers

```
[concat p1003]
[/timerlist p??]
```

— defun resetTimers —

```
(defun |resetTimers| ()
  (declare (special /timerlist))
  (dolist (timer /timerlist)
    (set (intern (concat timer ",TIMER")) 0)))
```

50.1.17 defun resetSpacers

```
[concat p1003]
[/spacelist p??]
```

— defun resetSpacers —

```
(defun |resetSpacers| ()
  (declare (special /spacelist))
  (dolist (spacer /spacelist)
    (set (intern (concat spacer ",SPACE")) 0)))
```

50.1.18 defun resetCounters

```
[concat p1003]
[/countlist p??]
```

— defun resetCounters —

```
(defun |resetCounters| ()
  (declare (special /countlist))
```

```
(dolist (k /countlist)
  (set (intern (concat k ",COUNT")) 0)))
```

50.1.19 defun ptimers

```
[sayBrightly p??]
[bright p??]
[quotient p??]
[concat p1003]
[float p??]
[/timerlist p??]
```

— defun ptimers —

```
(defun |ptimers| ()
  (declare (special /timerlist |$timerTicksPerSecond|))
  (if (null /timerlist)
    (|sayBrightly| " no functions are timed")
    (dolist (timer /timerlist)
      (|sayBrightly|
        ‘( " " ,@( |bright| timer) |:| " "
          ,(quotient (eval (intern (concat timer ",TIMER"))))
            (|float| |$timerTicksPerSecond|)) " sec."))))))
```

50.1.20 defun pspacers

```
[sayBrightly p??]
[bright p??]
[concat p1003]
[/spacelist p??]
```

— defun pspacers —

```
(defun |pspacers| ()
  (declare (special /spacelist))
  (if (null /spacelist)
    (|sayBrightly| " no functions have space monitored")
    (dolist (spacer /spacelist)
      (|sayBrightly|
        ‘( " " ,@( |bright| spacer) |:| |
```

```
,(eval (intern (concat spacer ",SPACE"))) " bytes")))))
```

50.1.21 defun pcounters

```
[sayBrightly p??]
[bright p??]
[concat p1003]
[/countlist p??]
```

— defun pcounters —

```
(defun |pcounters| ()
  (declare (special /countlist))
  (if (null /countlist)
      (|sayBrightly| " no functions are being counted")
      (dolist (k /countlist)
        (|sayBrightly|
         '(" " ,@( |bright| k) |:|| " " ,(eval (intern (concat k ",COUNT")))
         " times")))))
```

50.1.22 defun transOnlyOption

```
[transOnlyOption p834]
[upcase p??]
[stackTraceOptionError p835]
[pairp p??]
[qcar p??]
[qcdr p??]
```

— defun transOnlyOption —

```
(defun |transOnlyOption| (arg)
  (let (y n)
    (when (and (pairp arg) (progn (setq n (qcar arg)) (setq y (qcdr arg)) t))
      (cond
        ((integerp n) (cons n (|transOnlyOption| y)))
        ((member (setq n (upcase n)) '(v a c)) (cons n (|transOnlyOption| y)))
        (t
         (|stackTraceOptionError| (cons 's2it0006 (list (list n))))
         (|transOnlyOption| y)))))
```

50.1.23 defun stackTraceOptionError

[traceErrorStack p??]

— defun stackTraceOptionError —

```
(defun |stackTraceOptionError| (x)
  (declare (special |$traceErrorStack|))
  (push x |$traceErrorStack|)
  nil)
```

50.1.24 defun removeOption

[nequal p??]

— defun removeOption —

```
(defun |removeOption| (op options)
  (let (opt t0)
    (do ((t1 options (cdr t1)) (optentry nil))
      ((or (atom t1)
           (progn (setq optentry (car t1)) nil)
           (progn (progn (setq opt (car optentry)) optentry) nil))
        (nreverse0 t0))
      (when (nequal opt op) (setq t0 (cons optentry t0))))))
```

50.1.25 defun domainToGenvar

```
[unabbrevAndLoad p??]
[getdatabase p967]
[opOf p??]
[genDomainTraceName p836]
[evalDomain p??]
[$doNotAddEmptyModeIfTrue p??]
```

— defun domainToGenvar —

```
(defun |domainToGenvar| (arg)
  (let (|$doNotAddEmptyModeIfTrue| y g)
    (declare (special |$doNotAddEmptyModeIfTrue|))
    (setq |$doNotAddEmptyModeIfTrue| t)
    (when
      (and (setq y (|unabbrevAndLoad| arg))
           (eq (getdatabase (|opOf| y) 'constructorkind) '|domain|))
      (setq g (|genDomainTraceName| y))
      (set g (|evalDomain| y)
            g)))
```

50.1.26 defun genDomainTraceName

```
[lassoc p??]
[genvar p??]
[$domainTraceNameAssoc p??]
```

— defun genDomainTraceName —

```
(defun |genDomainTraceName| (y)
  (let (u g)
    (declare (special |$domainTraceNameAssoc|))
    (if (setq u (lassoc y |$domainTraceNameAssoc|))
        u
        (progn
          (setq g (genvar))
          (setq |$domainTraceNameAssoc| (cons (cons y g) |$domainTraceNameAssoc|)
                g))))
```

50.1.27 defun untrace

```
[copy p??]
[transTraceItem p837]
[/untrace,0 p??]
[lassocSub p845]
[removeTracedMapSigs p838]
[$lastUntraced p??]
[$mapSubNameAlist p??]
[/tracenames p??]
```

— defun untrace —

```

(defun |untrace| (arg)
  (let (untracelist)
    (declare (special |$lastUntraced| /tracenames |$mapSubNameAlist|))
    (if arg
      (setq |$lastUntraced| arg)
      (setq |$lastUntraced| (copy /tracenames)))
    (setq untracelist
      (do ((t1 arg (cdr t1)) (x nil) (t0 nil))
          ((or (atom t1) (progn (setq x (car t1)) nil))
           (nreverse0 t0))
          (push (|transTraceItem| x) t0)))
    (|/UNTRACE,0|
     (do ((t3 untracelist (cdr t3)) (|funName| nil) (t2 nil))
         ((or (atom t3) (progn (setq |funName| (car t3)) nil))
          (nreverse0 t2))
         (push (|lassocSub| |funName| |$mapSubNameAlist|) t2)))
    (|removeTracedMapSigs| untracelist)))

```

50.1.28 defun transTraceItem

```

[get p??]
[member p1004]
[objMode p??]
[objVal p??]
[domainToGenvar p835]
[unabbrev p??]
[constructor? p??]
[pairp p??]
[vecp p??]
[transTraceItem p837]
[devaluate p??]
[throwKeyedMsg p??]
[$doNotAddEmptyModeIfTrue p??]

```

— defun transTraceItem —

```

(defun |transTraceItem| (x)
  (prog (|doNotAddEmptyModeIfTrue| |value| y)
    (declare (special |$doNotAddEmptyModeIfTrue|))
    (return
     (progn
      (setq |$doNotAddEmptyModeIfTrue| t)
      (cond
       ((atom x)

```

```

(cond
  ((and (setq |value| (|get| x '|value| |$InteractiveFrame|))
        (|member| (|objModel| |value|)
                  '((|Mode|) (|Domain|) (|SubDomain| (|Domain|)))))
    (setq x (|objVal| |value|))
    (cond
      ((setq y (|domainToGenvar| x)) y)
      (t x)))
  ((upper-case-p (elt (princ-to-string x) 0))
    (setq y (|unabbrev| x))
    (cond
      ((|constructor?| y) y)
      ((and (pairp y) (|constructor?| (car y))) (car y))
      ((setq y (|domainToGenvar| x)) y)
      (t x)))
  (t x)))
((vecp (car x)) (|transTraceItem| (|devaluate| (car x))))
((setq y (|domainToGenvar| x)) y)
(t (|throwKeyedMsg| 's2it0018 (cons x nil))))))

```

50.1.29 defun removeTracedMapSigs

[*\$tracedMapSignatures* p820]

— defun removeTracedMapSigs —

```

(defun |removeTracedMapSigs| (untraceList)
  (declare (special |$tracedMapSignatures|))
  (dolist (name untraceList)
    (remprop name |$tracedMapSignatures|)))

```

50.1.30 defun coerceTraceArgs2E

[spadsysnamep p??]
 [pname p1001]
 [coerceSpadArgs2E p839]
 [objValUnwrap p??]
 [coerceInteractive p??]
 [objNewWrap p??]
 [\$OutputForm p??]
 [\$mathTraceList p??]

[[\\$tracedMapSignatures p820](#)]

— **defun coerceTraceArgs2E** —

```
(defun |coerceTraceArgs2E| (tracename subname args)
  (declare (ignore tracename))
  (let (name)
    (declare (special |$OutputForm| |$mathTraceList| |$tracedMapSignatures|))
    (cond
      ((member (setq name subname) |$mathTraceList|)
        (if (spadsysnamep (pname name))
          (|coerceSpadArgs2E| (reverse (cdr (reverse args)))))
        (do ((t1 '(|arg1| |arg2| |arg3| |arg4| |arg5| |arg6| |arg7| |arg8|
                    |arg9| |arg10| |arg11| |arg12| |arg13| |arg14| |arg15|
                    |arg16| |arg17| |arg18| |arg19|) (cdr t1))
              (name nil)
              (t2 args (cdr t2))
              (arg nil)
              (t3 (cdr (lassoc subname |$tracedMapSignatures|)) (cdr t3))
              (type nil)
              (t0 nil))
            ((or (atom t1)
                  (progn (setq name (car t1)) nil)
                  (atom t2)
                  (progn (setq arg (car t2)) nil)
                  (atom t3)
                  (progn (setq type (car t3)) nil))
              (nreverse0 t0))
          (setq t0
            (cons
              (list '= name
                (|objValUnwrap|
                  (|coerceInteractive|
                    (|objNewWrap| arg type) |$OutputForm|))) t0))))))
      ((spadsysnamep (pname name)) (reverse (cdr (reverse args))))
      (t args))))
```

50.1.31 defun coerceSpadArgs2E

[[seq p??](#)]
 [[exit p??](#)]
 [[objValUnwrap p??](#)]
 [[coerceInteractive p??](#)]
 [[objNewWrap p??](#)]
 [[\\$streamCount p778](#)]


```
[$OutputForm p??]
[$tracedSpadModemap p??]
```

— **defun coerceSpadArgs2E** —

```
(defun |coerceSpadArgs2E| (args)
  (let ((|$streamCount| 0))
    (declare (special |$streamCount| |$OutputForm| |$tracedSpadModemap|))
    (do ((t1 '(|arg1| |arg2| |arg3| |arg4| |arg5| |arg6| |arg7| |arg8|
              |arg9| |arg10| |arg11| |arg12| |arg13| |arg14| |arg15|
              |arg16| |arg17| |arg18| |arg19|) (cdr t1))
         (name nil)
         (t2 args (cdr t2))
         (arg nil)
         (t3 (cdr |$tracedSpadModemap|) (cdr t3))
         (type nil)
         (t0 nil))
        ((or (atom t1)
              (progn (setq name (car t1)) nil)
              (atom t2)
              (progn (setq arg (car t2)) nil)
              (atom t3)
              (progn (setq type (car t3)) nil))
         (nreverse0 t0))
      (seq
       (exit
        (setq t0
         (cons
          (cons '=
           (cons name
            (cons (|objValUnwrap|
                    (|coerceInteractive|
                     (|objNewWrap| arg type)
                     |$OutputForm|)) nil)))
          t0)))))))
```

50.1.32 defun subTypes

```
[lassoc p??]
[seq p??]
[exit p??]
[subTypes p840]
```

— **defun subTypes** —

```
(defun |subTypes| (|mm| |sublist|)
```

```
(prog (s)
  (return
    (seq
      (cond
        ((atom |mm|)
          (cond ((setq s (lassoc |mm| |sublist|)) s) (t |mm|)))
        (t
          (prog (t0)
            (setq t0 nil)
            (return
              (do ((t1 |mm| (cdr t1)) (|m| nil))
                ((or (atom t1) (progn (setq |m| (car t1)) nil)) (nreverse0 t0))
              (seq
                (exit
                  (setq t0 (cons (|subTypes| |m| |sublist|) t0))))))))))))))
```

50.1.33 defun coerceTraceFunValue2E

```
[spadsysnamep p??]
[pname p1001]
[coerceSpadFunValue2E p842]
[lassoc p??]
[objValUnwrap p??]
[coerceInteractive p??]
[objNewWrap p??]
[$tracedMapSignatures p820]
[$OutputForm p??]
[$mathTraceList p??]
```

— defun coerceTraceFunValue2E —

```
(defun |coerceTraceFunValue2E| (tracename subname |value|)
  (let (name u)
    (declare (special |$tracedMapSignatures| |$OutputForm| |$mathTraceList|))
    (if (member (setq name subname) |$mathTraceList|)
      (cond
        ((spadsysnamep (pname tracename)) (|coerceSpadFunValue2E| |value|))
        ((setq u (lassoc subname |$tracedMapSignatures|))
          (|objValUnwrap|
            (|coerceInteractive| (|objNewWrap| |value| (car u)) |$OutputForm|)))
        (t |value|))
      |value|)))
```

50.1.34 defun coerceSpadFunValue2E

```
[objValUnwrap p??]
[coerceInteractive p??]
[objNewWrap p??]
[$streamCount p778]
[$tracedSpadModemap p??]
[$OutputForm p??]
```

— **defun coerceSpadFunValue2E** —

```
(defun |coerceSpadFunValue2E| (|value|)
  (let (|$streamCount|)
    (declare (special |$streamCount| |$tracedSpadModemap| |$OutputForm|))
    (setq |$streamCount| 0)
    (|objValUnwrap|
     (|coerceInteractive|
      (|objNewWrap| |value| (car |$tracedSpadModemap|))
      |$OutputForm|))))
```

—————

50.1.35 defun isListOfIdentifiers

```
[seq p??]
[exit p??]
[identp p1003]
```

— **defun isListOfIdentifiers** —

```
(defun |isListOfIdentifiers| (arg)
  (prog ()
    (return
     (seq
      (prog (t0)
        (setq t0 t)
        (return
         (do ((t1 nil (null t0)) (t2 arg (cdr t2)) (x nil))
              ((or t1 (atom t2) (progn (setq x (car t2)) nil)) t0)
          (seq
           (exit
            (setq t0 (and t0 (identp x))))))))))))))
```

—————

50.1.36 defun isListOfIdentifiersOrStrings

```
[seq p??]
[exit p??]
[identp p1003]
```

— defun isListOfIdentifiersOrStrings —

```
(defun |isListOfIdentifiersOrStrings| (arg)
  (prog ()
    (return
      (seq
        (prog (t0)
          (setq t0 t)
          (return
            (do ((t1 nil (null t0)) (t2 arg (cdr t2)) (x nil))
              ((or t1 (atom t2) (progn (setq x (car t2)) nil)) t0)
            (seq
              (exit
                (setq t0 (and t0 (or (identp x) (stringp x))))))))))))))
```

50.1.37 defun getMapSubNames

```
[get p??]
[union p??]
[getPreviousMapSubNames p844]
[unionq p??]
[$lastUntraced p??]
[$InteractiveFrame p??]
[/tracenames p??]
```

— defun getMapSubNames —

```
(defun |getMapSubNames| (arg)
  (let (lmm subs)
    (declare (special /tracenames |$lastUntraced| |$InteractiveFrame|))
    (setq subs nil)
    (dolist (mapname arg)
      (when (setq lmm (|get| mapname '|localModemap| |$InteractiveFrame|))
        (setq subs
          (append
            (do ((t2 lmm (cdr t2)) (t1 nil) (lmm| nil))
              ((or (atom t2)
                (progn (setq lmm| (CAR t2)) nil)) (nreverse0 t1))
```

```

      (setq t1 (cons (cons mapname (cadr lmm)) t1)))
    subs))))
(|union| subs
(|getPreviousMapSubNames| (unionq /tracenames |$lastUntraced|))))))

```

50.1.38 defun getPreviousMapSubNames

```

[get p??]
[exit p??]
[seq p??]

```

— defun getPreviousMapSubNames —

```

(defun |getPreviousMapSubNames| (|traceNames|)
  (prog (lmm subs)
    (return
      (seq
        (progn
          (setq subs nil)
          (seq
            (do ((t0 (assocleft (caar |$InteractiveFrame|) (cdr t0))
              (mapname nil))
              ((or (atom t0) (progn (setq mapname (car t0)) nil)) nil)
            (seq
              (exit
                (cond
                  ((setq lmm
                    (|get| mapname '|localModemap| |$InteractiveFrame|))
                  (exit
                    (cond
                      ((member (cadar lmm) |traceNames|)
                     (exit
                       (do ((t1 lmm (cdr t1)) (|lmm| nil))
                         ((or (atom t1) (progn (setq |lmm| (car t1)) nil)) nil)
                       (seq
                        (exit
                          (setq subs
                            (cons (cons mapname (cadr lmm)) subs))))))))))))
                    (exit subs)))))))

```

50.1.39 defun lassocSub

[lassq p??]

— defun lassocSub —

```
(defun |lassocSub| (x subs)
  (let (y)
    (if (setq y (lassq x subs))
        y
        x)))
```

50.1.40 defun rassocSub

[rassoc p??]

— defun rassocSub —

```
(defun |rassocSub| (x subs)
  (let (y)
    (if (setq y (|rassoc| x subs))
        y
        x)))
```

50.1.41 defun isUncompiledMap

[get p??]

[\$InteractiveFrame p??]

— defun isUncompiledMap —

```
(defun |isUncompiledMap| (x)
  (let (y)
    (declare (special |$InteractiveFrame|))
    (when (setq y (|get| x '|value| |$InteractiveFrame|))
      (and
        (eq (caar y) 'map)
        (null (|get| x '|localModemap| |$InteractiveFrame|))))))
```

50.1.42 defun isInterpOnlyMap

```
[get p??]
[$InteractiveFrame p??]
```

— defun isInterpOnlyMap —

```
(defun |isInterpOnlyMap| (map)
  (let (x)
    (declare (special |$InteractiveFrame|))
    (when (setq x (|get| map '|localModemap| |$InteractiveFrame|))
      (eq (caaar x) '|interpOnly|))))
```

—————

50.1.43 defun augmentTraceNames

```
[get p??]
[$InteractiveFrame p??]
```

— defun augmentTraceNames —

```
(defun |augmentTraceNames| (arg)
  (let (mml res)
    (declare (special |$InteractiveFrame|))
    (dolist (tracename arg)
      (if (setq mml (|get| tracename '|localModemap| |$InteractiveFrame|))
        (setq res
          (append
            (prog (t1)
              (setq t1 nil)
              (return
                (do ((t2 mml (cdr t2)) (|mm| nil))
                  ((or (atom t2)
                      (progn (setq |mm| (CAR t2)) nil))
                   (nreverse0 t1))
                (setq t1 (cons (cadr |mm|) t1))))))
            res))
        (setq res (cons tracename res))))
  res))
```

—————

50.1.44 defun isSubForRedundantMapName

```
[rassocSub p845]
[member p1004]
[assocleft p??]
[$mapSubNameAlist p??]
```

— **defun isSubForRedundantMapName** —

```
(defun |isSubForRedundantMapName| (subname)
  (let (mapname tail)
    (declare (special |$mapSubNameAlist|))
    (when (setq mapname (|rassocSub| subname |$mapSubNameAlist|))
      (when (setq tail (|member| (cons mapname subname) |$mapSubNameAlist|))
        (member mapname (cdr (assocleft tail)))))))
```

50.1.45 defun untraceMapSubNames

```
[assocright p??]
[/untrace,2 p??]
[setdifference p??]
[getPreviousMapSubNames p844]
[$mapSubNameAlist p??]
[$lastUntraced p??]
```

— **defun untraceMapSubNames** —

```
(defun |untraceMapSubNames| (|traceNames|)
  (let (|$mapSubNameAlist| subs)
    (declare (special |$mapSubNameAlist| |$lastUntraced|))
    (if
      (null (setq |$mapSubNameAlist| (|getPreviousMapSubNames| |traceNames|)))
      nil
      (dolist (name (setq subs (assocright |$mapSubNameAlist|)))
        (when (member name /tracenames)
          (|/UNTRACE,2| name nil)
          (setq |$lastUntraced| (setdifference |$lastUntraced| subs)))))))
```

50.1.46 defun funfind,LAM

```
[pairp p??]
[qcar p??]
[SEQ p??]
[isFunctor p??]
[exit p??]
```

— defun funfind,LAM —

```
(defun |funfind,LAM| (functor opname)
  (prog (ops tmp1)
    (return
      (seq
        (progn
          (setq ops (|isFunctor| functor))
          (prog (t0)
            (setq t0 nil)
            (return
              (do ((t1 ops (cdr t1)) (u nil))
                ((or (atom t1) (progn (setq u (car t1)) nil)) (nreverse0 t0))
              (seq
                (exit
                  (cond
                    ((and (pairp u)
                        (progn
                          (setq tmp1 (qcar u))
                          (and (pairp tmp1) (equal (qcar tmp1) opname))))
                     (setq t0 (cons u t0))))))))))))))
```

—————

50.1.47 defmacro funfind

— defmacro funfind —

```
(defmacro |funfind| (&whole t0 &rest notused &aux t1)
  (declare (ignore notused))
  (dsetq t1 t0)
  (cons '|funfind,LAM| (wrap (cdr t1) '(quote quote))))
```

—————

50.1.48 defun isDomainOrPackage

[refvecp p??]

[poundsign p??]

[isFunction p??]

[opOf p??]

— defun isDomainOrPackage —

```
(defun |isDomainOrPackage| (dom)
  (and
    (refvecp dom)
    (> (|#| dom) 0)
    (|isFunction| (|opOf| (elt dom 0)))))
```

—————

50.1.49 defun isTraceGensym

[gensymp p??]

— defun isTraceGensym —

```
(defun |isTraceGensym| (x)
  (gensymp x))
```

—————

50.1.50 defun spadTrace,g

— defun spadTrace,g —

```
(defun |spadTrace,g| (x)
  (if (stringp x) (intern x) x))
```

—————

50.1.51 defun spadTrace,isTraceable

[seq p??]

[exit p??]

```
[gensymp p??]
[reportSpadTrace p866]
[bpiname p??]
```

— **defun spadTrace,isTraceable** —

```
(defun |spadTrace,isTraceable| (x |domain|)
  (prog (n |functionSlot|)
    (return
      (seq
        (progn
          (setq n (caddr x))
          x
          (seq
            (if (atom (elt |domain| n)) (exit nil))
            (setq |functionSlot| (car (elt |domain| n)))
            (if (gensymp |functionSlot|)
              (exit (seq (|reportSpadTrace| '|Already Traced| x) (exit nil))))
            (if (null (bpiname |functionSlot|))
              (exit
                (seq
                  (|reportSpadTrace| '|No function for| x)
                  (exit nil))))
              (exit t)))))))
```

50.1.52 defun spadTrace

```
[pairp p??]
[refvecp p??]
[aldorTrace p??]
[isDomainOrPackage p849]
[userError p??]
[seq p??]
[exit p??]
[spadTrace,g p849]
[getOption p866]
[removeOption p835]
[opOf p??]
[assoc p??]
[kdr p??]
[flattenOperationAlist p857]
[getOperationAlistFromLisplib p??]
[spadTrace,isTraceable p849]
[as-insert p??]
```

```

[bpiname p??]
[spadTraceAlias p865]
[subTypes p840]
[constructSubst p??]
[bpitrace p??]
[rplac p??]
[printDashedLine p??]
[reportSpadTrace p866]
[setletprintflag p??]
[spadReply p869]
[$tracedModemap p??]
[$fromSpadTrace p??]
[$letAssoc p??]
[$reportSpadTrace p866]
[$traceNoisely p820]
[/tracenames p??]

```

— defun `spadTrace` —

```

(defun |spadTrace| (domain options)
  (let (|$tracedModemap| listofoperations listofvariables
        listofbreakvars anyiftrue domainid currententry
        currentalist opstructurelist sig kind triple fn op
        mm n alias tracename sigslotnumberalist)
    (declare (special |$tracedModemap| /tracenames |$fromSpadTrace| |$letAssoc|
                      |$reportSpadTrace| |$traceNoisely|))
    (setq |$fromSpadTrace| t)
    (setq |$tracedModemap| nil)
    (cond
      ((and (pairp domain)
            (refvecp (car domain))
            (eql (elt (car domain) 0) 0))
        (|aldorTrace| domain options))
      ((null (|isDomainOrPackage| domain))
        (|userError| "bad argument to trace"))
      (t
       (setq listofoperations
              (prog (t0)
                    (setq t0 nil)
                    (return
                     (do ((t1 (|getOption| 'ops options) (cdr t1)) (x nil))
                         ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
                     (seq
                      (exit
                       (setq t0 (cons (|spadTrace,g| x) t0))))))))))
      (cond
        ((setq listofvariables (|getOption| 'vars options))
         (setq options (|removeOption| 'vars options))))

```

```

(cond
  ((setq listofbreakvars (|getOption| 'varbreak options))
    (setq options (|removeOption| 'varbreak options))))
(setq anyiftrue (null listofoperations))
(setq domainid (|opOf| (elt domain 0)))
(setq currententry (|assoc| domain /tracenames))
(setq currentalist (kdr currententry))
(setq opstructurelist
  (|flattenOperationAlist| (|getOperationAlistFromLisplib| domainid)))
(setq sigslotnumberalist
  (prog (t2)
    (setq t2 nil)
    (return
      (do ((t3 opstructurelist (cdr t3)) (t4 nil))
        ((or (atom t3)
          (progn (setq t4 (CAR t3)) nil)
          (progn
            (progn
              (setq op (car t4))
              (setq sig (cadr t4))
              (setq n (caddr t4))
              (setq kind (car (cddddr t4)))) t4)
            nil))
          (nreverse0 t2))
        (seq
          (exit
            (cond
              ((and (eq kind 'elt)
                (or anyiftrue (member op listofoperations))
                (integerp n)
                (|spadTrace, isTraceable|
                  (setq triple
                    (cons op (cons sig (cons n nil)))) domain))
                (setq t2 (cons triple t2))))))))))
      (cond
        (listofvariables
          (do ((t5 sigslotnumberalist (cdr t5)) (t6 nil))
            ((or (atom t5)
              (progn (setq t6 (car t5)) nil)
              (progn (progn (setq n (caddr t6)) t6) nil))
              nil)
            (seq
              (exit
                (progn
                  (setq fn (car (elt domain n)))
                  (setq |$letAssoc|
                    (as-insert (bpiname fn) listofvariables |$letAssoc|))))))
              (cond
                (listofbreakvars
                  (do ((t7 sigslotnumberalist (cdr t7)) (t8 nil))

```

```

      ((or (atom t7)
            (progn (setq t8 (car t7)) nil)
                   (progn (progn (setq n (caddr t8)) t8) nil))
            nil)
      (seq
        (exit
          (progn
            (setq fn (car (elt domain n)))
            (setq |$letAssoc|
              (as-insert (bpiname fn)
                (cons (cons 'break listofbreakvars) nil) |$letAssoc|))))))
      (do ((t9 sigslotnumberalist (cdr t9)) (|pair| nil))
        ((or (atom t9)
              (progn (setq |pair| (car t9)) nil)
                    (progn
                      (progn
                        (setq op (car |pair|))
                        (setq mm (cadr |pair|))
                        (setq n (caddr |pair|))
                        |pair|)
                      nil))
              nil)
          (seq
            (exit
              (progn
                (setq alias (|spadTraceAlias| domainid op n))
                (setq |$tracedModemap|
                  (|subTypes| mm (|constructSubst| (elt domain 0))))
                (setq tracename
                  (bpitrace (car (elt domain n)) alias options))
                (nconc |pair|
                  (cons listofvariables
                    (cons (car (elt domain n))
                      (cons tracename (cons alias nil))))))
                (rplac (car (elt domain n)) tracename))))
            (setq sigslotnumberalist
              (prog (t10)
                (setq t10 nil)
                (return
                  (do ((t11 sigslotnumberalist (cdr t11)) (x nil))
                    ((or (atom t11) (progn (setq x (car t11)) nil)) (nreverse0 t10))
                    (seq
                      (exit
                        (cond ((cdddr x) (setq t10 (cons x t10))))))))))
              (cond ((cdddr x) (setq t10 (cons x t10))))))
          (cond
            (|$reportSpadTrace|
              (cond (|$traceNoisely| (|printDashedLine|))
                (do ((t12 (|orderBySlotNumber| sigslotnumberalist) (cdr t12))
                    (x nil))
                  ((or (atom t12)

```

```

      (progn (setq x (car t12)) nil))
      nil)
      (seq (exit (|reportSpadTrace| 'tracing x))))))
      (cond (|$letAssoc| (setletprintflag t)))
      (cond
        (currententry
         (rplac (cdr currententry)
          (append sigslotnumberalist currentalist)))
        (t
         (setq /tracenames
          (cons (cons domain sigslotnumberalist) /tracenames))
          (|spadReply|))))))

```

50.1.53 defun traceDomainLocalOps

[sayMSG p333]

— defun traceDomainLocalOps —

```

(defun |traceDomainLocalOps| ()
  (|sayMSG| '(" The )local option has been withdrawn"))
  (|sayMSG| '(" Use )ltr to trace local functions.")))

```

50.1.54 defun untraceDomainLocalOps

[sayMSG p333]

— defun untraceDomainLocalOps —

```

(defun |untraceDomainLocalOps| ()
  (|sayMSG| '(" The )local option has been withdrawn"))
  (|sayMSG| '(" Use )ltr to trace local functions.")))

```

50.1.55 defun traceDomainConstructor

[getOption p866]

[seq p??]

```

[exit p??]
[spadTrace p850]
[concat p1003]
[embed p??]
[mkq p??]
[loadFunctor p??]
[traceDomainLocalOps p854]
[$ConstructorCache p??]

```

— **defun traceDomainConstructor** —

```

(defun |traceDomainConstructor| (domainConstructor options)
  (prog (listOfLocalOps arg1 domain innerDomainConstructor)
    (declare (special |$ConstructorCache|))
    (return
      (seq
        (progn
          (|loadFunctor| domainConstructor)
          (setq listOfLocalOps (|getOption| 'local options))
          (when listOfLocalOps (|traceDomainLocalOps|))
          (cond
            ((and listOfLocalOps (null (|getOption| 'ops options))) nil)
            (t
              (do ((t2 (hget |$ConstructorCache| domainConstructor) (cdr t2))
                  (t3 nil))
                ((or (atom t2)
                     (progn (setq t3 (car t2)) nil)
                     (progn
                       (progn
                         (setq arg1 (car t3))
                         (setq domain (cddr t3)) t3)
                       nil))
                  nil)
              (seq
                (exit
                  (|spadTrace| domain options))))
            (setq /tracenames (cons domainConstructor /tracenames))
            (setq innerDomainConstructor
              (intern (concat domainConstructor ";")))
            (cond
              ((fboundp innerDomainConstructor)
               (setq domainConstructor innerDomainConstructor)))
            (embed domainConstructor
              (cons 'lambda
                (cons '&rest
                  (cons 'args nil))
                (cons
                  (cons 'prog

```



```

(cons
  (cons 'domain nil)
  (cons
    (cons 'setq
      (cons 'domain
        (cons
          (cons 'apply (cons domainConstructor
            (cons 'args nil))) nil)))
    (cons
      (cons '|spadTrace|
        (cons 'domain
          (cons (mkq options) nil)))
      (cons (cons 'return (cons 'domain nil)) nil))))
    nil)))))))))

```

50.1.56 defun untraceDomainConstructor,keepTraced?

```

[seq p??]
[pairp p??]
[qcar p??]
[isDomainOrPackage p849]
[boot-equal p??]
[kar p??]
[devaluate p??]
[exit p??]
[/untrace,0 p??]

```

— defun untraceDomainConstructor,keepTraced? —

```

(defun |untraceDomainConstructor,keepTraced?| (df domainConstructor)
  (prog (dc)
    (return
      (seq
        (if (and
          (and
            (and (pairp df) (progn (setq dc (qcar df)) t))
            (|isDomainOrPackage| dc))
          (boot-equal (kar (|devaluate| dc)) domainConstructor))
          (exit (seq (|/UNTRACE,0| (cons dc nil)) (exit nil))))
          (exit t))))))

```

50.1.57 defun untraceDomainConstructor

```
[untraceDomainConstructor,keepTraced? p856]
[unembed p??]
[seq p??]
[exit p??]
[concat p1003]
[delete p??]
[/tracenames p??]
```

— **defun untraceDomainConstructor** —

```
(defun |untraceDomainConstructor| (domainConstructor)
  (prog (innerDomainConstructor)
    (declare (special /tracenames))
    (return
      (seq
        (progn
          (setq /tracenames
            (prog (t0)
              (setq t0 nil)
              (return
                (do ((t1 /tracenames (cdr t1)) (df nil))
                  ((or (atom t1) (progn (setq df (car t1)) nil)) (nreverse0 t0))
                (seq
                  (exit
                    (cond ((|untraceDomainConstructor,keepTraced?|
                          df domainConstructor)
                      (setq t0 (cons df t0))))))))))
          (setq innerDomainConstructor
            (intern (concat domainConstructor ";")))
          (cond
            ((fboundp innerDomainConstructor) (unembed innerDomainConstructor))
            (t (unembed domainConstructor)))
          (setq /tracenames (|delete| domainConstructor /tracenames))))))
```

—

50.1.58 defun flattenOperationAlist

```
[seq p??]
[exit p??]
```

— **defun flattenOperationAlist** —

```
(defun |flattenOperationAlist| (|opAlist|)
  (prog (op |mmList| |res|)
```

```

(return
  (seq
    (progn
      (setq |res| nil)
      (do ((t0 |opAlist| (cdr t0)) (t1 nil))
        ((or (atom t0)
              (progn (setq t1 (car t0)) nil)
              (progn
                (progn (setq op (car t1)) (setq |mmList| (cdr t1)) t1)
                nil))
            nil)
          (seq
            (exit
              (setq |res|
                (append |res|
                  (prog (t2)
                    (setq t2 nil)
                    (return
                      (do ((t3 |mmList| (cdr t3)) (mm nil))
                        ((or (atom t3)
                              (progn (setq mm (car t3)) nil)) (nreverse0 t2))
                        (seq
                          (exit
                            (setq t2 (cons (cons op mm) t2))))))))))
                    |res|))))))

```

50.1.59 defun mapLetPrint

```

[getAliasIfTracedMapParameter p863]
[getBpiNameIfTracedMap p864]
[letPrint p859]

```

— defun mapLetPrint —

```

(defun |mapLetPrint| (x val currentFunction)
  (setq x (|getAliasIfTracedMapParameter| x currentFunction))
  (setq currentFunction (|getBpiNameIfTracedMap| currentFunction))
  (|letPrint| x val currentFunction))

```

50.1.60 defun letPrint

```
[lassoc p??]
[isgenvar p860]
[isSharpVarWithNum p860]
[gensymp p??]
[sayBrightlyNT p??]
[bright p??]
[shortenForPrinting p865]
[hasPair p865]
[pname p1001]
[break p880]
[$letAssoc p??]
```

— defun letPrint —

```
(defun |letPrint| (x |val| |currentFunction|)
  (prog (y)
    (declare (special |$letAssoc|))
    (return
      (progn
        (cond ((and |$letAssoc|
                     (or
                      (setq y (lassoc |currentFunction| |$letAssoc|))
                      (setq y (lassoc '|all| |$letAssoc|))))
              (cond
                ((and (or (eq y '|all|)
                          (member x y))
                     (null
                      (or (isgenvar x) (|isSharpVarWithNum| x) (gensymp x))))
                  (|sayBrightlyNT| (append (|bright| x) (cons '|:| nil)))
                  (prin1 (|shortenForPrinting| |val|))
                  (terpri)))
                (cond
                  ((and (setq y (|hasPair| 'break y))
                       (or (eq y '|all|)
                           (and (member x y)
                                (null (member (elt (pname x) 0) '($ |#|)))
                                (null (gensymp x))))
                     (|break|
                      (append
                       (|bright| |currentFunction|)
                       (cons "breaks after"
                           (append
                            (|bright| x)
                            (cons " := " (cons (|shortenForPrinting| |val|) nil)))))))
                    (t nil))))
          |val|))))
```

50.1.61 defun Identifier beginning with a sharpsign-number?

This tests if x is an identifier beginning with # followed by a number. [isSharpVar p860]

```
[pname p1001]
[qcsize p??]
[digitp p1001]
[dig2fix p??]
```

— defun isSharpVarWithNum —

```
(defun |isSharpVarWithNum| (x)
  (let (p n d ok c)
    (cond
      ((null (|isSharpVar| x)) nil)
      ((> 2 (setq n (qcsize (setq p (pname x))))) nil)
      (t
       (setq ok t)
       (setq c 0)
       (do ((t1 (1- n)) (i 1 (1+ i)))
           ((or (> i t1) (null ok)) nil)
          (setq d (elt p i))
          (when (setq ok (digitp d))
            (setq c (+ (* 10 c) (dig2fix d))))))
       (when ok c)))))
```

50.1.62 defun Identifier beginning with a sharpsign?

This tests if x is an identifier beginning with # [identp p1003]

— defun isSharpVar —

```
(defun |isSharpVar| (x)
  (and (identp x) (char= (schar (symbol-name x) 0) #\#)))
```

50.1.63 defun isgenvar

```
[size p1001]
[digitp p1001]
```

[identp p1003]

— defun isgenvar —

```
(defun isgenvar (x)
  (and (identp x)
    (let ((y (symbol-name x)))
      (and (char= #\$ (elt y 0)) (> (size y) 1) (digitp (elt y 1))))))
```

—————

50.1.64 defun letPrint2

[letPrint2 p861]
 [lassoc p??]
 [isgenvar p860]
 [isSharpVarWithNum p860]
 [gensymp p??]
 [mathprint p??]
 [print p??]
 [hasPair p865]
 [pname p1001]
 [break p880]
 [bright p??]
 [\$BreakMode p635]
 [\$letAssoc p??]

— defun letPrint2 —

```
(defun |letPrint2| (x |printform| |currentFunction|)
  (prog (|$BreakMode| |flag| y)
    (declare (special |$BreakMode| |$letAssoc|))
    (return
      (progn
        (setq |$BreakMode| nil)
        (cond
          ((and |$letAssoc|
            (or (setq y (lassoc |currentFunction| |$letAssoc|))
              (setq y (lassoc '|all| |$letAssoc|))))
            (cond
              ((and
                (or (eq y '|all|) (member x y))
                (null (or (isgenvar x) (|isSharpVarWithNum| x) (gensymp x))))
                (setq |$BreakMode| '|letPrint2|)
                (setq |flag| nil)
                (catch '|letPrint2|
```

```

(|mathprint| (cons '= (cons x (cons |printform| nil)))) |flag|)
(cond
  ((eq |flag| '|letPrint2|) (|print| |printform|))
  (t nil)))
(cond
  ((and
    (setq y (|hasPair| 'break y))
    (or (eq y '|all|)
        (and
          (member x y)
          (null (member (elt (pname x) 0) '($ |#|)))
          (null (gensymp x)))))
    (|break|
     (append
      (|bright| |currentFunction|)
      (cons "breaks after"
            (append (|bright| x) (cons '|:= | (cons |printform| nil)))))))
    (t nil)))
  x)))

```

50.1.65 defun letPrint3

This is the version for use when we have our hands on a function to convert the data into type "Expression" [letPrint2 p861]

```

[lassoc p??]
[isgenvar p860]
[isSharpVarWithNum p860]
[gensymp p??]
[mathprint p??]
[spadcall p??]
[print p??]
[hasPair p865]
[pname p1001]
[break p880]
[bright p??]
[$BreakMode p635]
[$letAssoc p??]

```

— defun letPrint3 —

```

(defun |letPrint3| (x |xval| |printfn| |currentFunction|)
  (prog (|$BreakMode| |flag| y)
    (declare (special |$BreakMode| |$letAssoc|))
    (return

```

```

(progn
  (setq |$BreakMode| nil)
  (cond
    ((and |$letAssoc|
      (or (setq y (lassoc |currentFunction| |$letAssoc|))
        (setq y (lassoc '|all| |$letAssoc|))))
      (cond
        ((and
          (or (eq y '|all|) (member x y))
          (null (or (isgenvar x) (isSharpVarWithNum| x) (gensymp x))))
          (setq |$BreakMode| '|letPrint2|)
          (setq |flag| nil)
          (catch '|letPrint2|
            (|mathprint|
              (cons '= (cons x (cons (spadcall |xval| |printfn|) nil))))
            |flag|)
          (cond
            ((eq |flag| '|letPrint2|) (|print| |xval|))
            (t nil))))
        (cond
          ((and
            (setq y (|hasPair| 'break y))
            (or
              (eq y '|all|)
              (and
                (member x y)
                (null (member (elt (pname x) 0) '($ |#|)))
                (null (gensymp x))))
            (|break|
              (append
                (|bright| |currentFunction|)
                (cons "breaks after"
                  (append (|bright| x) (cons " := " (cons |xval| nil)))))))
            (t nil))))
      (t nil))))
  x)))

```

50.1.66 defun getAliasIfTracedMapParameter

```

[isSharpVarWithNum p860]
[get p??]
[exit p??]
[spaddifference p??]
[string2pint-n p??]
[substring p??]
[pname p1001]

```



```
[seq p??]
[$InteractiveFrame p??]
```

— **defun getAliasIfTracedMapParameter** —

```
(defun |getAliasIfTracedMapParameter| (x |currentFunction|)
  (prog (|aliasList|)
    (declare (special |$InteractiveFrame|))
    (return
      (seq
        (cond
          ((|isSharpVarWithNum| x)
            (cond
              ((setq |aliasList|
                (|get| |currentFunction| 'alias |$InteractiveFrame|))
              (exit
                (elt |aliasList|
                  (spaddifference
                    (string2pint-n (substring (pname x) 1 nil) 1) 1))))))
          (t x))))))
```

—————

50.1.67 **defun getBpiNameIfTracedMap**

```
[get p??]
[exit p??]
[seq p??]
[$InteractiveFrame p??]
[/tracenames p??]
```

— **defun getBpiNameIfTracedMap** —

```
(defun |getBpiNameIfTracedMap| (name)
  (prog (lmm bpiName)
    (declare (special |$InteractiveFrame| /tracenames))
    (return
      (seq
        (cond
          ((setq lmm (|get| name '|localModemap| |$InteractiveFrame|))
            (cond
              ((member (setq bpiName (cadar lmm)) /tracenames)
                (exit bpiName))))
          (t name))))))
```

—————

50.1.68 defun hasPair

```
[pairp p??]
[qcar p??]
[qcdr p??]
[hasPair p865]
```

— defun hasPair —

```
(defun |hasPair| (key arg)
  (prog (tmp1 a)
    (return
      (cond
        ((atom arg) nil)
        ((and (pairp arg)
              (progn
                (setq tmp1 (qcar arg))
                (and (pairp tmp1)
                     (equal (qcar tmp1) key)
                     (progn (setq a (qcdr tmp1)) t))))
          a)
        (t (|hasPair| key (cdr arg)))))))
```

—————

50.1.69 defun shortenForPrinting

```
[isDomainOrPackage p849]
[devaluate p??]
```

— defun shortenForPrinting —

```
(defun |shortenForPrinting| (|val|)
  (if (|isDomainOrPackage| |val|)
      (|devaluate| |val|)
      |val|))
```

—————

50.1.70 defun spadTraceAlias

```
[internl p??]
```

— defun spadTraceAlias —

```
(defun |spadTraceAlias| (domainid op n)
  (internl domainid (intern "." "boot") op '|,| (princ-to-string n)))
```

50.1.71 defun getOption

```
[assoc p??]
```

— defun getOption —

```
(defun |getOption| (opt l)
  (let (y)
    (when (setq y (|assoc| opt l)) (cdr y))))
```

50.1.72 defun reportSpadTrace

```
[pairp p??]
[qcar p??]
[sayBrightly p??]
[$traceNoisely p820]
```

— defun reportSpadTrace —

```
(defun |reportSpadTrace| (|header| t0)
  (prog (op sig n |t| |msg| |namePart| y |tracePart|)
    (declare (special |$traceNoisely|))
    (return
      (progn
        (setq op (car t0))
        (setq sig (cadr t0))
        (setq n (caddr t0))
        (setq |t| (cdddd t0))
        (cond
          ((null |$traceNoisely|) nil)
          (t
            (setq |msg|
              (cons |header|
                (cons '|%b|
                  (cons op
                    (cons '|:|
                      (cons '|%d|
                        (cons (CDR sig)))))))))))))
```

```

      (cons '| -> |
        (cons (car sig)
          (cons '| in slot |
            (cons n nil)))))))))
(setq |namePart| nil)
(setq |tracePart|
  (cond
    ((and (pairp |t|) (progn (setq y (qcar |t|)) t) (null (null y)))
      (cond
        ((eq y '|all|)
          (cons '|%b| (cons '|all| (cons '|%d| (cons '|vars| nil)))))
        (t (cons '| vars: | (cons y nil)))))
      (t nil)))
    (|sayBrightly| (append |msg| (append |namePart| |tracePart|)))))))))

```

50.1.73 defun orderBySlotNumber

```

[seq p??]
[assocright p??]
[orderList p??]
[exit p??]

```

— defun orderBySlotNumber —

```

(defun |orderBySlotNumber| (arg)
  (prog (n)
    (return
      (seq
        (assocright
          (|orderList|
            (prog (t0)
              (setq t0 nil)
              (return
                (do ((t1 arg (cdr t1)) (x nil))
                  ((or (atom t1)
                      (progn (setq x (car t1)) nil)
                      (progn (progn (setq n (caddr x)) x) nil))
                  (nreverse0 t0)))
            (seq
              (exit
                (setq t0 (cons (cons n x) t0))))))))))

```

50.1.74 defun /tracereply

```
[pairp p??]
[qcar p??]
[isDomainOrPackage p849]
[devaluate p??]
[seq p??]
[exit p??]
[/tracenames p??]
```

— **defun /tracereply** —

```
(defun /tracereply ()
  (prog (|d| domainlist |functionList|)
    (declare (special /tracenames))
    (return
      (seq
        (cond
          ((null /tracenames) " Nothing is traced.")
          (t
            (do ((t0 /tracenames (cdr t0)) (x nil))
              ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
            (seq
              (exit
                (cond
                  ((and (pairp x)
                     (progn (setq |d| (qcar x)) t)
                     (|isDomainOrPackage| |d|))
                   (setq domainlist (cons (|devaluate| |d|) domainlist)))
                (t
                  (setq |functionList| (cons x |functionList|)))))))
              (append |functionList|
                (append domainlist (cons '|traced| nil))))))))))
```

—————

50.1.75 defun spadReply,printName

```
[seq p??]
[pairp p??]
[qcar p??]
[isDomainOrPackage p849]
[exit p??]
[devaluate p??]
```

— **defun spadReply,printName** —

```
(defun |spadReply,printName| (x)
  (prog (|d|)
    (return
      (seq
        (if (and (and (pairp x) (progn (setq |d| (qcar x)) t))
              (|isDomainOrPackage| |d|))
            (exit (|devaluate| |d|)))
          (exit x))))))
```

50.1.76 defun spadReply

```
[seq p??]
[exit p??]
[spadReply,printName p868]
[/tracenames p??]
```

— defun spadReply —

```
(defun |spadReply| ()
  (prog ()
    (declare (special /tracenames))
    (return
      (seq
        (prog (t0)
          (setq t0 nil)
          (return
            (do ((t1 /tracenames (cdr t1)) (x nil))
              ((or (atom t1) (progn (setq x (car t1)) nil)) (nreverse0 t0))
            (seq
              (exit
                (setq t0 (cons (|spadReply,printName| x) t0)))))))))))
```

50.1.77 defun spadUntrace

```
[isDomainOrPackage p849]
[userError p??]
[getOption p866]
[devaluate p??]
[assoc p??]
[sayMSG p333]
[bright p??]
```

```

[prefix2String p??]
[bpname p??]
[remover p??]
[setletprintflag p??]
[bpiuntrace p??]
[rplac p??]
[seq p??]
[exit p??]
[delasc p??]
[spadReply p869]
[$letAssoc p??]
[/tracenames p??]

```

— **defun spadUntrace** —

```

(defun |spadUntrace| (domain options)
  (prog (anyiftrue listofoperations domainid |pair| sigslotnumberalist
        op sig n |lv| |bpiPointer| tracename alias |assocPair|
        |newSigSlotNumberAlist|)
    (declare (special |$letAssoc| /tracenames))
    (return
     (seq
      (cond
       ((null (|isDomainOrPackage| domain))
        (|userError| "bad argument to untrace")))
      (t
       (setq anyiftrue (null options))
       (setq listofoperations (|getOption| 'ops:| options))
       (setq domainid (|devaluate| domain))
       (cond
        ((null (setq |pair| (|assoc| domain /tracenames)))
         (|sayMSG|
          (cons " No functions in"
                (append
                 (|bright| (|prefix2String| domainid))
                 (cons "are now traced." nil))))))
        (t
         (setq sigslotnumberalist (cdr |pair|))
         (do ((t0 sigslotnumberalist (cdr t0)) (|pair| nil))
             ((or (atom t0)
                  (progn (setq |pair| (car t0)) nil)
                  (progn
                     (progn
                      (setq op (car |pair|))
                      (setq sig (cadr |pair|))
                      (setq n (caddr |pair|))
                      (setq |lv| (caddr |pair|))
                      (setq |bpiPointer| (car (cddddr |pair|)))
                      (setq tracename (cadr (cddddr |pair|)))

```

```

        (setq alias (caddr (cddddr |pair|)))
        |pair|)
      nil))
    nil)
  (seq
    (exit
      (cond
        ((or anyiftrue (member op listofoperations))
          (progn
            (bpiuntrace tracename alias)
            (rplac (car (elt domain n)) |bpiPointer|)
            (rplac (cddddr |pair|) nil)
            (cond
              ((setq |assocPair|
                (|assoc| (bpiname |bpiPointer|) |$letAssoc|))
                (setq |$letAssoc| (remover |$letAssoc| |assocPair|))
                (cond
                  ((null |$letAssoc|) (setletprintflag nil))
                  (t nil)))
              (t nil)))))))))
  (setq |newSigSlotNumberAlist|
    (prog (t1)
      (setq t1 nil)
      (return
        (do ((t2 sigslotnumberalist (cdr t2)) (x nil))
          ((or (atom t2) (progn (setq x (car t2)) nil)) (nreverse0 t1))
          (seq
            (exit
              (cond ((cddddr x) (setq t1 (cons x t1))))))))))
  (cond
    (|newSigSlotNumberAlist|
      (rplac (cdr |pair|) |newSigSlotNumberAlist|))
    (t
      (setq /tracenames (delasc domain /tracenames))
      (|spadReply|))))))

```

50.1.78 defun prTraceNames,fn

```

[seq p??]
[paip p??]
[qcar p??]
[qcdr p??]
[isDomainOrPackage p849]
[exit p??]
[devaluate p??]

```


— defun prTraceNames,fn —

```
(defun |prTraceNames,fn| (x)
  (prog (|d| |t|)
    (return
      (seq
        (if (and (and (pairp x)
                     (progn (setq |d| (qcar x)) (setq |t| (qcdr x)) t))
            (|isDomainOrPackage| |d|))
            (exit (cons (|devalue| |d|) |t|)))
          (exit x))))))
```

—————

50.1.79 defun prTraceNames

```
[seq p??]
[exit p??]
[prTraceNames,fn p871]
[/tracenames p??]
```

— defun prTraceNames —

```
(defun |prTraceNames| ()
  (declare (special /tracenames))
  (seq
    (progn
      (do ((t0 /tracenames (cdr t0)) (x nil))
          ((or (atom t0) (progn (setq x (car t0)) nil)) nil)
        (seq
          (exit
            (print (|prTraceNames,fn| x)))))) nil)))
```

—————

50.1.80 defvar \$constructors

— initvars —

```
(defvar |$constructors| nil)
```

—————

50.1.81 defun traceReply

```

[sayMessage p??]
[sayBrightly p??]
[pairp p??]
[qcar p??]
[isDomainOrPackage p849]
[addTraceItem p876]
[isFunctor p??]
[isgenvar p860]
[userError p??]
[seq p??]
[exit p??]
[isSubForRedundantMapName p847]
[rassocSub p845]
[poundsign p??]
[sayMSG p333]
[sayBrightlyLength p??]
[flowSegmentedMsg p??]
[concat p1003]
[prefix2String p??]
[abbreviate p??]
[$domains p??]
[$packages p??]
[$constructors p872]
[$linelength p751]
[/tracenames p??]

```

— defun traceReply —

```

(defun |traceReply| ()
  (prog (|$domains| |$packages| |$constructors| |d| |functionList|
        |displayList|)
    (declare (special |$domains| |$packages| |$constructors| /tracenames
                      $linelength))
    (return
     (seq
      (progn
       (setq |$domains| nil)
       (setq |$packages| nil)
       (setq |$constructors| nil)
       (cond
        ((null /tracenames) (|sayMessage| "  Nothing is traced now."))
        (t
         (|sayBrightly| " ")
         (do ((t0 /tracenames (cdr t0)) (x nil))
              ((or (atom t0) (progn (setq x (car t0)) nil)) nil)

```

```

(seq
  (exit
    (cond
      ((and (pairp x)
        (progn (setq |d| (qcar x)) t) (|isDomainOrPackage| |d|))
        (|addTraceItem| |d|))
      ((atom x)
        (cond
          ((|isFunctor| x) (|addTraceItem| x))
          ((|isgenvar| x) (|addTraceItem| (EVAL x)))
          (t (setq |functionList| (cons x |functionList|))))
        (t (|userError| "bad argument to trace")))))
  (setq |functionList|
    (prog (t1)
      (setq t1 nil)
      (return
        (do ((t2 |functionList| (cdr t2)) (x nil))
          ((or (atom t2) (progn (setq x (car t2)) nil)) t1)
          (seq
            (exit
              (cond
                ((null (|isSubForRedundantMapName| x))
                  (setq t1
                    (append t1
                      (cons (|rassocSub| x |$mapSubNameAlist|)
                        (cons " " nil))))))))))
        (cond
          (|functionList|
            (cond
              ((= 2 (|#| |functionList|))
                (|sayMSG| (cons '| Function traced: | |functionList|)))
              ((<= (+ 22 (|sayBrightlyLength| |functionList|)) $linelength)
                (|sayMSG| (cons '| Functions traced: | |functionList|)))
              (t
                (|sayBrightly| " Functions traced:")
                (|sayBrightly|
                  (|flowSegmentedMsg| |functionList| $linelength 6))))))
            (cond
              (|$domains|
                (setq |displayList|
                  (|concat|
                    (|prefix2String| (CAR |$domains|))
                    (prog (t3)
                      (setq t3 nil)
                      (return
                        (do ((t4 (cdr |$domains|) (cdr t4)) (x nil))
                          ((or (atom t4) (progn (setq x (car t4)) nil)) t3)
                          (seq
                            (exit
                              (setq t3

```

```

        (append t3 (|concat| "," " " (|prefix2String| x)))))))))
(cond
  ((atom |displayList|)
   (setq |displayList| (cons |displayList| nil)))
  (|sayBrightly| " Domains traced: ")
  (|sayBrightly| (|flowSegmentedMsg| |displayList| $linelength 6)))
(cond
  (|$packages|
   (setq |displayList|
    (|concat|
     (|prefix2String| (CAR |$packages|))
     (prog (t5)
      (setq t5 nil)
      (return
       (do ((t6 (cdr |$packages|) (cdr t6)) (x nil))
        ((or (atom t6) (progn (setq x (car t6)) nil)) t5)
        (seq
         (exit
          (setq t5
           (append t5 (|concat| ', | (|prefix2String| x))))))))))
    (cond ((atom |displayList|)
           (setq |displayList| (cons |displayList| nil)))
          (|sayBrightly| " Packages traced: ")
          (|sayBrightly| (|flowSegmentedMsg| |displayList| $linelength 6)))
  (|$constructors|
   (setq |displayList|
    (|concat|
     (|abbreviate| (CAR |$constructors|))
     (prog (t7)
      (setq t7 nil)
      (return
       (do ((t8 (cdr |$constructors|) (cdr t8)) (x nil))
        ((or (atom t8) (progn (setq x (car t8)) nil)) t7)
        (seq
         (exit
          (setq t7
           (append t7 (|concat| ', | (|abbreviate| x))))))))))
    (cond ((atom |displayList|)
           (setq |displayList| (cons |displayList| nil)))
          (|sayBrightly| " Parameterized constructors traced:")
          (|sayBrightly| (|flowSegmentedMsg| |displayList| $linelength 6)))
  (t nil))))))

```

50.1.82 defun addTraceItem

```
[constructor? p??]
[isDomain p??]
[devaluate p??]
[isDomainOrPackage p849]
[$constructors p872]
[$domains p??]
[$packages p??]
```

— **defun addTraceItem** —

```
(defun |addTraceItem| (|d|)
  (declare (special |$constructors| |$domains| |$packages|))
  (cond
    ((|constructor?| |d|)
     (setq |$constructors| (cons |d| |$constructors|)))
    ((|isDomain| |d|)
     (setq |$domains| (cons (|devaluate| |d|) |$domains|)))
    ((|isDomainOrPackage| |d|)
     (setq |$packages| (cons (|devaluate| |d|) |$packages|)))))
```

—————

50.1.83 defun ?t

```
[isgenvar p860]
[get p??]
[sayMSG p333]
[bright p??]
[rassocSub p845]
[pairp p??]
[qcar p??]
[qcdr p??]
[isDomainOrPackage p849]
[isDomain p??]
[reportSpadTrace p866]
[take p??]
[sayBrightly p??]
[devaluate p??]
[$mapSubNameAlist p??]
[$InteractiveFrame p??]
[/tracenames p??]
```

— **defun ?t** —

```

(defun |?t| ()
  (let (llm d suffix l)
    (declare (special /tracenames |$InteractiveFrame| |$mapSubNameAlist|))
    (if (null /tracenames)
      (|sayMSG| (|bright| "nothing is traced"))
      (progn
        (dolist (x /tracenames)
          (cond
            ((and (atom x) (null (isgenvar x)))
              (progn
                (cond
                  ((setq llm (|get| x '|localModemap| |$InteractiveFrame|))
                    (setq x (list (cadar llm)))))
                (|sayMSG|
                  '("Function" ,@( |bright| (|rassocSub| x |$mapSubNameAlist|))
                    "traced")))))
            (dolist (x /tracenames)
              (cond
                ((and (pairp x)
                  (progn (setq d (qcar x)) (setq l (qcdr x)) t)
                  (|isDomainOrPackage| d))
                  (progn
                    (setq suffix (cond ((|isDomain| d) "domain") (t "package")))
                    (|sayBrightly|
                      '(" Functions traced in " ,suffix |%b| ,(|devaluate| d) |%d| " :"))
                    (dolist (x (|orderBySlotNumber| l))
                      (|reportSpadTrace| '| | (TAKE 4 x)))
                    (terpri)))))))

```

50.1.84 defun tracelet

```

[gensymp p??]
[stupidIsSpadFunction p880]
[bpiname p??]
[lassoc p??]
[union p??]
[setletprintflag p??]
[isgenvar p860]
[compileBoot p880]
[delete p??]
[$traceletflag p??]
[$QuickLet p??]
[$letAssoc p??]
[$traceletFunctions p??]

```

— defun tracelet —

```

(defun |tracelet| (fn |vars|)
  (prog ($traceletflag |$QuickLet| 1)
    (declare (special $traceletflag |$QuickLet| |$letAssoc|
                      |$traceletFunctions|))
    (return
      (progn
        (cond
          ((and (gensymp fn) (|stupidIsSpadFunction| (eval fn)))
            (setq fn (eval fn))
            (cond
              ((compiled-function-p fn) (setq fn (bpiname fn)))
              (t nil))))
          (cond
            ((eq fn '|Undef|) nil)
            (t
              (setq |vars|
                (cond
                  ((eq |vars| '|all|) '|all|)
                  ((setq 1 (lassoc fn |$letAssoc|)) (|union| |vars| 1))
                  (t |vars|))))
              (setq |$letAssoc| (cons (cons fn |vars|) |$letAssoc|))
              (cond (|$letAssoc| (setletprintflag t)))
              (setq $traceletflag t)
              (setq |$QuickLet| nil)
              (cond
                ((and (null (member fn |$traceletFunctions|))
                  (null (isgenvar fn))
                  (compiled-function-p (symbol-function fn))
                  (null (|stupidIsSpadFunction| fn))
                  (null (gensymp fn)))
                  (progn
                    (setq |$traceletFunctions| (cons fn |$traceletFunctions|))
                    (|compileBoot| fn)
                    (setq |$traceletFunctions|
                      (|delete| fn |$traceletFunctions|))))))))))

```

50.1.85 defun breaklet

```

[gensymp p??]
[stupidIsSpadFunction p880]
[bpiname p??]
[lassoc p??]

```

```
[assoc p??]
[union p??]
[setletprintflag p??]
[compileBoot p880]
[delete p??]
[$QuickLet p??]
[$letAssoc p??]
[$traceletFunctions p??]
```

— defun breaklet —

```
(defun |breaklet| (fn |vars|)
  (prog (|$QuickLet| |fnEntry| |pair|)
    (declare (special |$QuickLet| |$letAssoc| |$traceletFunctions|))
    (return
      (progn
        (cond
          ((and (gensymp fn) (|stupidIsSpadFunction| (eval fn)))
            (setq fn (eval fn))
            (cond
              ((compiled-function-p fn) (setq fn (bpiname fn)))
              (t nil))))
          (cond
            ((eq fn '|Undef|) nil)
            (t
              (setq |fnEntry| (lassoc fn |$letAssoc|))
              (setq |vars|
                (cond
                  ((setq |pair| (lassoc '|break| |fnEntry|))
                    (|union| |vars| (cdr |pair|)))
                  (t |vars|)))
              (setq |$letAssoc|
                (cond
                  ((null |fnEntry|)
                    (cons (cons fn (list (cons '|break| |vars|))) |$letAssoc|)
                    (|pair| (rplacd |pair| |vars|) |$letAssoc|)))
                  (t (|$letAssoc| (setletprintflag t))))
              (setq |$QuickLet| nil)
              (cond
                ((and (null (member fn |$traceletFunctions|))
                  (null (|stupidIsSpadFunction| fn))
                  (null (gensymp fn)))
                  (progn
                    (setq |$traceletFunctions| (cons fn |$traceletFunctions|))
                    (|compileBoot| fn)
                    (setq |$traceletFunctions|
                      (|delete| fn |$traceletFunctions|))))))))))
```

50.1.86 defun stupidIsSpadFunction

```
[strpos p1002]
[pname p1001]
```

— defun stupidIsSpadFunction —

```
(defun |stupidIsSpadFunction| (fn)
  (strpos ";" (pname fn) 0 nil))
```

50.1.87 defun break

```
[MONITOR,EVALTRAN p??]
[enable-backtrace p??]
[sayBrightly p??]
[interrupt p??]
[/breakcondition p??]
```

— defun break —

```
(defun |break| (msg)
  (prog (condition)
    (declare (special /breakcondition))
    (return
      (progn
        (setq condition (|MONITOR,EVALTRAN| /breakcondition nil))
        (enable-backtrace nil)
        (when (eval condition)
          (|sayBrightly| msg)
          (interrupt))))))
```

50.1.88 defun compileBoot

```
[/D,1 p??]
```

— defun compileBoot —

```
(defun |compileBoot| (fn)
  (|/D,1| (list fn) '(/comp) nil nil))
```

Chapter 51

)undo help page Command

51.1 undo help page man page

— undo.help —

```
=====
A.27. )undo
=====
```

User Level Required: interpreter

Command Syntax:

-)undo
-)undo integer
-)undo integer [option]
-)undo)redo

where option is one of

-)after
-)before

Command Description:

This command is used to restore the state of the user environment to an earlier point in the interactive session. The argument of an)undo is an integer which must designate some step number in the interactive session.

```
)undo n
)undo n )after
```

These commands return the state of the interactive environment to that immediately after step *n*. If *n* is a positive number, then *n* refers to step number *n*. If *n* is a negative number, it refers to the *n*th previous command (that is, undoes the effects of the last *-n* commands).

A `)clear` all resets the `)undo` facility. Otherwise, an `)undo` undoes the effect of `)clear` with options `properties`, `value`, and `mode`, and that of a previous `undo`. If any such system commands are given between steps *n* and *n* + 1 (*n* > 0), their effect is undone for `)undo m` for any $0 < m \leq n$.

The command `)undo` is equivalent to `)undo -1` (it undoes the effect of the previous user expression). The command `)undo 0` undoes any of the above system commands issued since the last user expression.

`)undo n)before`

This command returns the state of the interactive environment to that immediately before step *n*. Any `)undo` or `)clear` system commands given before step *n* will not be undone.

`)undo)redo`

This command reads the file `redo.input`, created by the last `)undo` command. This file consists of all user input lines, excluding those backtracked over due to a previous `)undo`.

The command `)history)write` will eliminate the “undone” command lines of your program.

Also See:

- o `)history`

1

51.2 Data Structures

`$frameRecord = [delta1, delta2, ...]` where `delta(i)` contains changes in the “backwards” direction. Each `delta(i)` has the form `((var . proplist)...)` where `proplist` denotes an ordinary `proplist`. For example, an entry of the form `((x (value) (mode (Integer)))...)` indicates that to undo 1 step, *x*’s value is cleared and its mode should be set to `(Integer)`.

A `delta(i)` of the form `(systemCommand . delta)` is a special `delta` indicating changes due to system commands executed between the last command and the current command. By recording these `deltas` separately, it is possible to undo to either BEFORE or AFTER the

¹ “history” (34.4.7 p 560)

command. These special delta(i)s are given ONLY when a a system command is given which alters the environment.

Note: recordFrame('system) is called before a command is executed, and recordFrame('normal) is called after (see processInteractive1). If no changes are found for former, no special entry is given.

The \$previousBindings is a copy of the CAAR \$InteractiveFrame. This is used to compute the delta(i)s stored in \$frameRecord.

51.3 Functions

51.3.1 Initial Undo Variables

```
$undoFlag := true      --Default setting for undo is "on"
$frameRecord := nil    --Initial setting for frame record
$previousBindings := nil
```

51.3.2 defvar \$undoFlag

— initvars —

```
(defvar |$undoFlag| t "t means we record undo information")
```

—————

51.3.3 defvar \$frameRecord

— initvars —

```
(defvar |$frameRecord| nil "a list of value changes")
```

—————

51.3.4 defvar \$previousBindings

— initvars —

```
(defvar |$previousBindings| nil "a copy of Interactive Frame info for undo")
```

—————

51.3.5 defvar \$reportUndo

— initvars —

```
(defvar |$reportUndo| nil "t means we report the steps undo takes")
```

—————

51.3.6 defun undo

```
[stringPrefix? p??]  
[pname p1001]  
[read p620]  
[userError p??]  
[pairp p??]  
[qcdr p??]  
[qcar p??]  
[spaddifference p??]  
[identp p1003]  
[undoSteps p894]  
[undoCount p893]  
[$options p??]  
[$InteractiveFrame p??]
```

— defun undo —

```
(defun |undo| (l)  
  (let (tmp1 key s undoWhen n)  
    (declare (special |$options| |$InteractiveFrame|))  
    (setq undoWhen '|after|)  
    (when  
      (and (pairp |$options|)  
           (eq (qcdr |$options|) nil)  
           (progn  
             (setq tmp1 (qcar |$options|))  
             (and (pairp tmp1)  
                  (eq (qcdr tmp1) nil)  
                  (progn (setq key (qcar tmp1)) t))))  
      (cond  
        ((|stringPrefix?| (setq s (pname key)) "redo")  
         (setq |$options| nil)  
         (|read| '(|redo.input|)))  
        ((null (|stringPrefix?| s "before"))  
         (|userError| "only option to undo is \"redo\""))  
        (t
```

```

      (setq undoWhen '|before|))))))
(if (null 1)
  (setq n (spaddifference 1))
  (setq n (car 1)))
(when (identp n)
  (setq n (parse-integer (pname n)))
  (unless (integerp n)
    (userError| "undo argument must be an integer")))
(setq |$InteractiveFrame| (|undoSteps| (|undoCount| n) undoWhen))
nil))

```

51.3.7 defun recordFrame

```

[kar p??]
[diffAlist p888]
[seq p??]
[exit p??]
[$undoFlag p885]
[$frameRecord p885]
[$InteractiveFrame p??]
[$previousBindings p885]

```

— defun recordFrame —

```

(defun |recordFrame| (systemNormal)
  (prog (currentAlist delta)
    (declare (special |$undoFlag| |$frameRecord| |$InteractiveFrame|
      |$previousBindings|))
    (return
      (seq
        (cond
          ((null |$undoFlag|) nil)
          (t
            (setq currentAlist (kar |$frameRecord|))
            (setq delta
              (|diffAlist| (caar |$InteractiveFrame|) |$previousBindings|))
            (cond
              ((eq systemNormal '|system|)
                (cond
                  ((null delta)
                    (return nil)))
                (t
                  (setq delta (cons '|systemCommand| delta))))))
            (setq |$frameRecord| (cons delta |$frameRecord|))
            (setq |$previousBindings|

```



```

(prog (tmp0)
  (setq tmp0 nil)
  (return
    (do ((tmp1 (caar |$InteractiveFrame|) (cdr tmp1)) (x nil))
      ((or (atom tmp1)
        (progn (setq x (car tmp1)) nil))
       (nreverse0 tmp0))
    (seq
      (exit
        (setq tmp0
          (cons
            (cons
              (car x)
              (prog (tmp2)
                (setq tmp2 nil)
                (return
                  (do ((tmp3 (cdr x) (cdr tmp3)) (y nil))
                    ((or (atom tmp3)
                      (progn (setq y (car tmp3)) nil))
                     (nreverse0 tmp2)))
                (seq
                  (exit
                    (setq tmp2 (cons (cons (car y) (cdr y)) tmp2))))))
                  tmp0))))))
      (car |$frameRecord|))))))

```

51.3.8 defun diffAlist

```

diffAlist(new,old) ==
--record only those properties which are different
for (pair := [name,:proplist]) in new repeat
  -- name has an entry both in new and old world
  -- (1) if the old world had no proplist for that variable, then
  --   record NIL as the value of each new property
  -- (2) if the old world does have a proplist for that variable, then
  --   a) for each property with a value: give the old value
  --   b) for each property missing:      give NIL as the old value
oldPair := ASSQ(name,old) =>
  null (oldProplist := CDR oldPair) =>
  --record old values of new properties as NIL
  acc := [ [name,:[ [prop] for [prop,.] in proplist] ],:acc]
  deltas := nil
  for (propval := [prop,:val]) in proplist repeat
    null (oldPropval := ASSOC(prop,oldProplist)) => --missing property
    deltas := [ [prop],:deltas]
  EQ(CDR oldPropval,val) => 'skip

```

```

    deltas := [oldPropval,:deltas]
    deltas => acc := [ [name,:NREVERSE deltas],:acc]
    acc := [ [name,:[ [prop] for [prop,:.] in proplist] ],:acc]
--record properties absent on new list (say, from a )cl all)
    for (oldPair := [name,:r]) in old repeat
      r and null LASSQ(name,new) =>
        acc := [oldPair,:acc]
    -- name has an entry both in new and old world
    -- (1) if the new world has no proplist for that variable
    --     (a) if the old world does, record the old proplist
    --     (b) if the old world does not, record nothing
    -- (2) if the new world has a proplist for that variable, it has
    --     been handled by the first loop.
    res := NREVERSE acc
    if BOUNDP '$reportUndo and $reportUndo then reportUndo res
    res

```

```

[assq p1006]
[tmp1 p??]
[seq p??]
[exit p??]
[assoc p??]
[lassq p??]
[reportUndo p891]

```

— defun diffAlist —

```

(defun |diffAlist| (new old)
  (prog (proplist oldPair oldProplist val oldPropval deltas prop name r acc res)
    (return
      (seq
        (progn
          (do ((tmp0 new (cdr tmp0)) (pair nil))
            ((or (atom tmp0)
              (progn (setq pair (car tmp0)) nil)
              (progn
                (progn
                  (setq name (car pair))
                  (setq proplist (cdr pair))
                  pair)
                nil))
            nil)
          (seq
            (exit
              (cond
                ((setq oldPair (assq name old))
                 (cond
                   ((null (setq oldProplist (cdr oldPair)))
                    (setq acc

```

```

(cons
  (cons
    name
    (prog (tmp1)
      (setq tmp1 nil)
      (return
        (do ((tmp2 proplist (cdr tmp2)) (tmp3 nil))
          ((or (atom tmp2)
              (progn (setq tmp3 (car tmp2)) nil)
              (progn
                (progn (setq prop (car tmp3)) tmp3)
                nil))
            (nreverse0 tmp1)))
          (seq
            (exit
              (setq tmp1 (cons (cons prop nil) tmp1)))))))
    acc)))
(t
  (setq deltas nil)
  (do ((tmp4 proplist (cdr tmp4)) (|propval| nil))
    ((or (atom tmp4)
        (progn (setq |propval| (car tmp4)) nil)
        (progn
          (progn
            (setq prop (car |propval|))
            (setq val (cdr |propval|))
            |propval|)
          nil))
      nil)
    nil)
  (seq
    (exit
      (cond
        ((null (setq oldPropval (|assoc| prop oldProplist)))
          (setq deltas (cons (cons prop nil) deltas)))
        ((eq (cdr oldPropval) val) '|skip|)
        (t (setq deltas (cons oldPropval deltas))))))
    (when deltas
      (setq acc
        (cons (cons name (nreverse deltas)) acc))))))
(t
  (setq acc
    (cons
      (cons
        name
        (prog (tmp5)
          (setq tmp5 nil)
          (return
            (do ((tmp6 proplist (cdr tmp6)) (tmp7 nil))
              ((or (atom tmp6)
                  (progn (setq tmp7 (CAR tmp6)) nil)

```

```

      (progn
        (progn (setq prop (CAR tmp7)) tmp7)
        nil))
      (nreverse0 tmp5))
    (seq
      (exit
        (setq tmp5 (cons (cons prop nil) tmp5))))))
    acc))))))
(seq
  (do ((tmp8 old (cdr tmp8)) (oldPair nil))
      ((or (atom tmp8)
            (progn (setq oldPair (car tmp8)) nil)
            (progn
              (progn
                (setq name (car oldPair))
                (setq r (cdr oldPair))
                oldPair)
              nil))
           nil)
    (seq
      (exit
        (cond
          ((and r (null (lassq name new)))
            (exit
              (setq acc (cons oldPair acc))))))
      (setq res (nreverse acc))
      (cond
        ((and (boundp '$reportUndo) |$reportUndo|)
          (|reportUndo| res)))
      (exit res))))))

```

51.3.9 defun reportUndo

This function is enabled by setting `$reportUndo` to a non-nil value. An example of the output generated is:

```
r := binary(22/7)
```

```

      ---
(1)  11.001
                                         Type: BinaryExpansion

Properties of % ::
  value was: NIL
  value is: ((|BinaryExpansion|) WRAPPED . #(1 (1 1) NIL (0 0 1)))
Properties of r ::

```

```
value was: NIL
value is: ((|BinaryExpansion|) WRAPPED . #(1 (1 1) NIL (0 0 1)))
```

```
[seq p??]
[exit p??]
[sayBrightly p??]
[concat p1003]
[pname p1001]
[lassoc p??]
[sayBrightlyNT p??]
[pp p??]
[$InteractiveFrame p??]
```

— defun reportUndo —

```
(defun |reportUndo| (acc)
  (prog (name proplist curproplist prop value)
    (declare (special |$InteractiveFrame|))
    (return
      (seq
        (do ((tmp0 acc (cdr tmp0)) (tmp1 nil))
          ((or (atom tmp0)
              (progn (setq tmp1 (car tmp0)) nil)
              (progn
                (progn
                  (setq name (car tmp1))
                  (setq proplist (cdr tmp1))
                  tmp1)
                nil))
            nil))
          nil)
        (seq
          (exit
            (progn
              (|sayBrightly|
                (concat '|Properties of | (pname name) " ::"))
              (setq curproplist (lassoc name (caar |$InteractiveFrame|)))
              (do ((tmp2 proplist (cdr tmp2)) (tmp3 nil))
                ((or (atom tmp2)
                    (progn (setq tmp3 (car tmp2)) nil)
                    (progn
                      (progn
                        (setq prop (car tmp3))
                        (setq value (cdr tmp3))
                        tmp3)
                      nil))
                  nil)
                nil)
              (seq
                (exit
```

```
(progn
  (|sayBrightlyNT|
    (cons " " (cons prop (cons " was: " nil))))
  (|pp| value)
  (|sayBrightlyNT|
    (cons " " (cons prop (cons " is: " nil))))
  (|pp| (lassoc prop curproplist)))))))))
```

51.3.10 defun clearFrame

```
[clearCmdAll p483]
[$frameRecord p885]
[$previousBindings p885]
```

— defun clearFrame —

```
(defun |clearFrame| ()
  (declare (special |$frameRecord| |$previousBindings|))
  (|clearCmdAll|)
  (setq |$frameRecord| nil)
  (setq |$previousBindings| nil))
```

51.3.11 Undo previous n commands

```
[spaddifference p??]
[userError p??]
[concat p1003]
[$IOindex p??]
```

— defun undoCount —

```
(defun |undoCount| (n)
  "Undo previous n commands"
  (prog (m)
    (declare (special |$IOindex|))
    (return
      (progn
        (setq m
          (cond
            ((>= n 0) (spaddifference (spaddifference |$IOindex| n) 1))
            (t (spaddifference n))))
```

```
(cond
  ((>= m |$IOindex|)
   (|userError|
    (concat "Magnitude of undo argument must be less than step number ("
      (princ-to-string |$IOindex|) ")").)))
  (t m))))))
```

51.3.12 defun undoSteps

```
-- undoes m previous commands; if )before option, then undo one extra at end
--Example: if $IOindex now is 6 and m = 2 then general layout of $frameRecord,
-- after the call to recordFrame below will be:
-- (<change for systemcommands>
-- (<change for #5> <change for system commands>
-- (<change for #4> <change for system commands>
-- (<change for #3> <change for system commands>
-- <change for #2> <change for system commands>
-- <change for #1> <change for system commands>) where system
-- command entries are optional and identified by (systemCommand . change).
-- For a ")undo 3 )after", m = 2 and undoStep swill restore the environment
-- up to, but not including <change for #3>.
-- An "undo 3 )before" will additionally restore <change for #3>.
-- Thus, the later requires one extra undo at the end.
```

```
[writeInputLines p565]
[spaddifference p??]
[recordFrame p887]
[copy p??]
[undoSingleStep p895]
[pairp p??]
[qcdr p??]
[qcar p??]
[$IOindex p??]
[$InteractiveFrame p??]
[$frameRecord p885]
```

— defun undoSteps —

```
(defun |undoSteps| (m beforeOrAfter)
  (let (tmp1 tmp2 systemDelta lastTailSeen env)
    (declare (special |$IOindex| |$InteractiveFrame| |$frameRecord|))
    (|writeInputLines| 'redo| (spaddifference |$IOindex| m))
    (|recordFrame| '|normal|)
    (setq env (copy (caar |$InteractiveFrame|))))
```

```

(do ((i 0 (1+ i)) (framelist |$frameRecord| (cdr framelist)))
  ((or (> i m) (atom framelist)) nil)
  (setq env (|undoSingleStep| (CAR framelist) env))
  (if (and (pairp framelist)
    (progn
      (setq tmp1 (qcdr framelist))
      (and (pairp tmp1)
        (progn
          (setq tmp2 (qcar tmp1))
          (and (pairp tmp2)
            (eq (qcar tmp2) '|systemCommand|)
            (progn
              (setq systemDelta (qcdr tmp2))
              t))))))
      (progn
        (setq framelist (cdr framelist))
        (setq env (|undoSingleStep| systemDelta env)))
        (setq lastTailSeen framelist)))
  (cond
    ((eq beforeOrAfter '|before|)
     (setq env (|undoSingleStep| (car (cdr lastTailSeen)) env))))
  (setq |$frameRecord| (cdr |$frameRecord|))
  (setq |$InteractiveFrame| (list (list env)))))

```

51.3.13 defun undoSingleStep

```

undoSingleStep(changes,env) ==
--Each change is a name-proplist pair. For each change:
-- (1) if there exists a proplist in env, then for each prop-value change:
--   (a) if the prop exists in env, RPLAC in the change value
--   (b) otherwise, CONS it onto the front of prop-values for that name
-- (2) add change to the front of env
-- pp '"----Undoing 1 step-----"
-- pp changes

```

```

[assq p1006]
[seq p??]
[exit p??]
[lassoc p??]
[undoLocalModemapHack p897]

```

— defun undoSingleStep —

```

(defun |undoSingleStep| (changes env)
  (prog (name changeList pairlist proplist prop value node)

```



```

(return
  (seq
    (progn
      (do ((tmp0 changes (cdr tmp0)) (|change| nil))
        ((or (atom tmp0)
              (progn (setq |change| (car tmp0)) nil)
              (progn
                (progn
                  (setq name (car |change|))
                  (setq changeList (cdr |change|))
                  |change|)
                nil))
          nil)
      nil)
    (seq
      (exit
        (progn
          (when (lassoc '|localModemap| changeList)
            (setq changeList (|undoLocalModemapHack| changeList)))
          (cond
            ((setq pairlist (assq name env))
              (cond
                ((setq proplist (cdr pairlist))
                  (do ((tmp1 changeList (cdr tmp1)) (pair nil))
                    ((or (atom tmp1)
                          (progn (setq pair (car tmp1)) nil)
                          (progn
                            (progn
                              (setq prop (car pair))
                              (setq value (cdr pair))
                              pair)
                            nil))
                      nil)
                  (seq
                    (exit
                      (cond
                        ((setq node (assq prop proplist))
                          (rplacd node value))
                        (t
                           (rplacd proplist
                                (cons (car proplist) (cdr proplist)))
                           (rplaca proplist pair))))))
                    (t (rplacd pairlist changeList))))
              (t
                (setq env (cons |change| env))))))
          env))))

```

51.3.14 defun undoLocalModemapHack

```
[seq p??]
[exit p??]
```

— defun undoLocalModemapHack —

```
(defun |undoLocalModemapHack| (changeList)
  (prog (name value)
    (return
      (seq
        (prog (tmp0)
          (setq tmp0 nil)
          (return
            (do ((tmp1 changeList (cdr tmp1)) (pair nil))
              ((or (atom tmp1)
                   (progn (setq pair (car tmp1)) nil)
                   (progn
                     (progn
                       (setq name (car pair))
                       (setq value (cdr pair))
                       pair)
                     nil)))
              (nreverse0 tmp0)))
          (seq
            (exit
              (cond
                ((cond
                  ((eq name '|localModemap|) (cons name nil))
                  (t pair))
                 (setq tmp0
                   (cons
                     (cond
                       ((eq name '|localModemap|) (cons name nil))
                       (t pair)) tmp0))))))))))))))
```

—————

51.3.15 Remove undo lines from history write

Removing undo lines from `)hist)write linelist [stringPrefix? p??]`

```
[seq p??]
[exit p??]
[trimString p??]
[substring p??]
[nequal p??]
[charPosition p??]
```

```
[maxindex p??]
[undoCount p893]
[spaddifference p??]
[concat p1003]
[$currentLine p??]
[$IOindex p??]
```

— **defun removeUndoLines** —

```
(defun |removeUndoLines| (u)
  "Remove undo lines from history write"
  (prog (xtra savedIOindex s s1 m s2 x code c n acc)
    (declare (special |$currentLine| |$IOindex|))
    (return
      (seq
        (progn
          (setq xtra
            (cond
              ((stringp |$currentLine|) (cons |$currentLine| nil))
              (t (reverse |$currentLine|))))
          (setq xtra
            (prog (tmp0)
              (setq tmp0 nil)
              (return
                (do ((tmp1 xtra (cdr tmp1)) (x nil))
                  ((or (atom tmp1)
                      (progn (setq x (car tmp1)) nil))
                   (nreverse0 tmp0)))
                (seq
                  (exit
                    (cond
                      ((null (|stringPrefix?| ")history" x))
                      (setq tmp0 (cons x tmp0))))))))))
            (setq u (append u xtra))
            (cond
              ((null
                (prog (tmp2)
                  (setq tmp2 nil)
                  (return
                    (do ((tmp3 nil tmp2) (tmp4 u (cdr tmp4)) (x nil))
                      ((or tmp3 (atom tmp4) (progn (setq x (car tmp4)) nil)) tmp2)
                    (seq
                      (exit
                        (setq tmp2
                          (or tmp2 (|stringPrefix?| ")undo" x)))))))))
                  u)
                (t
                  (setq savedIOindex |$IOindex|)
                  (setq |$IOindex| 1)
                  (do ((y u (cdr y)))
```

```

      ((atom y) nil)
    (seq
      (exit
        (cond
          ((eql (elt (setq x (car y)) 0) #\ )
            (cond
              ((|stringPrefix?| ")undo"
                (setq s (|trimString| x)))
              (setq s1 (|trimString| (substring s 5 nil)))
              (cond
                ((nequal s1 ")redo")
                (setq m (|charPosition| #\ ) s1 0))
                (setq code
                  (cond
                    ((> (maxindex s1) m) (elt s1 (1+ m)))
                    (t #\a)))
                (setq s2 (|trimString| (substring s1 0 m))))))
            (setq n
              (cond
                ((string= s1 ")redo")
                0)
                ((nequal s2 "")
                 (|undoCount| (parse-integer s2)))
                (t (spaddifference 1))))
              (rplaca y
                (concat ">" code (princ-to-string n))))
            (t nil)))
          (t (setq |$IOindex| (1+ |$IOindex|))))))
    (setq acc nil)
    (do ((y (nreverse u) (cdr y)))
      ((atom y) nil)
      (seq
        (exit
          (cond
            ((eql (elt (setq x (car y)) 0) #\>)
              (setq code (elt x 1))
              (setq n (parse-integer (substring x 2 nil)))
              (setq y (cdr y))
              (do ()
                ((null y) nil)
                (seq
                  (exit
                    (progn
                     (setq c (car y))
                     (cond
                       ((or (eql (elt c 0) #\))
                        (eql (elt c 0) #\>))
                       (setq y (cdr y)))
                     ((eql n 0)
                      (return nil))

```

```
(t
  (setq n (spaddifference n 1))
  (setq y (cdr y))))))
(cond
  ((and y (nequal code #\b))
   (setq acc (cons c acc))))
  (t (setq acc (cons x acc))))))
(setq |$I0index| savedI0index)
acc))))))
```

Chapter 52

)what help page Command

52.1 what help page man page

— what.help —

```
=====
A.28.  )what
=====
```

User Level Required: interpreter

Command Syntax:

```
- )what categories pattern1 [pattern2 ...]
- )what commands  pattern1 [pattern2 ...]
- )what domains   pattern1 [pattern2 ...]
- )what operations pattern1 [pattern2 ...]
- )what packages  pattern1 [pattern2 ...]
- )what synonym   pattern1 [pattern2 ...]
- )what things    pattern1 [pattern2 ...]
- )apropos        pattern1 [pattern2 ...]
```

Command Description:

This command is used to display lists of things in the system. The patterns are all strings and, if present, restrict the contents of the lists. Only those items that contain one or more of the strings as substrings are displayed. For example,

```
)what synonym
```

displays all command synonyms,

)what synonym ver

displays all command synonyms containing the substring ‘‘ver’’,

)what synonym ver pr

displays all command synonyms containing the substring ‘‘ver’’ or the substring ‘‘pr’’. Output similar to the following will be displayed

----- System Command Synonyms -----

user-defined synonyms satisfying patterns:

ver pr

```
)apr ..... )what things
)apropos ..... )what things
)prompt ..... )set message prompt
)version ..... )lisp *yearweek*
```

Several other things can be listed with the)what command:

categories displays a list of category constructors.

commands displays a list of system commands available at your user-level. Your user-level is set via the)set userlevel command. To get a description of a particular command, such as ‘‘()what’’, issue)help what.

domains displays a list of domain constructors.

operations displays a list of operations in the system library.

It is recommended that you qualify this command with one or more patterns, as there are thousands of operations available. For example, say you are looking for functions that involve computation of eigenvalues. To find their names, try)what operations eig. A rather large list of operations is loaded into the workspace when this command is first issued. This list will be deleted when you clear the workspace via)clear all or)clear completely. It will be re-created if it is needed again.

packages displays a list of package constructors.

synonym lists system command synonyms.

things displays all of the above types for items containing the pattern strings as substrings. The command synonym)apropos is equivalent to)what things.

Also See:

- o)display
- o)set
- o)show

1

52.1.1 defvar \$whatOptions

— initvars —

```
(defvar |$whatOptions| '(|operations| |categories| |domains| |packages|
                        |commands| |synonyms| |things|))
```

52.1.2 defun what

[whatSpad2Cmd p904]

— defun what —

```
(defun |what| (l)
  (|whatSpad2Cmd| l))
```

52.1.3 defun whatSpad2Cmd,fixpat

```
[pairp p??]
[qcar p??]
[downcase p??]
```

— defun whatSpad2Cmd,fixpat —

```
(defun |whatSpad2Cmd,fixpat| (x)
  (let (xp)
    (if (and (pairp x) (progn (setq xp (qcar x)) t))
        (downcase xp)
        (downcase x))))
```

¹ “display” (29.2.1 p 513) “set” (44.37.1 p 785) “show” (45.1.1 p 792)

52.1.4 defun whatSpad2Cmd

```
[reportWhatOptions p905]
[selectOptionLC p459]
[sayKeyedMsg p331]
[seq p??]
[exit p??]
[whatSpad2Cmd,fixpat p903]
[whatSpad2Cmd p904]
[filterAndFormatConstructors p908]
[whatCommands p905]
[apropos p909]
[printSynonyms p454]
[$e p??]
[$whatOptions p903]
```

— defun whatSpad2Cmd —

```
(defun |whatSpad2Cmd| (arg)
  (prog (|$e| |key0| key args)
    (declare (special |$e| |$whatOptions|))
    (return
      (seq
        (progn
          (setq |$e| |$EmptyEnvironment|)
          (cond
            ((null arg) (|reportWhatOptions|))
            (t
              (setq |key0| (car arg))
              (setq args (cdr arg))
              (setq key (|selectOptionLC| |key0| |$whatOptions| nil))
              (cond
                ((null key) (|sayKeyedMsg| 's2iz0043 nil))
                (t
                  (setq args
                    (prog (t0)
                      (setq t0 nil)
                      (return
                        (do ((t1 args (cdr t1)) (p nil))
                          ((or (atom t1)
                              (progn (setq p (car t1)) nil))
                           (nreverse0 t0))
                        (seq
                          (exit
                            (setq t0 (cons (|whatSpad2Cmd,fixpat| p) t0))))))))
                  (seq
                    (exit
                      (setq t0 (cons (|whatSpad2Cmd,fixpat| p) t0))))))))
              (seq
                (cond
                  ((eq key '|things|)
```

```

      (do ((t2 |$whatOptions| (cdr t2)) (opt nil))
          ((or (atom t2) (progn (setq opt (CAR t2)) nil)) nil)
          (seq
            (exit
              (cond
                ((null (member opt '(|things|)))
                 (exit (|whatSpad2Cmd| (cons opt args)))))))))
      ((eq key '|categories|)
       (|filterAndFormatConstructors| '|category| "Categories" args))
      ((eq key '|commands|) (|whatCommands| args))
      ((eq key '|domains|)
       (|filterAndFormatConstructors| '|domain| "Domains" args))
      ((eq key '|operations|)
       (|apropos| args))
      ((eq key '|packages|)
       (|filterAndFormatConstructors| '|package| "Packages" args))
      (t
       (cond ((eq key '|synonyms|)
              (|printSynonyms| args)))))))))

```

52.1.5 defun Show keywords for)what command

```

[|sayBrightly| p??]
|$whatOptions| p903]

```

— defun reportWhatOptions —

```

(defun |reportWhatOptions| ()
  (let (optlist)
    (declare (special |$whatOptions|))
    (setq optlist
      (reduce #'append
        (mapcar #'(lambda (x) '(|%l| " " ,x)) |$whatOptions|)))
    (|sayBrightly|
      '(|%b| " )what" |%d| "argument keywords are" |%b| ,@optlist |%d|
        |%l| " or abbreviations thereof." |%l| |%l| " Issue" |%b| ")what ?"
        |%d| "for more information.")))

```

52.1.6 defun The)what commands implementation

```

[|centerAndHighlight| p??]
[|strconc| p??]

```

```
[specialChar p936]
[filterListOfStrings p906]
[commandsForUserLevel p428]
[sayMessage p??]
[blankList p??]
[sayAsManyPerLineAsPossible p??]
[say p??]
[sayKeyedMsg p331]
[$systemCommands p423]
[$linelength p751]
[$UserLevel p784]
```

— **defun whatCommands** —

```
(defun |whatCommands| (patterns)
  (let (label ell)
    (declare (special |$systemCommands| $linelength |$UserLevel|))
    (setq label
      (strconc '|System Commands for User Level: |
        (princ-to-string |$UserLevel|)))
    (|centerAndHighlight| label $linelength (|specialChar| 'hbar|))
    (setq ell
      (|filterListOfStrings| patterns
        (mapcar #'princ-to-string (|commandsForUserLevel| |$systemCommands|))))
    (when patterns
      (if ell
        (|sayMessage|
          ("System commands at this level matching patterns:" |%1| " " |%b|
            ,@(append (|blankList| patterns) (list '|%d|))))
        (|sayMessage|
          ("No system commands at this level matching patterns:" |%1| " " |%b|
            ,@(append (|blankList| patterns) (list '|%d|))))))
    (when ell
      (|sayAsManyPerLineAsPossible| ell)
      (say " "))
    (unless patterns (|sayKeyedMsg| 's2iz0046 nil))))
```

— — —

52.1.7 defun Find all names contained in a pattern

Names and patterns are lists of strings. This returns a list of strings in names that contains any of the strings in the patterns [satisfiesRegularExpressions p907]

— **defun filterListOfStrings** —

```
(defun |filterListOfStrings| (patterns names)
```

```
(let (result)
  (if (or (null patterns) (null names))
      names
      (dolist (name (reverse names) result)
        (when (|satisfiesRegularExpressions| name patterns)
          (push name result))))))
```

52.1.8 defun Find function of names contained in pattern

The argument names and patterns are lists of strings. The argument fn is something like CAR or CADR This returns a list of strings in names that contains any of the strings in patterns [satisfiesRegularExpressions p907]

— defun filterListOfStringsWithFn —

```
(defun |filterListOfStringsWithFn| (patterns names fn)
  (let (result)
    (if (or (null patterns) (null names))
        names
        (dolist (name (reverse names) result)
          (when (|satisfiesRegularExpressions| (funcall fn name) patterns)
            (push name result))))))
```

52.1.9 defun satisfiesRegularExpressions

[strpos p1002]

— defun satisfiesRegularExpressions —

```
(defun |satisfiesRegularExpressions| (name patterns)
  (let ((dname (downcase (copy name))))
    (dolist (pattern patterns)
      (when (strpos pattern dname 0 "@")
        (return-from nil t)))))
```

52.1.10 defun filterAndFormatConstructors

[sayMessage p??]
 [blankList p??]
 [pp2Cols p??]
 [centerAndHighlight p??]
 [specialChar p936]
 [filterListOfStringsWithFn p907]
 [whatConstructors p909]
 [function p??]
 [\$linelength p751]

— defun filterAndFormatConstructors —

```
(defun |filterAndFormatConstructors| (|constrType| label patterns)
  (prog (1)
    (declare (special $linelength))
    (return
      (progn (|centerAndHighlight| label $linelength (|specialChar| '|hbar|))
        (setq 1
          (|filterListOfStringsWithFn| patterns
            (|whatConstructors| |constrType|)
            (|function| cdr)))
          (cond (patterns)
            (cond
              ((null 1)
                (|sayMessage|
                  (cons " No "
                    (cons label
                      (cons " with names matching patterns:"
                        (cons '|%l|
                          (cons " "
                            (cons '|%b|
                              (append (|blankList| patterns)
                                (cons '|%d| nil)))))))))))
                (t
                  (|sayMessage|
                    (cons label
                      (cons " with names matching patterns:"
                        (cons '|%l|
                          (cons " "
                            (cons '|%b|
                              (append (|blankList| patterns)
                                (cons '|%d| nil)))))))))))
              (cond (1 (|pp2Cols| 1)))))))
```

52.1.11 defun whatConstructors

```
[boot-equal p??]
[getdatabase p967]
[seq p??]
[msort p??]
[exit p??]
```

— defun whatConstructors —

```
(defun |whatConstructors| (|constrType|)
  (prog nil
    (return
      (seq
        (msort
          (prog (t0)
            (setq t0 nil)
            (return
              (do ((t1 (|allConstructors|) (cdr t1)) (|con| nil))
                ((or (atom t1) (progn (setq |con| (car t1)) nil)) (nreverse0 t0))
              (seq
                (exit
                  (cond
                    ((boot-equal (getdatabase |con| 'constructorkind)
                               |constrType|)
                     (setq t0
                       (cons
                        (cons
                          (getdatabase |con| 'abbreviation)
                          (string |con|))
                        t0))))))))))))))
```

—————

52.1.12 Display all operation names containing the fragment

Argument *l* is a list of operation name fragments. This displays all operation names containing these fragments [allOperations p992]

```
[filterListOfStrings p906]
[seq p??]
[exit p??]
[downcase p??]
[sayMessage p??]
[sayAsManyPerLineAsPossible p??]
[msort p??]
[sayKeyedMsg p331]
```

— defun apropos —

```
(defun |apropos| (arg)
  "Display all operation names containing the fragment"
  (prog (ops)
    (return
      (seq
        (progn
          (setq ops
            (cond
              ((null arg) (|allOperations|))
              (t
               (|filterListOfStrings|
                (prog (t0)
                  (setq t0 nil)
                  (return
                    (do ((t1 arg (cdr t1)) (p nil))
                      ((or (atom t1) (progn (setq p (car t1)) nil))
                       (nreverse0 t0))
                    (seq (exit (setq t0 (cons (downcase (princ-to-string p)) t0))))))
                  (|allOperations|))))))
          (cond
            (ops
             (|sayMessage| "Operations whose names satisfy the above pattern(s):")
             (|sayAsManyPerLineAsPossible| (msort ops))
             (|sayKeyedMsg| 's2if0011 (cons (car ops) nil)))
            (t
             (|sayMessage| "  There are no operations containing those patterns")
             nil))))))
```

—————

Chapter 53

)with help page Command

53.1 with help page man page

— with.help —

This command is obsolete.
This has been renamed)library.

See also:
o)library

1

53.1.1 defun with

[library p971]

— defun with —

```
(defun |with| (args)
  (|library| args))
```

¹ “library” (63.1.34 p 971)

Chapter 54

)workfiles help page Command

54.1 workfiles help page man page

54.1.1 defun workfiles

[workfilesSpad2Cmd p913]

— defun workfiles —

```
(defun |workfiles| (1)
  (|workfilesSpad2Cmd| 1))
```

—————

54.1.2 defun workfilesSpad2Cmd

```
[throwKeyedMsg p??]
[selectOptionLC p459]
[pathname p998]
[delete p??]
[makeInputFilename p939]
[sayKeyedMsg p331]
[namestring p996]
[updateSourceFiles p524]
[say p??]
[centerAndHighlight p??]
[specialChar p936]
[sortby p??]
[sayBrightly p??]
```

```

[Options p??]
[sourceFiles p??]
[linelength p751]

```

— defun workfilesSpad2Cmd —

```

(defun |workfilesSpad2Cmd| (args)
  (let (deleteflag type flist type1 fl)
    (declare (special |Options| |sourceFiles| $linelength))
    (cond
      (args (|throwKeyedMsg| 's2iz0047 nil))
      (t
        (setq deleteflag nil)
        (do ((t0 |Options| (cdr t0)) (t1 nil))
          ((or (atom t0)
              (progn (setq t1 (car t0)) nil)
              (progn (progn (setq type (car t1)) t1) nil))
          nil)
        (setq type1
          (|selectOptionLC| type '(|boot| |lisp| |meta| |delete|) nil))
        (cond
          ((null type1) (|throwKeyedMsg| 's2iz0048 (cons type nil)))
          ((eq type1 '|delete|) (setq deleteflag t))))
      (do ((t2 |Options| (cdr t2)) (t3 nil))
        ((or (atom t2)
            (progn (setq t3 (CAR t2)) nil)
            (progn
              (progn
                (setq type (car t3))
                (setq flist (cdr t3)) t3)
              nil))
          nil)
        (setq type1 (|selectOptionLC| type '(|boot| |lisp| |meta| |delete|) nil))
        (unless (eq type1 '|delete|)
          (dolist (file flist)
            (setq fl (|pathname| (list file type1 "*")))
            (cond
              (deleteflag
                (setq |sourceFiles| (|delete| fl |sourceFiles|)))
              ((null (makeInputFilename fl))
                (|sayKeyedMsg| 's2iz0035 (list (|namestring| fl))))
              (t (|updateSourceFiles| fl))))))
      (say " ")
      (|centerAndHighlight|
        '| User-specified work files |
        $linelength
        (|specialChar| '|hbar|))
      (say " ")
      (if (null |sourceFiles|)

```

```
(say "  no files specified")
(progn
  (setq |$sourceFiles| (sortby '|pathnameType| |$sourceFiles|))
  (do ((t5 |$sourceFiles| (cdr t5)) (fl nil))
      ((or (atom t5) (progn (setq fl (car t5)) nil)) nil)
      (|sayBrightly| (list "  " (|namestring| fl))))))
```

Chapter 55

)zsystemdevelopment help page Command

55.1 zsystemdevelopment help page man page

55.1.1 defun zsystemdevelopment

[zsystemDevelopmentSpad2Cmd p917]

— defun zsystemdevelopment —

```
(defun |zsystemdevelopment| (arg)
  (|zsystemDevelopmentSpad2Cmd| arg))
```

—————

55.1.2 defun zsystemDevelopmentSpad2Cmd

[zsystemdevelopment1 p918]

[\$InteractiveMode p24]

— defun zsystemDevelopmentSpad2Cmd —

```
(defun |zsystemDevelopmentSpad2Cmd| (arg)
  (declare (special |$InteractiveMode|))
  (|zsystemdevelopment1| arg |$InteractiveMode|))
```

—————

55.1.3 defun zsystemdevelopment1

```
[filenam p??]
[selectOptionLC p459]
[/D,1 p??]
[/comp p??]
[version p??]
[defiostream p938]
[next p38]
[shut p938]
[kar p??]
[kadr p??]
[kaddr p??]
[sayMessage p??]
[sayBrightly p??]
[bright p??]
[$InteractiveMode p24]
[$options p??]
[/wsname p??]
[/version p??]
```

— defun zsystemdevelopment1 —

```
(defun |zsystemdevelopment1| (arg im)
  (let (|$InteractiveMode| fromopt opt optargs newopt opt1 constream upf fun)
    (declare (special |$InteractiveMode| /wsname /version |$options|))
    (setq |$InteractiveMode| im)
    (setq fromopt nil)
    (do ((t0 |$options| (cdr t0)) (t1 nil))
        ((or (atom t0)
              (progn (setq t1 (car t0)) nil)
              (progn
                 (progn
                    (setq opt (CAR t1))
                    (setq optargs (CDR t1))
                    t1)
                 nil)))
         nil)
      (setq opt1 (|selectOptionLC| opt '(|from|) nil))
      (when (eq opt1 '(|from|) (setq fromopt (cons (cons 'from optargs) nil))))
    (do ((t2 |$options| (cdr t2)) (t3 nil))
        ((or (atom t2)
              (progn (setq t3 (car t2)) nil)
              (progn
                 (progn
                    (setq opt (car t3))
                    (setq optargs (cdr t3))
                    t3)
                 nil)))
         nil))
```

```

        nil))
    nil)
(unless optargs (setq optargs arg))
(setq newopt (append optargs fromopt))
(setq opt1 (|selectOptionLC| opt '(|from|) nil))
(cond
  ((eq opt1 '|from|) nil)
  ((eq opt '|c|) (|/D,1| newopt (/COMP) nil nil))
  ((eq opt '|d|) (|/D,1| newopt 'define nil nil))
  ((eq opt '|dt|) (|/D,1| newopt 'define nil t))
  ((eq opt '|ct|) (|/D,1| newopt (/COMP) nil t))
  ((eq opt '|ctl|) (|/D,1| newopt (/COMP) nil 'tracelet))
  ((eq opt '|ec|) (|/D,1| newopt (/COMP) t nil))
  ((eq opt '|ect|) (|/D,1| newopt (/COMP) t t))
  ((eq opt '|e|) (|/D,1| newopt nil t nil))
  ((eq opt '|version|) (|version|))
  ((eq opt '|pause|)
   (setq constream
    (defiostream '((device . console) (qual . v)) 120 0))
   (next constream)
   (shut constream))
  ((or
    (eq opt '|update|)
    (eq opt '|patch|))
   (setq |$InteractiveMode| nil)
   (setq upf
    (cons
      (or (kar optargs) /version)
      (cons
        (or (kadr optargs) /wsname)
        (cons (or (kaddr optargs) '* ) nil))))))
   (setq fun
    (cond
      ((eq opt '|patch|) '/update-lib-1)
      (t '/update-1)))
   (catch 'filenam (funcall fun upf))
   (|sayMessage| " Update/patch is completed."))
  ((null optargs)
   (|sayBrightly| '(" An argument is required for" ,@( |bright| opt))))
  (t
   (|sayMessage|
    '(" Unknown option:" ,@( |bright| opt)
      |%1| " Available options are"
      ,@( |bright|
        "c ct e ec ect cls pause update patch compare record"))))))))

```

Chapter 56

Handling input files

56.0.4 defun Handle .axiom.input file

[/editfile p493]

— defun readSpadProfileIfThere —

```
(defun |readSpadProfileIfThere| ()  
  (let ((file (list '|.axiom| '|input|)))  
    (declare (special /editfile))  
    (when (makeInputFilename file) (setq /editfile file) (/rq))))
```

—————

56.0.5 defun /rq

[/rq /rf-1 (vol9)]

[echo-meta p923]

— defun /rq —

```
(defun /RQ (&rest foo &aux (echo-meta nil))  
  (declare (special Echo-Meta) (ignore foo))  
  (/rf-1 nil))
```

—————

56.0.6 defun /rf

Compile with noisy output [/rf /rf-1 (vol9)]
 [echo-meta p923]

— defun /rf —

```
(defun /rf (&rest foo &aux (echo-meta t))
  (declare (special echo-meta) (ignore foo))
  (/rf-1 nil))
```

—————

56.0.7 defvar \$boot-line-stack

— initvars —

```
(defvar boot-line-stack nil "List of lines returned from preparse")
```

—————

56.0.8 defvar \$in-stream

— initvars —

```
(defvar in-stream t "Current input stream.")
```

—————

56.0.9 defvar \$out-stream

— initvars —

```
(defvar out-stream t "Current output stream.")
```

—————

56.0.10 defvar \$file-closed

— initvars —

```
(defvar file-closed nil "Way to stop EOF tests for console input.")
```

—————

56.0.11 defvar \$echo-meta

— initvars —

```
(defvar echo-meta nil "T if you want a listing of what has been read.")
```

—————

56.0.12 defvar \$noSubsumption

— initvars —

```
(defvar |$noSubsumption| t)
```

—————

56.0.13 defvar \$envHashTable

The `$envHashTable` variable is a hashtable that optimizes lookups in the environment, which normally involve search. This gets populated in the `addBinding` function.

— initvars —

```
(defvar |$envHashTable| nil)
```

—————

56.0.14 defun Dynamically add bindings to the environment

[getProplist p924]

[addBindingInteractive p927]

```
[hput p1000]
[$InteractiveMode p24]
[$envHashTable p923]
```

— **defun addBinding** —

```
(defun |addBinding| (var proplist e)
  (let (tailContour tailEnv tmp1 curContour lx)
    (declare (special |$InteractiveMode| |$envHashTable|))
    (setq curContour (caar e))
    (setq tailContour (cdar e))
    (setq tailEnv (cdr e))
    (cond
      ((eq proplist (|getProplist| var e)) e)
      (t
       (when |$envHashTable|
         (do ((prop proplist (cdr prop)) (u nil))
             ((or (atom prop)
                  (progn (setq u (car prop)) nil))
              nil)
          (hput |$envHashTable| (list var (car u)) t)))
        (cond
          (|$InteractiveMode| (|addBindingInteractive| var proplist e))
          (t
           (when (and (pairp curContour)
                      (progn
                        (setq tmp1 (qcar curContour))
                        (and (pairp tmp1) (equal (qcar tmp1) var))))
             (setq curContour (cdr curContour)))
            (setq lx (cons var proplist))
            (cons (cons (cons lx curContour) tailContour) tailEnv)))))))
```

—————

56.0.15 defun Fetch a property list for a symbol from CategoryFrame

```
[getProplist p924]
[search p925]
[$CategoryFrame p??]
```

— **defun getProplist** —

```
(defun |getProplist| (x e)
  (let (u pl)
    (declare (special |$CategoryFrame|))
    (cond
      ((null (atom x)) (|getProplist| (car x) e))
```

```
((setq u (|search| x e)) u)
((setq pl (|search| x |$CategoryFrame|) pl))))
```

56.0.16 defun Search for a binding in the environment list

```
[searchCurrentEnv p925]
[searchTailEnv p926]
```

— defun search —

```
(defun |search| (x e)
  (let ((curEnv (car e)) (tailEnv (cdr e)))
    (or (|searchCurrentEnv| x curEnv) (|searchTailEnv| x tailEnv))))
```

56.0.17 defun Search for a binding in the current environment

```
searchCurrentEnv(x,currentEnv) ==
  for contour in currentEnv repeat
    if u:= ASSQ(x,contour) then return (signal:= u)
  KDR signal
```

```
[assq p1006]
[kdr p??]
```

— defun searchCurrentEnv —

```
(defun |searchCurrentEnv| (x currentEnv)
  (prog (u signal)
    (return
      (seq
        (progn
          (do ((thisenv currentEnv (cdr thisenv)) (contour nil))
              ((or (atom thisenv) (progn (setq contour (car thisenv)) nil)) nil)
            (seq
              (exit
                (cond
                  ((setq u (assq x contour)) (return (setq signal u)))
                  (t nil))))))
          (kdr signal))))))
```

56.0.18 defun searchTailEnv

```

;searchTailEnv(x,e) ==
;  for env in e repeat
;    signal:=
;      for contour in env repeat
;        if (u:= ASSQ(x,contour)) and ASSQ("FLUID",u) then return (signal:= u)
;      if signal then return signal
;  KDR signal

```

```

[assq p1006]
[kdr p??]

```

— defun searchTailEnv —

```

(defun |searchTailEnv| (x e)
  (prog (u signal)
    (return
      (seq
        (progn
          (do ((thise e (cdr thise)) (env nil))
            ((or (atom thise) (progn (setq env (car thise)) nil)) nil)
          (seq
            (exit
              (setq signal
                (progn
                  (do ((cone env (cdr cone)) (contour nil))
                    ((or (atom cone) (progn (setq contour (car cone)) nil)) nil)
                  (seq
                    (exit
                      (cond
                        ((and (setq u (assq x contour)) (assq 'fluid u))
                          (return (setq signal u)))
                        (t nil))))))
                (t nil))))))
            (cond
              (signal (return signal))
              (t nil))))))
          (kdr signal))))))

```

Chapter 57

File Parsing

57.0.19 defun Bind a variable in the interactive environment

[assq p1006]

— defun addBindingInteractive —

```
(defun |addBindingInteractive| (var proplist e)
  (let ((curContour (caar e)) u)
    (cond
      ((setq u (assq var curContour)) (rplacd u proplist) e)
      (t (rplac (caar e) (cons (cons var proplist) curContour)) e))))
```

—————

57.0.20 defvar \$line-handler

— initvars —

```
(defparameter line-handler 'next-META-line "Who grabs lines for us.")
```

—————

57.0.21 defvar \$spad-errors

— initvars —


```
(defvar $spad_errors (vector 0 0 0))
```

57.0.22 defvar \$xtokenreader

— initvars —

```
(defvar xtokenreader 'spadtok)
```

57.0.23 defun Initialize the spad reader

```
[next-lines-clear p929]
[ioclear p929]
[$spad-errors p927]
[spaderrorstream p??]
[*standard-output* p??]
[xtokenreader p928]
[line-handler p927]
[meta-error-handler p??]
[file-closed p923]
[boot-line-stack p922]
```

— defun init-boot/spad-reader —

```
(defun init-boot/spad-reader ()
  (declare (special $spad_errors spaderrorstream *standard-output*
                  xtokenreader line-handler meta-error-handler file-closed
                  boot-line-stack))
  (setq $spad_errors (vector 0 0 0))
  (setq spaderrorstream *standard-output*)
  (setq xtokenreader 'get-BOOT-token)
  (setq line-Handler 'next-BOOT-line)
  (setq meta-error-handler 'spad-syntax-error)
  (setq file-closed nil)
  (next-lines-clear)
  (ioclear))
```

57.0.24 defun ioclear

The IO state manipulation routines assume that

- one I/O stream pair is in effect at any moment
- there is a current line
- there is a current token and a next token
- there is a reduction stack

```
[line-clear p??]
[token-stack-clear p??]
[reduce-stack-clear p??]
[current-fragment p??]
[current-line p??]
[$boot p25]
[$spad p20]
```

— defun ioclear —

```
(defun ioclear (&optional (in t) (out t))
  (declare (special current-fragment current-line $boot $spad))
  (setq current-fragment nil)
  (line-clear current-line)
  (token-stack-clear)
  (reduce-stack-clear)
  (if (or $boot $spad) (next-lines-clear))
  nil)
```

—————

57.0.25 defun Set boot-line-stack to nil

```
[boot-line-stack p922]
```

— defun next-lines-clear —

```
(defun next-lines-clear ()
  (setq boot-line-stack nil))
```

—————

Chapter 58

Handling output

58.1 Special Character Tables

58.1.1 `defvar $defaultSpecialCharacters`

— initvars —

```
(defvar |$defaultSpecialCharacters| (list
  (int-char 28)    ; upper left corner
  (int-char 27)    ; upper right corner
  (int-char 30)    ; lower left corner
  (int-char 31)    ; lower right corner
  (int-char 79)    ; vertical bar
  (int-char 45)    ; horizontal bar
  (int-char 144)   ; APL quad
  (int-char 173)   ; left bracket
  (int-char 189)   ; right bracket
  (int-char 192)   ; left brace
  (int-char 208)   ; right brace
  (int-char 59)    ; top    box tee
  (int-char 62)    ; bottom box tee
  (int-char 63)    ; right  box tee
  (int-char 61)    ; left   box tee
  (int-char 44)    ; center box tee
  (int-char 224))) ; back slash
```

—————

58.1.2 defvar \$plainSpecialCharacters0

— initvars —

```
(defvar |$plainSpecialCharacters0| (list
  (int-char 78)      ; upper left corner  (+)
  (int-char 78)      ; upper right corner (+)
  (int-char 78)      ; lower left corner  (+)
  (int-char 78)      ; lower right corner (+)
  (int-char 79)      ; vertical bar
  (int-char 96)      ; horizontal bar      (-)
  (int-char 111)     ; APL quad            (?)
  (int-char 173)     ; left bracket
  (int-char 189)     ; right bracket
  (int-char 192)     ; left brace
  (int-char 208)     ; right brace
  (int-char 78)      ; top    box tee      (+)
  (int-char 78)      ; bottom box tee      (+)
  (int-char 78)      ; right  box tee      (+)
  (int-char 78)      ; left   box tee      (+)
  (int-char 78)      ; center box tee      (+)
  (int-char 224)))   ; back slash
```

—————

58.1.3 defvar \$plainSpecialCharacters1

— initvars —

```
(defvar |$plainSpecialCharacters1| (list
  (int-char 107)     ; upper left corner  (,)
  (int-char 107)     ; upper right corner (,)
  (int-char 125)     ; lower left corner  (')
  (int-char 125)     ; lower right corner (')
  (int-char 79)      ; vertical bar
  (int-char 96)      ; horizontal bar      (-)
  (int-char 111)     ; APL quad            (?)
  (int-char 173)     ; left bracket
  (int-char 189)     ; right bracket
  (int-char 192)     ; left brace
  (int-char 208)     ; right brace
  (int-char 78)      ; top    box tee      (+)
  (int-char 78)      ; bottom box tee      (+)
  (int-char 78)      ; right  box tee      (+)
  (int-char 78)      ; left   box tee      (+)
```

```
(int-char 78)      ; center box tee      (+)
(int-char 224)))   ; back slash
```

58.1.4 defvar \$plainSpecialCharacters2

— initvars —

```
(defvar |$plainSpecialCharacters2| (list
  (int-char 79)      ; upper left corner  (|)
  (int-char 79)      ; upper right corner (|)
  (int-char 79)      ; lower left corner  (|)
  (int-char 79)      ; lower right corner (|)
  (int-char 79)      ; vertical bar
  (int-char 96)      ; horizontal bar      (-)
  (int-char 111)     ; APL quad            (?)
  (int-char 173)     ; left bracket
  (int-char 189)     ; right bracket
  (int-char 192)     ; left brace
  (int-char 208)     ; right brace
  (int-char 78)      ; top    box tee      (+)
  (int-char 78)      ; bottom box tee     (+)
  (int-char 78)      ; right  box tee     (+)
  (int-char 78)      ; left   box tee     (+)
  (int-char 78)      ; center box tee     (+)
  (int-char 224)))   ; back slash
```

58.1.5 defvar \$plainSpecialCharacters3

— initvars —

```
(defvar |$plainSpecialCharacters3| (list
  (int-char 96)      ; upper left corner  (-)
  (int-char 96)      ; upper right corner (-)
  (int-char 96)      ; lower left corner  (-)
  (int-char 96)      ; lower right corner (-)
  (int-char 79)      ; vertical bar
  (int-char 96)      ; horizontal bar      (-)
  (int-char 111)     ; APL quad            (?)
  (int-char 173)     ; left bracket
```

```

(int-char 189)    ; right bracket
(int-char 192)    ; left brace
(int-char 208)    ; right brace
(int-char 78)     ; top    box tee      (+)
(int-char 78)     ; bottom box tee      (+)
(int-char 78)     ; right  box tee      (+)
(int-char 78)     ; left   box tee      (+)
(int-char 78)     ; center box tee      (+)
(int-char 224)))  ; back slash

```

58.1.6 defvar \$plainRTspecialCharacters

— initvars —

```

(defvar |$plainRTspecialCharacters| (list
  (QUOTE +)      ; upper left corner  (+)
  (QUOTE +)      ; upper right corner (+)
  (QUOTE +)      ; lower left corner  (+)
  (QUOTE +)      ; lower right corner (+)
  (QUOTE |\\|)   ; vertical bar
  (QUOTE -)      ; horizontal bar      (-)
  (QUOTE ?)      ; APL quad            (?)
  (QUOTE [)      ; left bracket
  (QUOTE ])      ; right bracket
  (QUOTE {)      ; left brace
  (QUOTE })      ; right brace
  (QUOTE +)      ; top    box tee      (+)
  (QUOTE +)      ; bottom box tee      (+)
  (QUOTE +)      ; right  box tee      (+)
  (QUOTE +)      ; left   box tee      (+)
  (QUOTE +)      ; center box tee      (+)
  (QUOTE |\\|))) ; back slash

```

58.1.7 defvar \$RTspecialCharacters

— initvars —

```

(defvar |$RTspecialCharacters| (list
  (intern (string (code-char 218))) ;-- upper left corner  (+)

```

```

(intern (string (code-char 191))) ;-- upper right corner (+)
(intern (string (code-char 192))) ;-- lower left corner (+)
(intern (string (code-char 217))) ;-- lower right corner (+)
(intern (string (code-char 179))) ;-- vertical bar
(intern (string (code-char 196))) ;-- horizontal bar      (-)
(list (code-char #x1d) (code-char #xe2))
                                ;-- APL quad            (?)
(QUOTE [])                      ;-- left bracket
(QUOTE ])                      ;-- right bracket
(QUOTE {)                      ;-- left brace
(QUOTE })                      ;-- right brace
(intern (string (code-char 194))) ;-- top box tee        (+)
(intern (string (code-char 193))) ;-- bottom box tee     (+)
(intern (string (code-char 180))) ;-- right box tee       (+)
(intern (string (code-char 195))) ;-- left box tee        (+)
(intern (string (code-char 197))) ;-- center box tee      (+)
(QUOTE |\\|))                  ;-- back slash

```

58.1.8 defvar \$specialCharacters

— initvars —

```
(defvar |$specialCharacters| |$RTspecialCharacters|)
```

58.1.9 defvar \$specialCharacterAlist

— initvars —

```

(defvar |$specialCharacterAlist|
  '([ulc| . 0)
    (|urc| . 1)
    (|llc| . 2)
    (|lrc| . 3)
    (|vbar| . 4)
    (|hbar| . 5)
    (|quad| . 6)
    (|lbrk| . 7)
    (|rbrk| . 8)
    (|lbrc| . 9)

```



```
(|rbrcl| . 10)
(|ttee| . 11)
(|btee| . 12)
(|rtee| . 13)
(|ltee| . 14)
(|ctee| . 15)
(|bslash| . 16)))
```

58.1.10 defun Look up a special character code for a symbol

This function looks up a symbol in `$specialCharacterAlist`, gets the index into the EBCDIC table, and returns the appropriate character. **TPDHERE: Make this more international, not EBCDIC** [ifcdr p??]

```
[assq p1006]
[$specialCharacters p935]
[$specialCharacterAlist p935]
```

— defun specialChar —

```
(defun |specialChar| (symbol)
  (let (code)
    (declare (special |$specialCharacters| |$specialCharacterAlist|))
    (if (setq code (ifcdr (assq symbol |$specialCharacterAlist|)))
        (elt |$specialCharacters| code)
        "?")))
```

Chapter 59

Stream and File Handling

59.0.11 defun make-instream

[makeInputFilename p939]

— defun make-instream —

```
(defun make-instream (filespec &optional (recnum 0))
  (declare (ignore recnum))
  (cond ((numberp filespec) (make-synonym-stream '*terminal-io*))
        ((null filespec) (error "not handled yet"))
        (t (open (makeInputFilename filespec)
                  :direction :input :if-does-not-exist nil))))
```

—————

59.0.12 defun make-outstream

[make-filename p??]

— defun make-outstream —

```
(defun make-outstream (filespec &optional (width nil) (recnum 0))
  (declare (ignore width) (ignore recnum))
  (cond ((numberp filespec) (make-synonym-stream '*terminal-io*))
        ((null filespec) (error "not handled yet"))
        (t (open (make-filename filespec) :direction :output))))
```

—————

59.0.13 defun make-appendstream

[make-filename p??]

— defun make-appendstream —

```
(defun make-appendstream (filespec &optional (width nil) (recnum 0))
  "fortran support"
  (declare (ignore width) (ignore recnum))
  (cond
   ((numberp filespec) (make-synonym-stream '*terminal-io*))
   ((null filespec) (error "make-appendstream: not handled yet"))
   ('else (open (make-filename filespec) :direction :output
                 :if-exists :append :if-does-not-exist :create))))
```

59.0.14 defun defiostream

— defun defiostream —

```
(defun defiostream (stream-alist buffer-size char-position)
  (declare (ignore buffer-size))
  (let ((mode (or (cdr (assoc 'mode stream-alist)) 'input))
        (filename (cdr (assoc 'file stream-alist)))
        (dev (cdr (assoc 'device stream-alist))))
    (if (eq dev 'console) (make-synonym-stream '*terminal-io*)
        (let ((strm (case mode
                      ((output o) (open (make-filename filename)
                                           :direction :output))
                      ((input i) (open (makeInputFilename filename)
                                           :direction :input)))))
          (if (and (numberp char-position) (> char-position 0))
              (file-position strm char-position)
              strm))))
```

59.0.15 defun shut

[shut is-console (vol9)]

— defun shut —

```
(defun shut (st)
  (if (is-console st)
      st
      (if (streamp st) (close st) -1))))
```

59.0.16 defun eofp

— defun eofp —

```
(defun eofp (stream) (null (peek-char nil stream nil nil)))
```

59.0.17 defun makeStream

```
[make-appendstream p938]
[make-outstream p937]
```

— defun makeStream —

```
(defun |makeStream| (append filename i j)
  (if append
      (make-appendstream filename i j)
      (make-outstream filename i j)))
```

59.0.18 defun Construct a new input file name

— defun makeInputFilename —

```
(defun makeInputFilename (filearg &optional (filetype nil))
  (let*
    ((filename (make-filename filearg filetype))
     (dirname (pathname-directory filename))
     (ft (pathname-type filename))
     (dirs (getDirectoryList ft))
     (newfn nil))
    (if (or (null dirname) (eqcar dirname :relative))
```

```
(dolist (dir dirs (probeName filename))
  (when (probe-file (setq newfn (concatenate 'string dir filename)))
    (return newfn)))
(probeName filename))))
```

59.0.19 defun getDirectoryList

```
[$current-directory p7]
[$UserLevel p784]
[$library-directory-list p9]
[$directory-list p8]
```

— defun getDirectoryList —

```
(defun getDirectoryList (ft &aux (cd (namestring $current-directory)))
  (declare (special $current-directory |$UserLevel| $library-directory-list
                    $directory-list))
  (if (member ft '("nrlib" "daase" "exposed")) :test #'string=)
  (if (eq |$UserLevel| '|development|)
    (cons cd $library-directory-list)
    $library-directory-list)
  (adjoin cd
    (adjoin (namestring (user-homedir-pathname)) $directory-list
      :test #'string=)
    :test #'string=)))
```

59.0.20 defun probeName

Sometimes we are given a file and sometimes we are given the name of an Axiom KAF (Keyed-Access File). KAF files are actually directories with a single file called “index.kaf”. We check for the latter case and return the directory name as the filename, per Axiom convention.

— defun probeName —

```
(defun probeName (file)
  (when (or (probe-file file)
    (probe-file (concatenate 'string (namestring file) "/index.kaf")))
    (namestring file)))
```

59.0.21 defun makeFullNamestring

— defun makeFullNamestring —

```
(defun makeFullNamestring (filearg &optional (filetype nil))
  (namestring (merge-pathnames (make-filename filearg filetype))))
```

—————

59.0.22 defun Replace a file by erase and rename

[makeFullNamestring p941]

— defun replaceFile —

```
(defun replaceFile (filespec1 filespec2)
  ($erase (setq filespec1 (makeFullNamestring filespec1)))
  (rename-file (makeFullNamestring filespec2) filespec1))
```

—————

Chapter 60

The Spad Server Mechanism

60.0.23 defun openserver

This is a cover function for the C code used for communication interface.

— **defun openserver** —

```
(defun openserver (name)
  (open_server name))
```

—————

Chapter 61

Axiom Build-time Functions

61.0.24 defun spad-save

The **spad-save** function is just a cover function for more lisp system specific save functions. There is no standard name for saving a lisp image so we make one and conditionalize it at compile time.

This function is passed the name of an image that will be saved. The saved image contains all of the loaded functions.

This is used in the `src/interp/Makefile.pamphlet` in three places:

- creating `depsys`, an image for compiling axiom.

Some of the Common Lisp code we compile uses macros which are assumed to be available at compile time. The **DEPSYS** image is created to contain the compile time environment and saved. We pipe compile commands into this environment to compile from Common Lisp to machine dependent code.

```
DEPSYS=${OBJ}/${SYS}/bin/depsys
```

- creating `savesys`, an image for running axiom.

Once we've compile all of the Common Lisp files we fire up a clean lisp image called **LOADSYS**, load all of the final executable code and save it out as **SAVESYS**. The **SAVESYS** image is copied to the `${MNT}/${SYS}/bin` subdirectory and becomes the axiom executable image.

```
LOADSYS= ${OBJ}/${SYS}/bin/lisp
SAVESYS= ${OBJ}/${SYS}/bin/interpsys
AXIOMSYS= ${MNT}/${SYS}/bin/AXIOMsys
```

- creating `debugsys`, an image with all interpreted functions loaded.

Occasionally we need to really get into the system internals. The best way to do this is to run almost all of the lisp code interpreted rather than compiled (note that `cfuns.lisp` and `sockio.lisp` still need to be loaded in compiled form as they depend on the loader to link with lisp internals). This image is nothing more than a load of the file `src/interp/debugsys.lisp.pamphlet`. If you need to make test modifications you can add code to that file and it will show up here.

```
DEBUGSYS=${OBJ}/${SYS}/bin/debugsys
```

```
[save-system p??]  
[$SpadServer p12]  
[$openServerIfTrue p10]
```

— defun spad-save —

```
(defun user::spad-save (save-file)
  (declare (special |$SpadServer| $openServerIfTrue))
  (setq |$SpadServer| nil)
  (setq $openServerIfTrue t)
#+:AKCL
  (system::save-system save-file)
#+:allegro
  (if (fboundp 'boot::restart)
      (excl::dumplisp :name save-file :restart-function #'boot::restart)
      (excl::dumplisp :name save-file))
#+:Lucid
  (if (fboundp 'boot::restart)
      (sys::disksave save-file :restart-function #'boot::restart)
      (sys::disksave save-file))
#+:CCL
  (preserve)
)
```

—————

Chapter 62

Exposure Groups

Exposure groups are a way of controlling the namespace available to the user. Certain algebra files are only useful for internal purposes but they contain functions have common names (like “map”). In order to separate the user visible functions from the internal functions the algebra files are collected into “exposure groups”. These large groups are grouped into sets in the variable `$globalExposureGroupAlist`.

Exposure group information is kept in the local frame. For more information “The Frame Mechanism” 32.3.1 on page 530.

Chapter 63

Databases

63.1 Database structure

In order to understand this program you need to understand some details of the structure of the databases it reads. Axiom has 5 databases, the `interp.daase`, `operation.daase`, `category.daase`, `compress.daase`, and `browse.daase`. The `compress.daase` is special and does not follow the normal database format.

63.1.1 kaf File Format

This documentation refers to `kaf` files which are random access files. `nrllib` files are `kaf` files (look for `nrllib/index.kaf`) The format of a random access file is

```
byte-offset-of-key-table
first-entry
second-entry
...
last-entry
((key1 . first-entry-byte-address)
 (key2 . second-entry-byte-address)
 ...
 (keyN . last-entry-byte-address))
```

The key table is a standard lisp alist.

To open a database you fetch the first number, seek to that location, and `(read)` which returns the key-data alist. To look up data you index into the key-data alist, find the `ith-entry-byte-address`, seek to that address, and `(read)`.

For instance, see `src/share/algebra/users.daase/index.kaf`

One existing optimization is that if the data is a simple thing like a symbol then the `nth-entry-byte-address` is replaced by immediate data.

Another existing one is a compression algorithm applied to the data so that the very long names don't take up so much space. We could probably remove the compression algorithm as 64k is no longer considered 'huge'. The database-abbreviation routine handles this on read and write-compress handles this on write. The squeeze routine is used to compress the keys, the unsqueeze routine uncompresses them. Making these two routines disappear should remove all of the compression.

Indeed, a faster optimization is to simply read the whole database into the image before it is saved. The system would be easier to understand and the interpreter would be faster.

The fastest optimization is to fix the time stamp mechanism which is currently broken. Making this work requires a small bit of coordination at 'make' time which I forgot to implement.

63.1.2 Database Files

Database files are very similar to kaf files except that there is an optimization (currently broken) which makes the first item a pair of two numbers. The first number in the pair is the offset of the key-value table, the second is a time stamp. If the time stamp in the database matches the time stamp in the image the database is not needed (since the internal hash tables already contain all of the information). When the database is built the time stamp is saved in both the gcl image and the database.

Regarding the 'ancestors field for a category: At database build time there exists a `*ancestors-hash*` hash table that gets filled with CATEGORY (not domain) ancestor information. This later provides the information that goes into `interp.daase`. This `*ancestors-hash*` does not exist at normal runtime (it can be made by a call to `genCategoryTable`). Note that the ancestor information in `*ancestors-hash*` (and hence `interp.daase`) involves `#1`, `#2`, etc instead of `R`, `Coef`, etc. The latter thingies appear in all `.nrlib/index.kaf` files. So we need to be careful when we `)lib` categories and update the ancestor info.

This file contains the code to build, open and access the `.daase` files. This file contains the code to `)library` `nrlibs` and `asy` files

There is a major issue about the data that resides in these databases. the fundamental problem is that the system requires more information to build the databases than it needs to run the interpreter. in particular, `modemap.daase` is constructed using properties like "modemaps" but the interpreter will never ask for this information.

So, the design is as follows:

- the `modemap.daase` needs to be built. this is done by doing a `)library` on ALL of the `nrlib` files that are going into the system. this will bring in "modemap" information and add it to the `*modemaps-hash*` hashtable.
- database build proceeds, accessing the "modemap" property from the hashtables. once this completes this information is never used again.
- the `interp.daase` database is built. this contains only the information necessary to run the interpreter. note that during the running of the interpreter users can extend the

system by do a)library on a new nrlib file. this will cause fields such as "modemap" to be read and hashed.

Each constructor (e.g. LIST) had one library directory (e.g. LIST.nrlib). This directory contained a random access file called the index.kaf file. These files contain runtime information such as the operationAlist and the ConstructorModemap. At system build time we merge all of these .nrlib/index.kaf files into one database, INTERP.daase. Requests to get information from this database are cached so that multiple references do not cause additional disk i/o.

This database is left open at all times as it is used frequently by the interpreter. one minor complication is that newly compiled files need to override information that exists in this database.

The design calls for constructing a random read (kaf format) file that is accessed by functions that cache their results. when the database is opened the list of constructor-index pairs is hashed by constructor name. a request for information about a constructor causes the information to replace the index in the hash table. since the index is a number and the data is a non-numeric sexpr there is no source of confusion about when the data needs to be read.

The format of this new database is as follows:

```
first entry:
  an integer giving the byte offset to the constructor alist
  at the bottom of the file
second and subsequent entries (one per constructor)
  (operationAlist)
  (constructorModemap)
....
last entry: (pointed at by the first entry)
  an alist of (constructor . index) e.g.
    ( (PI offset-of-operationAlist offset-of-constructorModemap)
      (NMI offset-of-operationAlist offset-of-constructorModemap)
      ....)
This list is read at open time and hashed by the car of each item.
```

The system has been changed to use the property list of the symbols rather than hash tables. since we already hashed once to get the symbol we need only an offset to get the property list. this also has the advantage that eq hash tables no longer need to be moved during garbage collection.

There are 3 potential speedups that could be done.

- the best would be to use the value cell of the symbol rather than the property list but i'm unable to determine all uses of the value cell at the present time.
- a second speedup is to guarantee that the property list is a single item, namely the database structure. this removes an assoc but leaves one open to breaking the system if someone adds something to the property list. this was not done because of the danger mentioned.

- a third speedup is to make the `getdatabase` call go away, either by making it a macro or eliding it entirely. this was not done because we want to keep the flexibility of changing the database forms.

The new design does not use hash tables. the database structure contains an entry for each item that used to be in a hash table. initially the structure contains file-position pointers and these are replaced by real data when they are first looked up. the database structure is kept on the property list of the constructor, thus, `(get '—DenavitHartenbergMatrix— 'database)` will return the database structure object.

Each operation has a property on its symbol name called `'operation` which is a list of all of the signatures of operations with that name.

63.1.3 `defstruct $database`

— initvars —

```
(defstruct database
  abbreviation      ; interp.
  ancestors         ; interp.
  constructor       ; interp.
  constructorcategory ; interp.
  constructorkind   ; interp.
  constructormodemap ; interp.
  cosig            ; interp.
  defaultdomain    ; interp.
  modemaps         ; interp.
  niladic          ; interp.
  object           ; interp.
  operationalist    ; interp.
  documentation     ; browse.
  constructorform   ; browse.
  attributes        ; browse.
  predicates        ; browse.
  sourcefile        ; browse.
  parents           ; browse.
  users             ; browse.
  dependents        ; browse.
  spare             ; superstition
) ; database structure
```

—

63.1.4 defvar \$*defaultdomain-list*

There are only a small number of domains that have default domains. rather than keep this slot in every domain we maintain a list here.

— initvars —

```
(defvar *defaultdomain-list* '(
  (|MultisetAggregate| |Multiset|)
  (|FunctionSpace| |Expression|)
  (|AlgebraicallyClosedFunctionSpace| |Expression|)
  (|ThreeSpaceCategory| |ThreeSpace|)
  (|DequeueAggregate| |Dequeue|)
  (|ComplexCategory| |Complex|)
  (|LazyStreamAggregate| |Stream|)
  (|AssociationListAggregate| |AssociationList|)
  (|QuaternionCategory| |Quaternion|)
  (|PriorityQueueAggregate| |Heap|)
  (|PointCategory| |Point|)
  (|PlottableSpaceCurveCategory| |Plot3D|)
  (|PermutationCategory| |Permutation|)
  (|StringCategory| |String|)
  (|FileNameCategory| |FileName|)
  (|OctonionCategory| |Octonion|)))
```

—————

63.1.5 defvar \$*operation-hash*

— initvars —

```
(defvar *operation-hash* nil "given an operation name, what are its modemaps?")
```

—————

63.1.6 defvar \$*hasCategory-hash*

This hash table is used to answer the question“does domain x have category y?”. this is answered by constructing a pair of (x . y) and doing an equal hash into this table.

— initvars —

```
(defvar *hasCategory-hash* nil "answers x has y category questions")
```

—————

63.1.7 defvar \$*miss*

This variable is used for debugging. If a hash table lookup fails and this variable is non-nil then a message is printed.

— initvars —

```
(defvar *miss* nil "print out cache misses on getdatabase calls")
```

Note that constructorcategory information need only be kept for items of type category. this will be fixed in the next iteration when the need for the various caches are reviewed

Note that the *modemaps-hash* information does not need to be kept for system files. these are precomputed and kept in modemap.daase however, for user-defined files these are needed. Currently these are added to the database for 2 reasons; there is a still-unresolved issue of user database extensions and this information is used during database build time

63.1.8 Database streams

This are the streams for the databases. They are always open. There is an optimization for speeding up system startup. If the database is opened and the ..-stream-stamp* variable matches the position information in the database then the database is NOT read in and is assumed to match the in-core version

63.1.9 defvar \$*compressvector*

— initvars —

```
(defvar *compressvector* nil "a vector of things to compress in the databases")
```

63.1.10 defvar \$*compressVectorLength*

— initvars —

```
(defvar *compressVectorLength* 0 "length of the compress vector")
```

63.1.11 defvar \$*compress-stream*

— initvars —

```
(defvar *compress-stream* nil "an stream containing the compress vector")
```

—————

63.1.12 defvar \$*compress-stream-stamp*

— initvars —

```
(defvar *compress-stream-stamp* 0 "*compress-stream* (position . time)")
```

—————

63.1.13 defvar \$*interp-stream*

— initvars —

```
(defvar *interp-stream* nil "an open stream to the interpreter database")
```

—————

63.1.14 defvar \$*interp-stream-stamp*

— initvars —

```
(defvar *interp-stream-stamp* 0 "*interp-stream* (position . time)")
```

—————

63.1.15 defvar \$*operation-stream*

This is indexed by operation, not constructor

— initvars —

```
(defvar *operation-stream* nil "the stream to operation.daase")
```

63.1.16 defvar \$*operation-stream-stamp*

— initvars —

```
(defvar *operation-stream-stamp* 0 "*operation-stream* (position . time)")
```

63.1.17 defvar \$*browse-stream*

— initvars —

```
(defvar *browse-stream* nil "an open stream to the browser database")
```

63.1.18 defvar \$*browse-stream-stamp*

— initvars —

```
(defvar *browse-stream-stamp* 0 "*browse-stream* (position . time)")
```

63.1.19 defvar \$*category-stream*

This is indexed by (domain . category)

— initvars —

```
(defvar *category-stream* nil "an open stream to the category table")
```

63.1.20 defvar \$*category-stream-stamp*

— initvars —

```
(defvar *category-stream-stamp* 0 "category-stream* (position . time)")
```

—————

63.1.21 defvar \$*allconstructors*

— initvars —

```
(defvar *allconstructors* nil "a list of all the constructors in the system")
```

—————

63.1.22 defvar \$*allOperations*

— initvars —

```
(defvar *allOperations* nil "a list of all the operations in the system")
```

—————

63.1.23 defun Reset all hash tables before saving system

```
[compressopen p??]
[interpopen p??]
[operationopen p??]
[browseopen p??]
[categoryopen p??]
[initial-getdatabase p958]
[*sourcefiles* p??]
[*interp-stream* p955]
[*operation-stream* p955]
[*category-stream* p956]
[*browse-stream* p956]
[*category-stream-stamp* p957]
[*operation-stream-stamp* p956]
```

```

[*interp-stream-stamp* p955]
[*compress-stream-stamp* p955]
[*compressvector* p954]
[*allconstructors* p957]
[*operation-hash* p953]
[*hascategory-hash* p??]

```

— defun resethashtables —

```

(defun resethashtables ()
  "set all -hash* to clean values. used to clean up core before saving system"
  (declare (special *sourcefiles* *interp-stream* *operation-stream*
                    *category-stream* *browse-stream* *category-stream-stamp*
                    *operation-stream-stamp* *interp-stream-stamp*
                    *compress-stream-stamp* *compressvector*
                    *allconstructors* *operation-hash* *hascategory-hash*))
  (setq *hascategory-hash* (make-hash-table :test #'equal))
  (setq *operation-hash* (make-hash-table))
  (setq *allconstructors* nil)
  (setq *compressvector* nil)
  (setq *sourcefiles* nil)
  (setq *compress-stream-stamp* '(0 . 0))
  (compressopen)
  (setq *interp-stream-stamp* '(0 . 0))
  (interpopen)
  (setq *operation-stream-stamp* '(0 . 0))
  (operationopen)
  (setq *browse-stream-stamp* '(0 . 0))
  (browseopen)
  (setq *category-stream-stamp* '(0 . 0))
  (categoryopen) ;note: this depends on constructorform in browse.daase
  (initial-getdatabase)
  (close *interp-stream*)
  (close *operation-stream*)
  (close *category-stream*)
  (close *browse-stream*)
  (gbc t))

```

— — —

63.1.24 defun Preload algebra into saved system

```

[getdatabase p967]
[getEnv p??]

```

— defun initial-getdatabase —

```

(defun initial-getdatabase ()
  "fetch data we want in the saved system"
  (let (hascategory constructormodemapAndoperationalist operation constr)
    (format t "Initial getdatabase~%")
    (setq hascategory '(
      (|Equation| . |Ring|)
      (|Expression| . |CoercibleTo|) (|Expression| . |CommutativeRing|)
      (|Expression| . |IntegralDomain|) (|Expression| . |Ring|)
      (|Float| . |RetractableTo|)
      (|Fraction| . |Algebra|) (|Fraction| . |CoercibleTo|)
      (|Fraction| . |OrderedSet|) (|Fraction| . |RetractableTo|)
      (|Integer| . |Algebra|) (|Integer| . |CoercibleTo|)
      (|Integer| . |ConvertibleTo|) (|Integer| . |LinearlyExplicitRingOver|)
      (|Integer| . |RetractableTo|)
      (|List| . |CoercibleTo|) (|List| . |FiniteLinearAggregate|)
      (|List| . |OrderedSet|)
      (|Polynomial| . |CoercibleTo|) (|Polynomial| . |CommutativeRing|)
      (|Polynomial| . |ConvertibleTo|) (|Polynomial| . |OrderedSet|)
      (|Polynomial| . |RetractableTo|)
      (|Symbol| . |CoercibleTo|) (|Symbol| . |ConvertibleTo|)
      (|Variable| . |CoercibleTo|)))
    (dolist (pair hascategory) (getdatabase pair 'hascategory))
    (setq constructormodemapAndoperationalist '(
      |BasicOperator| |Boolean|
      |CardinalNumber| |Color| |Complex|
      |Database|
      |Equation| |EquationFunctions2| |Expression|
      |Float| |Fraction| |FractionFunctions2|
      |Integer| |IntegralDomain|
      |Kernel|
      |List|
      |Matrix| |MappingPackage1|
      |Operator| |OutputForm|
      |NonNegativeInteger|
      |ParametricPlaneCurve| |ParametricSpaceCurve| |Point| |Polynomial|
      |PolynomialFunctions2| |PositiveInteger|
      |Ring|
      |SetCategory| |SegmentBinding| |SegmentBindingFunctions2| |DoubleFloat|
      |SparseMultivariatePolynomial| |SparseUnivariatePolynomial| |Segment|
      |String| |Symbol|
      |UniversalSegment|
      |Variable| |Vector|))
    (dolist (con constructormodemapAndoperationalist)
      (getdatabase con 'constructormodemap)
      (getdatabase con 'operationalist))
    (setq operation '(
      |+| |-| |*| |/| |**| |coerce| |convert| |elt| |equation|
      |float| |sin| |cos| |map| |SEGMENT|))
    (dolist (op operation) (getdatabase op 'operation))
    (setq constr '( ;these are sorted least-to-most freq. delete early ones first

```



```

|Factored| |SparseUnivariatePolynomialFunctions2| |TableAggregate&| | | | |
|RetractableTo&| |RecursiveAggregate&| |UserDefinedPartialOrdering|
|None| |UnivariatePolynomialCategoryFunctions2| |IntegerPrimesPackage|
|SetCategory&| |IndexedExponents| |QuotientFieldCategory&| |Polynomial|
|EltableAggregate&| |PartialDifferentialRing&| |Set|
|UnivariatePolynomialCategory&| |FlexibleArray|
|SparseMultivariatePolynomial| |PolynomialCategory&|
|DifferentialExtension&| |IndexedFlexibleArray| |AbelianMonoidRing&|
|FiniteAbelianMonoidRing&| |DivisionRing&| |FullyLinearlyExplicitRingOver&|
|IndexedVector| |IndexedOneDimensionalArray| |LocalAlgebra| |Localize|
|Boolean| |Field&| |Vector| |IndexedDirectProductObject| |Aggregate&|
|PolynomialRing| |FreeModule| |IndexedDirectProductAbelianGroup|
|IndexedDirectProductAbelianMonoid| |SingletonAsOrderedSet|
|SparseUnivariatePolynomial| |Fraction| |Collection&| |HomogeneousAggregate&|
|RepeatedSquaring| |IntegerNumberSystem&| |AbelianSemiGroup&|
|AssociationList| |OrderedRing&| |SemiGroup&| |Symbol|
|UniqueFactorizationDomain&| |EuclideanDomain&| |IndexedAggregate&|
|GcdDomain&| |IntegralDomain&| |DifferentialRing&| |Monoid&| |Reference|
|UnaryRecursiveAggregate&| |OrderedSet&| |AbelianGroup&| |Algebra&|
|Module&| |Ring&| |StringAggregate&| |AbelianMonoid&|
|ExtensibleLinearAggregate&| |PositiveInteger| |StreamAggregate&|
|IndexedString| |IndexedList| |ListAggregate&| |LinearAggregate&|
|Character| |String| |NonNegativeInteger| |SingleInteger|
|OneDimensionalArrayAggregate&| |FiniteLinearAggregate&| |PrimitiveArray|
|Integer| |List| |OutputForm|))
(dolist (con constr)
  (let ((c (concatenate 'string
                        (|getEnv| "AXIOM") "/algebra/"
                        (string (getdatabase con 'abbreviation)) ".o"))))
    (format t "  preloading ~a.." c)
    (if (probe-file c)
        (progn
          (put con 'loaded c)
          (load c)
          (format t "loaded.~%"))
        (format t "skipped.~%"))))
  (format t "~%"))

```

63.1.25 defun Open the interp database

Format of an entry in interp.daase:

```

(constructor-name
 operationalist
 constructormodemap
 modemaps -- this should not be needed. eliminate it.

```

```

    object          -- the name of the object file to load for this con.
    constructorcategory -- note that this info is the cadar of the
                        constructormodemap for domains and packages so it is stored
                        as NIL for them. it is valid for categories.
    niladic          -- t or nil directly
    unused
    cosig            -- kept directly
    constructorkind   -- kept directly
    defaultdomain     -- a short list, for %i
    ancestors         -- used to compute new category updates
)

[unsqueeze p982]
[make-database p??]
[DaaseName p979]
[$spadroot p12]
[*allconstructors* p957]
[*interp-stream* p955]
[*interp-stream-stamp* p955]

```

— defun interpOpen —

```

(defun interpOpen ()
  "open the interpreter database and hash the keys"
  (declare (special $spadroot *allconstructors* *interp-stream*
                    *interp-stream-stamp*))
  (let (constructors pos stamp dbstruct)
    (setq *interp-stream* (open (DaaseName "interp.daase" nil)))
    (setq stamp (read *interp-stream*))
    (unless (equal stamp *interp-stream-stamp*)
      (format t "  Re-reading interp.daase")
      (setq *interp-stream-stamp* stamp)
      (setq pos (car stamp))
      (file-position *interp-stream* pos)
      (setq constructors (read *interp-stream*))
      (dolist (item constructors)
        (setq item (unsqueeze item))
        (setq *allconstructors* (adjoin (first item) *allconstructors*))
        (setq dbstruct (make-database))
        (setf (get (car item) 'database) dbstruct)
        (setf (database-operationalist dbstruct) (second item))
        (setf (database-constructormodemap dbstruct) (third item))
        (setf (database-modemaps dbstruct) (fourth item))
        (setf (database-object dbstruct) (fifth item))
        (setf (database-constructorcategory dbstruct) (sixth item))
        (setf (database-niladic dbstruct) (seventh item))
        (setf (database-abbreviation dbstruct) (eighth item))
        (setf (get (eighth item) 'abbreviationfor) (first item)) ;invert
        (setf (database-cosig dbstruct) (ninth item)))

```

```

      (setf (database-constructorkind dbstruct) (tenth item))
      (setf (database-ancestors dbstruct) (nth 11 item))))
  (format t "~&"))

```

This is an initialization function for the constructor database it sets up 2 hash tables, opens the database and hashes the index values.

There is a slight asymmetry in this code. The sourcefile information for system files is only the filename and extension. For user files it contains the full pathname. when the database is first opened the sourcefile slot contains system names. The lookup function has to prefix the “\$spadroot” information if the directory-namestring is null (we don’t know the real root at database build time).

An object-hash table is set up to look up nrlib and ao information. this slot is empty until a user does a)library call. We remember the location of the nrlib or ao file for the users local library at that time. A NIL result from this probe means that the library is in the system-specified place. When we get into multiple library locations this will also contain system files.

63.1.26 defun Open the browse database

Format of an entry in browse.daase:

```

( constructorname
  sourcefile
  constructorform
  documentation
  attributes
  predicates
)

[unsqueeze p982]
[$spadroot p12]
[*allconstructors* p957]
[*browse-stream* p956]
[*browse-stream-stamp* p956]

```

— defun browseOpen —

```

(defun browseOpen ()
  "open the constructor database and hash the keys"
  (declare (special $spadroot *allconstructors* *browse-stream*
                    *browse-stream-stamp*))
  (let (constructors pos stamp dbstruct)
    (setq *browse-stream* (open (DaaseName "browse.daase" nil))))

```

```

(setq stamp (read *browse-stream*))
(unless (equal stamp *browse-stream-stamp*)
  (format t "    Re-reading browse.daase")
  (setq *browse-stream-stamp* stamp)
  (setq pos (car stamp))
  (file-position *browse-stream* pos)
  (setq constructors (read *browse-stream*))
  (dolist (item constructors)
    (setq item (unsqueeze item))
    (unless (setq dbstruct (get (car item) 'database))
      (format t "browseOpen:~%")
      (format t "the browse database contains a constructor ~a~%" item)
      (format t "that is not in the interp.daase file. we cannot~%")
      (format t "get the database structure for this constructor and~%")
      (warn "will create a new one~%")
      (setf (get (car item) 'database) (setq dbstruct (make-database)))
      (setq *allconstructors* (adjoin item *allconstructors*)))
    (setf (database-sourcefile dbstruct) (second item))
    (setf (database-constructorform dbstruct) (third item))
    (setf (database-documentation dbstruct) (fourth item))
    (setf (database-attributes dbstruct) (fifth item))
    (setf (database-predicates dbstruct) (sixth item))
    (setf (database-parents dbstruct) (seventh item))))
  (format t "~&"))

```

63.1.27 defun Open the category database

```

[unsqueeze p982]
[$spadroot p12]
[*hasCategory-hash* p953]
[*category-stream* p956]
[*category-stream-stamp* p957]

```

— defun categoryOpen —

```

(defun categoryOpen ()
  "open category.daase and hash the keys"
  (declare (special $spadroot *hasCategory-hash* *category-stream*
                    *category-stream-stamp*))
  (let (pos keys stamp)
    (setq *category-stream* (open (DaaseName "category.daase" nil)))
    (setq stamp (read *category-stream*))
    (unless (equal stamp *category-stream-stamp*)
      (format t "    Re-reading category.daase")
      (setq *category-stream-stamp* stamp)

```

```

(setq pos (car stamp))
(file-position *category-stream* pos)
(setq keys (read *category-stream*))
(setq *hasCategory-hash* (make-hash-table :test #'equal))
(dolist (item keys)
  (setq item (unsqueeze item))
  (setf (gethash (first item) *hasCategory-hash*) (second item))))
(format t "~&"))

```

63.1.28 defun Open the operations database

```

[unsqueeze p982]
[$spadroot p12]
[*operation-hash* p953]
[*operation-stream* p955]
[*operation-stream-stamp* p956]

```

— defun operationOpen —

```

(defun operationOpen ()
  "read operation database and hash the keys"
  (declare (special $spadroot *operation-hash* *operation-stream*
                    *operation-stream-stamp*))
  (let (operations pos stamp)
    (setq *operation-stream* (open (DaaseName "operation.daase" nil)))
    (setq stamp (read *operation-stream*))
    (unless (equal stamp *operation-stream-stamp*)
      (format t " Re-reading operation.daase")
      (setq *operation-stream-stamp* stamp)
      (setq pos (car stamp))
      (file-position *operation-stream* pos)
      (setq operations (read *operation-stream*))
      (dolist (item operations)
        (setq item (unsqueeze item))
        (setf (gethash (car item) *operation-hash*) (cdr item))))
    (format t "~&"))

```

63.1.29 defun Add operations from newly compiled code

```

[getdatabase p967]
[*operation-hash* p953]

```

— defun addoperations —

```
(defun addoperations (constructor oldmaps)
  "add ops from a )library domain to *operation-hash*"
  (declare (special *operation-hash*))
  (dolist (map oldmaps) ; out with the old
    (let (oldop op)
      (setq op (car map))
      (setq oldop (getdatabase op 'operation))
      (setq oldop (lisp::delete (cdr map) oldop :test #'equal))
      (setf (gethash op *operation-hash*) oldop)))
  (dolist (map (getdatabase constructor 'modemaps)) ; in with the new
    (let (op newmap)
      (setq op (car map))
      (setq newmap (getdatabase op 'operation))
      (setf (gethash op *operation-hash*) (cons (cdr map) newmap)))))
```

63.1.30 defun Show all database attributes of a constructor

[getdatabase p967]

— defun showdatabase —

```
(defun showdatabase (constructor)
  (format t "~&a: ~a%" 'constructorkind
    (getdatabase constructor 'constructorkind))
  (format t "~&a: ~a%" 'cosig
    (getdatabase constructor 'cosig))
  (format t "~&a: ~a%" 'operation
    (getdatabase constructor 'operation))
  (format t "~&a: ~%" 'constructormodemap)
  (pprint (getdatabase constructor 'constructormodemap))
  (format t "~&a: ~%" 'constructorcategory)
  (pprint (getdatabase constructor 'constructorcategory))
  (format t "~&a: ~%" 'operationalist)
  (pprint (getdatabase constructor 'operationalist))
  (format t "~&a: ~%" 'modemaps)
  (pprint (getdatabase constructor 'modemaps))
  (format t "~&a: ~a%" 'hascategory
    (getdatabase constructor 'hascategory))
  (format t "~&a: ~a%" 'object
    (getdatabase constructor 'object))
  (format t "~&a: ~a%" 'niladic
    (getdatabase constructor 'niladic)))
```

```

(format t "~&~a: ~a~%" 'abbreviation
  (getdatabase constructor 'abbreviation))
(format t "~&~a: ~a~%" 'constructor?
  (getdatabase constructor 'constructor?))
(format t "~&~a: ~a~%" 'constructor
  (getdatabase constructor 'constructor))
(format t "~&~a: ~a~%" 'defaultdomain
  (getdatabase constructor 'defaultdomain))
(format t "~&~a: ~a~%" 'ancestors
  (getdatabase constructor 'ancestors))
(format t "~&~a: ~a~%" 'sourcefile
  (getdatabase constructor 'sourcefile))
(format t "~&~a: ~a~%" 'constructorform
  (getdatabase constructor 'constructorform))
(format t "~&~a: ~a~%" 'constructorargs
  (getdatabase constructor 'constructorargs))
(format t "~&~a: ~a~%" 'attributes
  (getdatabase constructor 'attributes))
(format t "~&~a: ~%" 'predicates)
  (pprint (getdatabase constructor 'predicates))
(format t "~&~a: ~a~%" 'documentation
  (getdatabase constructor 'documentation))
(format t "~&~a: ~a~%" 'parents
  (getdatabase constructor 'parents)))

```

63.1.31 defun Set a value for a constructor key in the database

[make-database p??]

— defun setdatabase —

```

(defun setdatabase (constructor key value)
  (let (struct)
    (when (symbolp constructor)
      (unless (setq struct (get constructor 'database))
        (setq struct (make-database))
        (setf (get constructor 'database) struct)))
    (case key
      (abbreviation
       (setf (database-abbreviation struct) value)
       (when (symbolp value)
         (setf (get value 'abbreviationfor) constructor))))
      (constructorkind
       (setf (database-constructorkind struct) value))))))

```

63.1.32 defun Delete a value for a constructor key in the database

— defun deldatabase —

```
(defun deldatabase (constructor key)
  (when (symbolp constructor)
    (case key
      (abbreviation
       (setf (get constructor 'abbreviationfor) nil))))))
```

63.1.33 defun Get constructor information for a database key

```
[warn p??]
[unsqueeze p982]
[$spadroot p12]
[*miss* p954]
[*hascategory-hash* p??]
[*operation-hash* p953]
[*browse-stream* p956]
[*defaultdomain-list* p953]
[*interp-stream* p955]
[*category-stream* p956]
[*hasCategory-hash* p953]
[*operation-stream* p955]
```

— defun getdatabase —

```
(defun getdatabase (constructor key)
  (declare (special $spadroot) (special *miss*))
  (when (eq *miss* t) (format t "getdatabase call: ~20a ~a%" constructor key))
  (let (data table stream ignore struct)
    (declare (ignore ignore)
      (special *hascategory-hash* *operation-hash*
        *browse-stream* *defaultdomain-list* *interp-stream*
        *category-stream* *hasCategory-hash* *operation-stream*))
    (when (or (symbolp constructor)
      (and (eq key 'hascategory) (pairp constructor)))
      (case key
        ; note that abbreviation, constructorkind and cosig are heavy hitters
        ; thus they occur first in the list of things to check
```



```

(abbreviation
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-abbreviation struct))))
(constructorkind
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-constructorkind struct))))
(cosig
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-cosig struct))))
(operation
  (setq stream *operation-stream*)
  (setq data (gethash constructor *operation-hash*)))
(constructormodemap
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-constructormodemap struct))))
(constructcategory
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-constructcategory struct))
    (when (null data) ;domain or package then subfield of constructormodemap
      (setq data (cadar (getdatabase constructor 'constructormodemap))))))
(operationalist
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-operationalist struct))))
(modemaps
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-modemaps struct))))
(hascategory
  (setq table *hasCategory-hash*)
  (setq stream *category-stream*)
  (setq data (gethash constructor table)))
(object
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-object struct))))
(asharp?
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-object struct))))
(niladic
  (setq stream *interp-stream*)
  (when (setq struct (get constructor 'database))
    (setq data (database-niladic struct))))
(constructor?

```

```

    (when (setq struct (get constructor 'database))
      (setq data (when (database-operationalist struct) t))))
(superdomain ; only 2 superdomains in the world
 (case constructor
   (|NonNegativeInteger|
    (setq data '((|Integer|) (IF (< |#1| 0) |false| |true|))))))
   (|PositiveInteger|
    (setq data '((|NonNegativeInteger|) (< 0 |#1|))))))
(constructor
 (when (setq data (get constructor 'abbreviationfor))))
(defaultdomain
 (setq data (cadr (assoc constructor *defaultdomain-list*))))
(ancestors
 (setq stream *interp-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-ancestors struct)))))
(sourcefile
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-sourcefile struct)))))
(constructorform
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-constructorform struct)))))
(constructorargs
 (setq data (cdr (getdatabase constructor 'constructorform))))
(attributes
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-attributes struct)))))
(predicates
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-predicates struct)))))
(documentation
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-documentation struct)))))
(parents
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-parents struct)))))
(users
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-users struct)))))
(dependents
 (setq stream *browse-stream*)
 (when (setq struct (get constructor 'database))
   (setq data (database-dependents struct)))))

```

```

(otherwise (warn "%(GETDATABASE ~a ~a) failed%" constructor key)))
(when (numberp data) ;fetch the real data
(when *miss* (format t "getdatabase miss: ~20a ~a%" constructor key))
(file-position stream data)
(setq data (unsqueeze (read stream)))
(case key ; cache the result of the database read
(operation (setf (gethash constructor *operation-hash*) data))
(hascategory (setf (gethash constructor *hascategory-hash*) data))
(constructorkind (setf (database-constructorkind struct) data))
(cosig (setf (database-cosig struct) data))
(constructormodemap (setf (database-constructormodemap struct) data))
(constructorcategory (setf (database-constructorcategory struct) data))
(operationalist (setf (database-operationalist struct) data))
(modemaps (setf (database-modemaps struct) data))
(object (setf (database-object struct) data))
(niladic (setf (database-niladic struct) data))
(abbreviation (setf (database-abbreviation struct) data))
(constructor (setf (database-constructor struct) data))
(ancestors (setf (database-ancestors struct) data))
(constructorform (setf (database-constructorform struct) data))
(attributes (setf (database-attributes struct) data))
(predicates (setf (database-predicates struct) data))
(documentation (setf (database-documentation struct) data))
(parents (setf (database-parents struct) data))
(users (setf (database-users struct) data))
(dependents (setf (database-dependents struct) data))
(sourcefile (setf (database-sourcefile struct) data))))
(case key ; fixup the special cases
(sourcefile
(when (and data (string= (directory-namestring data) ""))
(string= (pathname-type data) "spad"))
(setq data
(concatenate 'string $spadroot "/../../src/algebra/" data))))
(asharp? ; is this asharp code?
(if (consp data)
(setq data (cdr data))
(setq data nil)))
(object ; fix up system object pathname
(if (consp data)
(setq data
(if (string= (directory-namestring (car data)) "")
(concatenate 'string $spadroot "/algebra/" (car data) ".o")
(car data)))
(when (and data (string= (directory-namestring data) ""))
(setq data (concatenate 'string $spadroot "/algebra/" data ".o"))))))
data))

```

63.1.34 defun The)library top level command

[localdatabase p971]
 [extendLocalLibdb p??]
 [tersyscommand p432]
 [\$newConlist p??]
 [\$options p??]

— **defun library** —

```
(defun |library| (args)
  (let (original-directory)
    (declare (special |$options| |$newConlist|))
    (setq original-directory (get-current-directory))
    (setq |$newConlist| nil)
    (localdatabase args |$options|)
    (|extendLocalLibdb| |$newConlist|)
    (system::chdir original-directory)
    (tersyscommand)))
```

—————

63.1.35 defun Read a local filename and update the hash tables

The localdatabase function tries to find files in the order of:

- nrlib/index.kaf
- .asy
- .ao,
- asharp to .asy

[sayKeyedMsg p331]
 [localnrlib p973]
 [localasy p??]
 [asharp p??]
 [astran p??]
 [localasy p??]
 [\$forceDatabaseUpdate p??]
 [\$ConstructorCache p??]
 [*index-filename* p??]

— **defun localdatabase** —

```

(defun localdatabase (filelist options &optional (make-database? nil))
  "read a local filename and update the hash tables"
  (labels (
    (processOptions (options)
      (let (only dir noexpose)
        (when (setq only (assoc '|only| options))
          (setq options (lisp::delete only options :test #'equal))
          (setq only (cdr only)))
        (when (setq dir (assoc '|dir| options))
          (setq options (lisp::delete dir options :test #'equal))
          (setq dir (second dir))
          (when (null dir)
            (|sayKeyedMsg| 'S2IU0002 nil) ))
        (when (setq noexpose (assoc '|noexpose| options))
          (setq options (lisp::delete noexpose options :test #'equal))
          (setq noexpose 't) )
        (when options
          (format t " Ignoring unknown )library option: ~a~%" options))
        (values only dir noexpose)))
    (processDir (dirarg thisdir)
      (let (allfiles)
        (declare (special vmlisp::*index-filename*))
        (system:chdir (string dirarg))
        (setq allfiles (directory "*"))
        (system:chdir thisdir)
        (mapcan #'(lambda (f)
          (when (string-equal (pathname-type f) "nrlib")
            (list (concatenate 'string (namestring f) "/"
                                vmlisp::*index-filename*)))) allfiles))))
    (let (thisdir nrlibs object only dir key (|$forceDatabaseUpdate| t) noexpose)
      (declare (special |$forceDatabaseUpdate| vmlisp::*index-filename*
                        |$ConstructorCache|))
      (setq thisdir (namestring (truename ".")))
      (setq noexpose nil)
      (multiple-value-setq (only dir noexpose) (processOptions options))
      ;don't force exposure during database build
      (if make-database? (setq noexpose t))
      (when dir (setq nrlibs (processDir dir thisdir)))
      (dolist (file filelist)
        (let ((filename (pathname-name file))
              (namedir (directory-namestring file)))
          (unless namedir (setq thisdir (concatenate 'string thisdir "/")))
          (cond
            ((setq file (probe-file
              (concatenate 'string namedir filename ".nrlib/"
                            vmlisp::*index-filename*)))
              (push (namestring file) nrlibs))
            ('else (format t " )library cannot find the file ~a.~%" filename))))
      (dolist (file (nreverse nrlibs))
        (setq key (pathname-name (first (last (pathname-directory file))))))

```

```
(setq object (concatenate 'string (directory-namestring file) "code"))
(localnrlib key file object make-database? noexpose))
(clrhash |$ConstructorCache|)))
```

63.1.36 defun Update the database from an nrlib index.kaf file

```
[getdatabase p967]
[make-database p??]
[addoperations p964]
[sublislis p??]
[updateDatabase p??]
[installConstructor p??]
[updateCategoryTable p??]
[categoryForm? p??]
[setExposeAddConstr p674]
[startTimingProcess p??]
[loadLibNoUpdate p??]
[sayKeyedMsg p331]
[$FormalMapVariableList p??]
[*allOperations* p957]
[*allconstructors* p957]
```

— defun localnrlib —

```
(defun localnrlib (key nrlib object make-database? noexpose)
  "given a string pathname of an index.kaf and the object update the database"
  (labels (
    (fetchdata (alist in index)
      (let (pos)
        (setq pos (third (assoc index alist :test #'string=)))
        (when pos
          (file-position in pos)
          (read in))))))
    (let (alist kind (systemdir? nil) pos constructorform oldmaps abbrev dbstruct)
      (declare (special *allOperations* *allconstructors*
                        |$FormalMapVariableList|))
      (with-open-file (in nrlib)
        (file-position in (read in))
        (setq alist (read in))
        (setq pos (third (assoc "constructorForm" alist :test #'string=)))
        (file-position in pos)
        (setq constructorform (read in))
        (setq key (car constructorform))
        (setq oldmaps (getdatabase key 'modemaps)))
```

```

(setq dbstruct (make-database))
(setq *allconstructors* (adjoin key *allconstructors*))
(setf (get key 'database) dbstruct) ; store the struct, side-effect it...
(setf (database-constructorform dbstruct) constructorform)
(setq *allOperations* nil) ; force this to recompute
(setf (database-object dbstruct) object)
(setq abbrev
  (intern (pathname-name (first (last (pathname-directory object))))))
(setf (database-abbreviation dbstruct) abbrev)
(setf (get abbrev 'abbreviationfor) key)
(setf (database-operationalist dbstruct) nil)
(setf (database-operationalist dbstruct)
  (fetchdata alist in "operationAlist"))
(setf (database-constructormodemap dbstruct)
  (fetchdata alist in "constructorModemap"))
(setf (database-modemaps dbstruct) (fetchdata alist in "modemaps"))
(setf (database-sourcefile dbstruct) (fetchdata alist in "sourceFile"))
(when make-database?
  (setf (database-sourcefile dbstruct)
    (file-namestring (database-sourcefile dbstruct))))
(setf (database-constructorkind dbstruct)
  (setq kind (fetchdata alist in "constructorKind")))
(setf (database-constructorkind dbstruct)
  (fetchdata alist in "constructorCategory"))
(setf (database-documentation dbstruct)
  (fetchdata alist in "documentation"))
(setf (database-attributes dbstruct)
  (fetchdata alist in "attributes"))
(setf (database-predicates dbstruct)
  (fetchdata alist in "predicates"))
(setf (database-niladic dbstruct)
  (when (fetchdata alist in "NILADIC") t))
(addoperations key oldmaps)
(unless make-database?
  (if (eq kind '|category|)
    (setf (database-ancestors dbstruct)
      (sublislis |$FormalMapVariableList|
        (cdr constructorform) (fetchdata alist in "ancestors"))))
  (|updateDatabase| key key systemdir?) ;makes many hashtables???
  (|installConstructor| key kind) ;used to be key cname ...
  (|updateCategoryTable| key kind)
  (if |$InteractiveMode| (setq |$CategoryFrame| |$EmptyEnvironment|)))
(setf (database-cosig dbstruct)
  (cons nil (mapcar #'|categoryForm|
    (cddar (database-constructormodemap dbstruct)))))
(remprop key 'loaded)
(if (null noexpose) (|setExposeAddConstr| (cons key nil)))
(setf (symbol-function key) ; sets the autoload property for cname
  #'(lambda (&rest args)
    (unless (get key 'loaded)

```

```
(|startTimingProcess| '|load|)
(|loadLibNoUpdate| key key object)) ; used to be cname key
(apply key args)))
(|sayKeyedMsg| 'S2IU0001 (list key object))))))
```

63.1.37 defun Make new databases

Making new databases consists of:

1. reset all of the system hash tables
2. set up Union, Record and Mapping
3. map)library across all of the system files (fills the databases)
4. loading some normally autoloading files
5. making some database entries that are computed (like ancestors)
6. writing out the databases
7. write out 'warm' data to be loaded into the image at build time

Note that this process should be done in a clean image followed by a rebuild of the system image to include the new index pointers (e.g. *interp-stream-stamp*).

The system will work without a rebuild but it needs to re-read the databases on startup. Rebuilding the system will cache the information into the image and the databases are opened but not read, saving considerable startup time. Also note that the order the databases are written out is critical. The interp.daase depends on prior computations and has to be written out last.

The build-name-to-pamphlet-hash builds a hash table whose key-*i* value is:

- abbreviation -*i* pamphlet file name
- abbreviation-line -*i* pamphlet file position
- constructor -*i* pamphlet file name
- constructor-line -*i* pamphlet file position

is the symbol of the constructor name and whose value is the name of the source file without any path information. We hash the constructor abbreviation to pamphlet file name. [local-database p971]

[getEnv p??]

[oldCompilerAutoloadOnceTrigger p??]


```

[browserAutoloadOnceTrigger p??]
[mkTopicHashTable p??]
[buildLibdb p??]
[dbSplitLibdb p??]
[mkUsersHashTable p??]
[saveUsersHashTable p??]
[mkDependentsHashTable p??]
[saveDependentsHashTable p??]
[write-compress p980]
[write-browsedb p989]
[write-operationdb p991]
[write-categorydb p990]
[allConstructors p992]
[categoryForm? p??]
[domainsOf p??]
[getConstructorForm p??]
[write-interpdb p987]
[write-warmdata p991]
[$constructorList p??]
[*sourcefiles* p??]
[*compressvector* p954]
[*allconstructors* p957]
[*operation-hash* p953]

```

— defun make-databases —

```

(defun make-databases (ext dirlist)
  (labels (
    (build-name-to-pamphlet-hash (dir)
      (let ((ht (make-hash-table)) (eof '(done)) point mark abbrev name file ns)
        (dolist (fn (directory dir))
          (with-open-file (f fn)
            (do ((ln (read-line f nil eof) (read-line f nil eof))
                (line 0 (incf line)))
              ((eq ln eof))
              (when (and (setq mark (search ")abb" ln)) (= mark 0))
                (setq mark (position #\space ln :from-end t))
                (setq name (intern (string-trim '(#\space) (subseq ln mark))))
                (cond
                  ((setq mark (search "domain" ln)) (setq mark (+ mark 7)))
                  ((setq mark (search "package" ln)) (setq mark (+ mark 8)))
                  ((setq mark (search "category" ln)) (setq mark (+ mark 9))))
                (setq point (position #\space ln :start (+ mark 1)))
                (setq abbrev
                  (intern (string-trim '(#\space) (subseq ln mark point))))
                (setq ns (namestring fn))
                (setq mark (position #\ / ns :from-end t))
                (setq file (subseq ns (+ mark 1)))

```

```

    (setf (gethash abbrev ht) file)
    (setf (gethash (format nil "~a-line" abbrev) ht) line)
    (setf (gethash name ht) file)
    (setf (gethash (format nil "~a-line" name) ht) line))))
  ht))
;; these are types which have no library object associated with them.
;; we store some constructed data to make them perform like library
;; objects, the *operationalist-hash* key entry is used by allConstructors
(withSpecialConstructors ()
  (declare (special *allconstructors*))
  ; note: if item is not in *operationalist-hash* it will not be written
  ; Category
  (setf (get '|Category| 'database)
    (make-database :operationalist nil :niladic t))
  (push '|Category| *allconstructors*)
  ; UNION
  (setf (get '|Union| 'database)
    (make-database :operationalist nil :constructorkind '|domain|))
  (push '|Union| *allconstructors*)
  ; RECORD
  (setf (get '|Record| 'database)
    (make-database :operationalist nil :constructorkind '|domain|))
  (push '|Record| *allconstructors*)
  ; MAPPING
  (setf (get '|Mapping| 'database)
    (make-database :operationalist nil :constructorkind '|domain|))
  (push '|Mapping| *allconstructors*)
  ; ENUMERATION
  (setf (get '|Enumeration| 'database)
    (make-database :operationalist nil :constructorkind '|domain|))
  (push '|Enumeration| *allconstructors*)
  )
  (final-name (root)
    (format nil "~a.daase~a" root ext))
  )
  (let (d)
    (declare (special |$constructorList| *sourcefiles* *compressvector*
      *allconstructors* *operation-hash*))
    (do-symbols (symbol)
      (when (get symbol 'database)
        (setf (get symbol 'database) nil)))
    (setq *hascategory-hash* (make-hash-table :test #'equal))
    (setq *operation-hash* (make-hash-table))
    (setq *allconstructors* nil)
    (setq *compressvector* nil)
    (withSpecialConstructors)
    (localdatabase nil
      (list (list '|dir| (namestring (truename ". /")) ))
        'make-database)
    (dolist (dir dirlist)

```

```

(localdatabase nil
  (list (list 'dir| (namestring (truename (format nil "~a" dir))))
    'make-database))
;browse.daase
(load (concatenate 'string (|getEnv| "AXIOM") "/autoload/topics")) ;; hack
(|oldCompilerAutoloadOnceTrigger|)
(|browserAutoloadOnceTrigger|)
(|mkTopicHashTable|)
(setq |$constructorList| nil) ;; affects buildLibdb
(setq *sourcefiles* (build-name-to-pamphlet-hash
  (concatenate 'string (|getEnv| "AXIOM")
    "/../../src/algebra/*.pamphlet")))
(|buildLibdb|)
(|dbSplitLibdb|)
; (|dbAugmentConstructorDataTable|)
(|mkUsersHashTable|)
(|saveUsersHashTable|)
(|mkDependentsHashTable|)
(|saveDependentsHashTable|)
; (|buildGloss|)
(write-compress)
(write-browsedb)
(write-operationdb)
; note: genCategoryTable creates a new *hascategory-hash* table
; this smashes the existing table and regenerates it.
; write-categorydb does getdatabase calls to write the new information
(write-categorydb)
(dolist (con (|allConstructors|))
  (let (dbstruct)
    (when (setq dbstruct (get con 'database))
      (setf (database-cosig dbstruct)
        (cons nil (mapcar #'|categoryForm?|
          (cddar (database-constructormodemap dbstruct))))))
    (when (and (|categoryForm?| con)
      (= (length (setq d (|domainsOf| (list con) NIL NIL))) 1))
      (setq d (caar d))
      (when (= (length d) (length (|getConstructorForm| con)))
        (format t "  ~a has a default domain of ~a~%" con (car d))
        (setf (database-defaultdomain dbstruct) (car d))))))
    ; note: genCategoryTable creates *ancestors-hash*. write-interpdb
    ; does gethash calls into it rather than doing a getdatabase call.
  (write-interpdb)
  (write-warmdata)
  (when (probe-file (final-name "compress"))
    (delete-file (final-name "compress")))
  (rename-file "compress.build" (final-name "compress"))
  (when (probe-file (final-name "interp"))
    (delete-file (final-name "interp")))
  (rename-file "interp.build" (final-name "interp"))
  (when (probe-file (final-name "operation"))

```

```

      (delete-file (final-name "operation"))))
(rename-file "operation.build" (final-name "operation"))
(when (probe-file (final-name "browse"))
      (delete-file (final-name "browse")))
(rename-file "browse.build"
            (final-name "browse"))
(when (probe-file (final-name "category"))
      (delete-file (final-name "category")))
(rename-file "category.build"
            (final-name "category"))))

```

63.1.38 defun Construct the proper database full pathname

```

[getEnv p??]
[$spadroot p12]

```

— defun DaaseName —

```

(defun DaaseName (name erase?)
  (let (daase filename)
    (declare (special $spadroot))
    (if (setq daase (|getEnv| "DAASE"))
        (progn
          (setq filename (concatenate 'string daase "/algebra/" name))
          (format t " Using local database ~a." filename))
        (setq filename (concatenate 'string $spadroot "/algebra/" name)))
    (when erase? (system::system (concatenate 'string "rm -f " filename)))
    filename))

```

63.1.39 compress.daase

The compress database is special. It contains a list of symbols. The character string name of a symbol in the other databases is represented by a negative number. To get the real symbol back you take the absolute value of the number and use it as a byte index into the compress database. In this way long symbol names become short negative numbers.

63.1.40 defun Set up compression vectors for the databases

```

[DaaseName p979]
[$spadroot p12]

```

```
[*compressvector* p954]
[*compressVectorLength* p954]
[*compress-stream* p955]
[*compress-stream-stamp* p955]
```

— defun compressOpen —

```
(defun compressOpen ()
  (let (lst stamp pos)
    (declare (special $spadroot *compressvector* *compressVectorLength*
                      *compress-stream* *compress-stream-stamp*))
    (setq *compress-stream*
          (open (DaaseName "compress.daase" nil) :direction :input))
    (setq stamp (read *compress-stream*))
    (unless (equal stamp *compress-stream-stamp*)
      (format t " Re-reading compress.daase")
      (setq *compress-stream-stamp* stamp)
      (setq pos (car stamp))
      (file-position *compress-stream* pos)
      (setq lst (read *compress-stream*))
      (setq *compressVectorLength* (car lst))
      (setq *compressvector*
            (make-array (car lst) :initial-contents (cdr lst))))))
```

—

63.1.41 defvar \$*attributes*

— initvars —

```
(defvar *attributes*
  '(|nil| |infinite| |arbitraryExponent| |approximate| |complex|
    |shallowMutable| |canonical| |noetherian| |central|
    |partiallyOrderedSet| |arbitraryPrecision| |canonicalsClosed|
    |noZeroDivisors| |rightUnitary| |leftUnitary|
    |additiveValuation| |unitsKnown| |canonicalUnitNormal|
    |multiplicativeValuation| |finiteAggregate| |shallowlyMutable|
    |commutative|) "The list of known algebra attributes")
```

—

63.1.42 defun Write out the compress database

```
[allConstructors p992]
[allOperations p992]
```

```
[*compress-stream* p955]
[*attributes* p980]
[*compressVectorLength* p954]
```

— **defun write-compress** —

```
(defun write-compress ()
  (let (compresslist masterpos out)
    (declare (special *compress-stream* *attributes* *compressVectorLength*))
    (close *compress-stream*)
    (setq out (open "compress.build" :direction :output))
    (princ " " out)
    (finish-output out)
    (setq masterpos (file-position out))
    (setq compresslist
      (append (|allConstructors|) (|allOperations|) *attributes*))
    (push "algebra" compresslist)
    (push "failed" compresslist)
    (push 'signature compresslist)
    (push '|ofType| compresslist)
    (push '|Join| compresslist)
    (push 'and compresslist)
    (push '|nobranch| compresslist)
    (push 'category compresslist)
    (push '|category| compresslist)
    (push '|domain| compresslist)
    (push '|package| compresslist)
    (push 'attribute compresslist)
    (push '|isDomain| compresslist)
    (push '|ofCategory| compresslist)
    (push '|Union| compresslist)
    (push '|Record| compresslist)
    (push '|Mapping| compresslist)
    (push '|Enumeration| compresslist)
    (setq *compressVectorLength* (length compresslist))
    (setq *compressvector*
      (make-array *compressVectorLength* :initial-contents compresslist))
    (print (cons (length compresslist) compresslist) out)
    (finish-output out)
    (file-position out 0)
    (print (cons masterpos (get-universal-time)) out)
    (finish-output out)
    (close out)))
```

63.1.43 defun Compress an expression using the compress vector

This function is used to minimize the size of the databases by replacing symbols with indexes into the compression vector. [**compressvector** p954]

— defun squeeze —

```
(defun squeeze (expr)
  (declare (special *compressvector*))
  (let (leaves pos (bound (length *compressvector*)))
    (labels (
      (flat (expr)
        (when (and (numberp expr) (< expr 0) (>= expr bound))
          (print expr)
          (break "squeeze found a negative number"))
        (if (atom expr)
            (unless (or (null expr)
                        (and (symbolp expr) (char= (schar (symbol-name expr) 0) #\*)))
              (setq leaves (adjoin expr leaves)))
            (progn
              (flat (car expr))
              (flat (cdr expr))))))
      (setq leaves nil)
      (flat expr)
      (dolist (leaf leaves)
        (when (setq pos (position leaf *compressvector*))
          (nsubst (- pos) leaf expr)))
      expr)))
```

—————

63.1.44 defun Uncompress an expression using the compress vector

This function is used to recover symbols from the databases by using integers as indexes into the compression vector. [**compressvector** p954]

— defun unsqueeze —

```
(defun unsqueeze (expr)
  (declare (special *compressvector*))
  (cond ((atom expr)
    (cond ((and (numberp expr) (<= expr 0))
      (svref *compressvector* (- expr)))
      (t expr)))
    (t (rplaca expr (unsqueeze (car expr)))
      (rplacd expr (unsqueeze (cdr expr)))
      expr)))
```

63.1.45 Building the interp.daase from hash tables

```

format of an entry in interp.daase:
  (constructor-name
    operationalist
    constructormodemap
    modemaps          -- this should not be needed. eliminate it.
    object            -- the name of the object file to load for this con.
    constructorcategory -- note that this info is the cadar of the
                        constructormodemap for domains and packages so it is stored
                        as NIL for them. it is valid for categories.
    niladic           -- t or nil directly
    unused
    cosig             -- kept directly
    constructorkind   -- kept directly
    defaultdomain     -- a short list, for %i
    ancestors         -- used to compute new category updates
  )

```

Here I'll try to outline the interp database write procedure

```

(defun write-interpdb ()
  "build interp.daase from hash tables"
  (declare (special $spadroot *ancestors-hash*))
  (let (opalistpos modemapspos cmodemappos master masterpos obj *print-pretty*
        concategory categorypos kind niladic cosig abbrev defaultdomain
        ancestors ancestorspos out)
    (declare (special *print-pretty*))
    (print "building interp.daase")

; 1. We open the file we're going to create

    (setq out (open "interp.build" :direction :output))

; 2. We reserve some space at the top of the file for the key-time pair
;    We will overwrite these spaces just before we close the file.

    (princ " " out)

; 3. Make sure we write it out
    (finish-output out)

; 4. For every constructor in the system we write the parts:

```



```

(dolist (constructor (|allConstructors|))
  (let (struct)

; 4a. Each constructor has a property list. A property list is a list
;    of (key . value) pairs. The property we want is called 'database
;    so there is a ('database . something) in the property list

    (setq struct (get constructor 'database))

; 5 We write the "operationsalist"
; 5a. We remember the current file position before we write
;    We need this information so we can seek to this position on read

    (setq opalistpos (file-position out))

; 5b. We get the "operationalist", compress it, and write it out

    (print (squeeze (database-operationalist struct)) out)

; 5c. We make sure it was written

    (finish-output out)

; 6 We write the "constructormodemap"
; 6a. We remember the current file position before we write

    (setq cmodemappos (file-position out))

; 6b. We get the "constructormodemap", compress it, and write it out

    (print (squeeze (database-constructormodemap struct)) out)

; 6c. We make sure it was written

    (finish-output out)

; 7. We write the "modemaps"
; 7a. We remember the current file position before we write

    (setq modemapspos (file-position out))

; 7b. We get the "modemaps", compress it, and write it out

    (print (squeeze (database-modemaps struct)) out)

; 7c. We make sure it was written

    (finish-output out)

; 8. We remember source file pathnames in the obj variable

```

```

(if (consp (database-object struct)) ; if asharp code ...
  (setq obj
    (cons (pathname-name (car (database-object struct)))
          (cdr (database-object struct))))
  (setq obj
    (pathname-name
      (first (last (pathname-directory (database-object struct)))))))

; 9. We write the "constructorcategory", if it is a category, else nil
; 9a. Get the constructorcategory and compress it

(setq concategory (squeeze (database-constructorcategory struct)))

; 9b. If we have any data we write it out, else we don't write it
;     Note that if there is no data then the byte index for the
;     constructorcategory will not be a number but will be nil.

(if concategory ; if category then write data else write nil
  (progn
    (setq categorypos (file-position out))
    (print concategory out)
    (finish-output out))
  (setq categorypos nil))

; 10. We get a set of properties which are kept as "immediate" data
;     This means that the key table will hold this data directly
;     rather than as a byte index into the file.
; 10a. niladic data

(setq niladic (database-niladic struct))

; 10b. abbreviation data (e.g. POLY for polynomial)

(setq abbrev (database-abbreviation struct))

; 10c. cosig data

(setq cosig (database-cosig struct))

; 10d. kind data

(setq kind (database-constructorkind struct))

; 10e. defaultdomain data

(setq defaultdomain (database-defaultdomain struct))

; 11. The ancestor data might exist. If it does we fetch it,
;     compress it, and write it out. If it does not we place

```

```

;      and immediate value of nil in the key-value table

(setq ancestors (squeeze (gethash constructor *ancestors-hash*))) ;cattable.boot
(if ancestors
  (progn
    (setq ancestorspos (file-position out))
    (print ancestors out)
    (finish-output out))
  (setq ancestorspos nil))

; 12. "master" is an alist. Each element of the alist has the name of
;      the constructor and all of the above attributes. When the loop
;      finishes we will have constructed all of the data for the key-value
;      table

(push (list constructor opalistpos cmodemappos modemapspos
  obj categorypos niladic abbrev cosig kind defaultdomain
  ancestorspos) master)))

; 13. The loop is done, we make sure all of the data is written

(finish-output out)

; 14. We remember where the key-value table will be written in the file

(setq masterpos (file-position out))

; 15. We compress and print the key-value table

(print (mapcar #'squeeze master) out)

; 16. We make sure we write the table

(finish-output out)

; 17. We go to the top of the file

(file-position out 0)

; 18. We write out the (master-byte-position . universal-time) pair
;      Note that if the universal-time value matches the value of
;      *interp-stream-stamp* then there is no reason to read the
;      interp database because all of the data is already cached in
;      the image. This happens if you build a database and immediatly
;      save the image. The saved image already has the data since we
;      just wrote it out. If the *interp-stream-stamp* and the database
;      time stamp differ we "reread" the database on startup. Actually
;      we just open the database and fetch as needed. You can see fetches
;      by setting the *miss* variable non-nil.

```

```

(print (cons masterpos (get-universal-time)) out)

; 19. We make sure we write it.

(finish-output out)

; 20 And we are done

(close out)))

```

63.1.46 defun Write the interp database

```

[squeeze p982]
[$spadroot p12]
[*ancestors-hash* p??]
[*print-pretty* p??]

```

— defun write-interpdb —

```

(defun write-interpdb ()
  "build interp.daase from hash tables"
  (declare (special $spadroot *ancestors-hash*))
  (let (opalistpos modemapspos cmodemappos master masterpos obj *print-pretty*
        concategory categorypos kind niladic cosig abbrev defaultdomain
        ancestors ancestorspos out)
    (declare (special *print-pretty*))
    (print "building interp.daase")
    (setq out (open "interp.build" :direction :output))
    (princ " " out)
    (finish-output out)
    (dolist (constructor (|allConstructors|))
      (let (struct)
        (setq struct (get constructor 'database))
        (setq opalistpos (file-position out))
        (print (squeeze (database-operationalist struct)) out)
        (finish-output out)
        (setq cmodemappos (file-position out))
        (print (squeeze (database-constructormodemap struct)) out)
        (finish-output out)
        (setq modemapspos (file-position out))
        (print (squeeze (database-modemaps struct)) out)
        (finish-output out)
        (if (consp (database-object struct)) ; if asharp code ...
            (setq obj
              (cons (pathname-name (car (database-object struct)))
                    (cdr (database-object struct))))
            (setq obj
              (pathname-name

```

```

      (first (last (pathname-directory (database-object struct))))))
    (setq concategory (squeeze (database-constructcategory struct)))
    (if concategory ; if category then write data else write nil
      (progn
        (setq categorypos (file-position out))
        (print concategory out)
        (finish-output out))
      (setq categorypos nil))
    (setq niladic (database-niladic struct))
    (setq abbrev (database-abbreviation struct))
    (setq cosig (database-cosig struct))
    (setq kind (database-constructorkind struct))
    (setq defaultdomain (database-defaultdomain struct))
    (setq ancestors
      (squeeze (gethash constructor *ancestors-hash*))) ;cattable.boot
    (if ancestors
      (progn
        (setq ancestorspos (file-position out))
        (print ancestors out)
        (finish-output out))
      (setq ancestorspos nil))
    (push (list constructor opalistpos cmodemappos modemappos
      obj categorypos niladic abbrev cosig kind defaultdomain
      ancestorspos) master))
    (finish-output out)
    (setq masterpos (file-position out))
    (print (mapcar #'squeeze master) out)
    (finish-output out)
    (file-position out 0)
    (print (cons masterpos (get-universal-time)) out)
    (finish-output out)
    (close out)))

```

63.1.47 Building the browse.daase from hash tables

```

format of an entry in browse.daase:
( constructorname
  sourcefile
  constructorform
  documentation
  attributes
  predicates
)

```

This is essentially the same overall process as write-interpdb.

We reserve some space for the (key-table-byte-position . timestamp)

We loop across the list of constructors dumping the data and remembering the byte positions in a key-value pair table.

We dump the final key-value pair table, write the byte position and time stamp at the top of the file and close the file.

63.1.48 defun Write the browse database

```
[allConstructors p992]
[squeeze p982]
[$spadroot p12]
[*sourcefiles* p??]
[*print-pretty* p??]
```

— defun write-browsedb —

```
(defun write-browsedb ()
  "make browse.daase from hash tables"
  (declare (special $spadroot *sourcefiles*))
  (let (master masterpos src formpos docpos attpos predpos *print-pretty* out)
    (declare (special *print-pretty*))
    (print "building browse.daase")
    (setq out (open "browse.build" :direction :output))
    (princ " " out)
    (finish-output out)
    (dolist (constructor (|allConstructors|))
      (let (struct)
        (setq struct (get constructor 'database))
        ; sourcefile is small. store the string directly
        (setq src (gethash constructor *sourcefiles*))
        (setq formpos (file-position out))
        (print (squeeze (database-structorform struct)) out)
        (finish-output out)
        (setq docpos (file-position out))
        (print (database-documentation struct) out)
        (finish-output out)
        (setq attpos (file-position out))
        (print (squeeze (database-attributes struct)) out)
        (finish-output out)
        (setq predpos (file-position out))
        (print (squeeze (database-predicates struct)) out)
        (finish-output out)
        (push (list constructor src formpos docpos attpos predpos) master)))
    (finish-output out)
    (setq masterpos (file-position out))
    (print (mapcar #'squeeze master) out)
    (finish-output out))
```

```

(file-position out 0)
(print (cons masterpos (get-universal-time)) out)
(finish-output out)
(close out)))

```

63.1.49 Building the category.daase from hash tables

This is a single table of category hash table information, dumped in the database format.

63.1.50 defun Write the category database

```

[genCategoryTable p??]
[squeeze p982]
[*print-pretty* p??]
[*hasCategory-hash* p953]

```

— defun write-categorydb —

```

(defun write-categorydb ()
  "make category.daase from scratch. contains the *hasCategory-hash* table"
  (let (out master pos *print-pretty*)
    (declare (special *print-pretty* *hasCategory-hash*))
    (print "building category.daase")
    (|genCategoryTable|)
    (setq out (open "category.build" :direction :output))
    (princ " " out)
    (finish-output out)
    (maphash #'(lambda (key value)
      (if (or (null value) (eq value t))
        (setq pos value)
        (progn
          (setq pos (file-position out))
          (print (squeeze value) out)
          (finish-output out)))
        (push (list key pos) master))
      *hasCategory-hash*)
    (setq pos (file-position out))
    (print (mapcar #'squeeze master) out)
    (finish-output out)
    (file-position out 0)
    (print (cons pos (get-universal-time)) out)
    (finish-output out)
    (close out)))

```

63.1.51 Building the operation.daase from hash tables

This is a single table of operations hash table information, dumped in the database format.

63.1.52 defun Write the operations database

[squeeze p982]
[*operation-hash* p953]

— defun write-operationdb —

```
(defun write-operationdb ()
  (let (pos master out)
    (declare (special leaves *operation-hash*))
    (setq out (open "operation.build" :direction :output))
    (princ " " out)
    (finish-output out)
    (maphash #'(lambda (key value)
      (setq pos (file-position out))
      (print (squeeze value) out)
      (finish-output out)
      (push (cons key pos) master))
      *operation-hash*)
    (finish-output out)
    (setq pos (file-position out))
    (print (mapcar #'squeeze master) out)
    (file-position out 0)
    (print (cons pos (get-universal-time)) out)
    (finish-output out)
    (close out)))
```

63.1.53 Database support operations

63.1.54 defun Data preloaded into the image at build time

[\$topicHash p??]

— defun write-warmdata —

```
(defun write-warmdata ()
  "write out information to be loaded into the image at build time"
```



```
(declare (special |$topicHash|))
(with-open-file (out "warm.data" :direction :output)
  (format out "(in-package \"BOOT\")~%")
  (format out "(setq |$topicHash| (make-hash-table))~%")
  (maphash #'(lambda (k v)
    (format out "(setf (gethash '|~a| |$topicHash|) ~a)~%" k v)) |$topicHash|)))
```

63.1.55 defun Return all constructors

[*allconstructors* p957]

— defun allConstructors —

```
(defun |allConstructors| ()
  (declare (special *allconstructors*))
  *allconstructors*)
```

63.1.56 defun Return all operations

[*allOperations* p957]

[*operation-hash* p953]

— defun allOperations —

```
(defun |allOperations| ()
  (declare (special *allOperations* *operation-hash*))
  (unless *allOperations*
    (maphash #'(lambda (k v) (declare (ignore v)) (push k *allOperations*))
      *operation-hash*))
  *allOperations*)
```

Chapter 64

System Statistics

[gbc-time p??]

— defun statisticsInitialization —

```
(defun |statisticsInitialization| ()  
  "initialize the garbage collection timer"  
  #+:akcl (system:gbc-time 0)  
  nil)
```

—————

Chapter 65

Special Lisp Functions

65.1 Axiom control structure macros

Axiom used various control structures in the boot code which are not available in Common Lisp. We write some macros here to make the boot to lisp translations easier to read.

65.1.1 defun put

— defun put —

```
(defun put (sym ind val) (setf (get sym ind) val))
```

—————

65.1.2 defmacro while

While the condition is true, repeat the body. When the condition is false, return t.

— defmacro while —

```
(defmacro while (condition &rest body)
  '(loop (if (not ,condition) (return t)) ,@body))
```

—————

65.1.3 defmacro whileWithResult

While the condition is true, repeat the body. When the condition is false, return the result form's value.

— **defmacro whileWithResult** —

```
(defmacro whileWithResult (condition result &rest body)
  '(loop (if (not ,condition) ,@result) ,@body))
```

—————

65.2 Filename Handling

This code implements the Common Lisp pathname functions for Lisp/VM. On VM, a filename is 3-list consisting of the filename, filetype and filemode. We also UPCASE everything.

65.2.1 defun namestring

[pathname p998]

— **defun namestring** —

```
(defun |namestring| (arg)
  (namestring (|pathname| arg)))
```

—————

65.2.2 defun pathnameName

[pathname p998]

— **defun pathnameName** —

```
(defun |pathnameName| (arg)
  (pathname-name (|pathname| arg)))
```

—————

65.2.3 defun pathnameType

[pathname p998]

— **defun pathnameType** —

```
(defun |pathnameType| (arg)
  (pathname-type (|pathname| arg)))
```

65.2.4 **defun pathnameTypeId**

```
[upcase p??]
[object2Identifier p??]
[pathnameType p996]
```

— **defun pathnameTypeId** —

```
(defun |pathnameTypeId| (arg)
  (upcase (|object2Identifier| (|pathnameType| arg))))
```

65.2.5 **defun mergePathnames**

```
[pathnameName p996]
[nequal p??]
[pathnameType p996]
[pathnameDirectory p998]
```

— **defun mergePathnames** —

```
(defun |mergePathnames| (a b)
  (let (fn ft fm)
    (cond
      ((string= (setq fn (|pathnameName| a)) "*") b)
      ((nequal fn (|pathnameName| b)) a)
      ((string= (setq ft (|pathnameType| a)) "*") b)
      ((nequal ft (|pathnameType| b)) a)
      ((equal (setq fm (|pathnameDirectory| a)) (list "*" )) b)
      (t a))))
```

65.2.6 defun pathnameDirectory

[pathname p998]

— defun pathnameDirectory —

```
(defun |pathnameDirectory| (arg)
  (namestring (make-pathname :directory (pathname-directory (|pathname| arg))))))
```

—————

65.2.7 defun Axiom pathnames

[pairp p??]

[pathname p998]

[make-filename p??]

— defun pathname —

```
(defun |pathname| (p)
  (cond
    ((null p) p)
    ((pathnamep p) p)
    ((null (pairp p)) (pathname p))
    (t
     (when (> (|#| p) 2) (setq p (cons (elt p 0) (cons (elt p 1) nil))))
     (pathname (apply #'make-filename p)))))
```

—————

65.2.8 defun makePathname

[pathname p998]

[object2String p??]

— defun makePathname —

```
(defun |makePathname| (name type dir)
  (declare (ignore dir))
  (|pathname| (list (|object2String| name) (|object2String| type))))
```

—————

65.2.9 defun Delete a file

```
[erase p??]
[pathname p998]
[$erase p??]
```

— **defun deleteFile** —

```
(defun |deleteFile| (arg)
  (declare (special $erase))
  ($erase (|pathname| arg)))
```

65.2.10 defun wrap

```
[pairp p??]
[lotsof p999]
[wrap p999]
```

— **defun wrap** —

```
(defun wrap (list-of-items wrapper)
  (prog nil
    (cond
      ((or (not (pairp list-of-items)) (not wrapper))
        (return list-of-items))
      ((not (consp wrapper))
        (setq wrapper (lotsof wrapper))))
    (return
      (cons
        (if (first wrapper)
          '(. (first wrapper) (first list-of-items))
          (first list-of-items))
        (wrap (cdr list-of-items) (cdr wrapper))))))
```

65.2.11 defun lotsof

— **defun lotsof** —

```
(defun lotsof (&rest items)
```



```
(setq items (copy-list items))
(nconc items items))
```

65.2.12 defmacro startsId?

— defmacro startsId? —

```
(defmacro |startsId?| (x)
  '(or (alpha-char-p ,x) (member ,x '(#\? #\% #\!) :test #'char=)))
```

65.2.13 defun hput

— defun hput —

```
(defun hput (table key value)
  (setf (gethash key table) value))
```

65.2.14 defmacro hget

— defmacro hget —

```
(defmacro HGET (table key &rest default)
  '(gethash ,key ,table ,@default))
```

65.2.15 defun hkeys

— defun hkeys —

```
(defun hkeys (table)
  (let (keys)
    (maphash
      #'(lambda (key val) (declare (ignore val)) (push key keys)) table)
    keys))
```

65.2.16 defun digitp

[digitp p1001]

— defun digitp —

```
(defun digitp (x)
  (or (and (symbolp x) (digitp (symbol-name x)))
      (and (characterp x) (digit-char-p x))
      (and (stringp x) (= (length x) 1) (digit-char-p (char x 0)))))
```

65.2.17 defun pname

Note it is important that PNAME returns nil not an error for non-symbols

— defun pname 0 —

```
(defun pname (x)
  (cond ((symbolp x) (symbol-name x))
        ((characterp x) (string x))
        (t nil)))
```

65.2.18 defun size

— defun size —

```
(defun size (l)
  (cond
    ((vectorp l) (length l))
    ((consp l) (list-length l))
    (t 0)))
```

65.2.19 defun strpos

— defun strpos —

```
(defun strpos (what in start dontcare)
  (setq what (string what) in (string in))
  (if dontcare
    (progn
      (setq dontcare (character dontcare))
      (search what in :start2 start
               :test #'(lambda (x y) (or (eql x dontcare) (eql x y)))))
    (if (= start 0)
        (search what in)
        (search what in :start2 start))))
```

65.2.20 defun strposl

Note that this assumes “table” is a string.

— defun strposl —

```
(defun strposl (table cvec sint item)
  (setq cvec (string cvec))
  (if (not item)
      (position table cvec :test #'(lambda (x y) (position y x)) :start sint)
      (position table cvec :test-not #'(lambda (x y) (position y x)) :start sint)))
```

65.2.21 defun qenum

— defun qenum 0 —

```
(defun qenum (cvec ind)
  (char-code (char cvec ind)))
```

65.2.22 defmacro identp— **defmacro identp 0** —

```
(defmacro identp (x)
  (if (atom x)
      '(and ,x (symbolp ,x))
      (let ((xx (gensym)))
        '(let ((,xx ,x)
              (and ,xx (symbolp ,xx)))))))
```

65.2.23 defun concat

[string-concatenate p??]

— **defun concat 0** —

```
(defun concat (a b &rest l)
  (if (bit-vector-p a)
      (if l
          (apply #'concatenate 'bit-vector a b l)
          (concatenate 'bit-vector a b))
      (if l
          (apply #'system:string-concatenate a b l)
          (system:string-concatenate a b))))
```

65.2.24 defun functionp

[identp p1003]

— **defun functionp** —

```
(defun |functionp| (fn)
  (if (identp fn)
      (and (fboundp fn) (not (macro-function fn)))
      (functionp fn)))
```

;; —————¿ NEW DEFINITION (override in msgdb.boot.pamphlet)

65.2.25 defun brightprint

```
[messageprint p1004]
```

```
— defun brightprint —
```

```
(defun brightprint (x)
  (messageprint x))
```

```
—————
```

```
:: —————; NEW DEFINITION (override in msgdb.boot.pamphlet)
```

65.2.26 defun brightprint-0

```
[messageprint-1 p1005]
```

```
— defun brightprint-0 —
```

```
(defun brightprint-0 (x)
  (messageprint-1 x))
```

```
—————
```

65.2.27 defun member

```
— defun member 0 —
```

```
(defun |member| (item sequence)
  (cond
    ((symbolp item) (member item sequence :test #'eq))
    ((stringp item) (member item sequence :test #'equal))
    ((and (atom item) (not (arrayp item))) (member item sequence))
    (t (member item sequence :test #'equalp))))
```

```
—————
```

65.2.28 defun messageprint

```
— defun messageprint —
```

```
(defun messageprint (x)
  (mapc #'messageprint-1 x))
```

65.2.29 defun messageprint-1

```
[identp p1003]
[messageprint-1 p1005]
[messageprint-2 p1005]
```

— defun messageprint-1 —

```
(defun messageprint-1 (x)
  (cond
    ((or (eq x '|%l|) (equal x "%l")) (terpri))
    ((stringp x) (princ x))
    ((identp x) (princ x))
    ((atom x) (princ x))
    ((princ "(")
     (messageprint-1 (car x))
     (messageprint-2 (cdr x))
     (princ ")"))))
```

65.2.30 defun messageprint-2

```
[messageprint-1 p1005]
[messageprint-2 p1005]
```

— defun messageprint-2 —

```
(defun messageprint-2 (x)
  (if (atom x)
      (unless x (progn (princ " . ") (messageprint-1 x)))
      (progn (princ " ") (messageprint-1 (car x)) (messageprint-2 (cdr x)))))
```

65.2.31 defun sayBrightly1

```
[brightprint-0 p1004]
[brightprint p1004]
```

— defun sayBrightly1 —

```
(defun sayBrightly1 (x *standard-output*)  
  (if (atom x)  
      (progn (brightprint-0 x) (terpri) (force-output))  
      (progn (brightprint x) (terpri) (force-output))))
```

—————

65.2.32 defmacro assq

TPDHERE: This could probably be replaced by the default assoc using eql

— defmacro assq —

```
(defmacro assq (a b)  
  '(assoc ,a ,b :test #'eq))
```

—————

Chapter 66

Common Lisp Algebra Support

These functions are called directly from the algebra source code. They fall into two basic categories, one are the functions that are raw Common Lisp calls and the other are Axiom specific functions or macros.

Raw function calls are used where there is an alignment of the Axiom type and the underlying representation in Common Lisp. These form the support pillars upon which Axiom rests. For instance, the 'EQ' function is called to support the Axiom equivalent 'eq?' function.

Macros are used to add type information in order to make low level operations faster. An example is the use of macros in DoubleFloat to add Common Lisp type information. Since DoubleFloat is machine arithmetic we give the compiler explicit type information so it can generate fast code.

Functions are used to do manipulations which are Common Lisp operations but the Axiom semantics are not the same. Because Axiom was originally written in Maclisp, then VMLisp, and then Common Lisp some of these old semantics survive.

66.1 SingleInteger

66.1.1 defun qsquotient

— defun qsquotient 0 —

```
(defun qsquotient (a b)
  (the fixnum (truncate (the fixnum a) (the fixnum b))))
```

—————

66.1.2 defun qsremainder

— defun qsremainder 0 —

```
(defun qsremainder (a b)
  (the fixnum (rem (the fixnum a) (the fixnum b))))
```

—————

66.1.3 defmacro qsdifference

— defmacro qsdifference 0 —

```
(defmacro qsdifference (x y)
  '(the fixnum (- (the fixnum ,x) (the fixnum ,y))))
```

—————

66.1.4 defmacro qslessp

— defmacro qslessp 0 —

```
(defmacro qslessp (a b)
  '(< (the fixnum ,a) (the fixnum ,b)))
```

—————

66.1.5 defmacro qsadd1

— defmacro qsadd1 0 —

```
(defmacro qsadd1 (x)
  '(the fixnum (1+ (the fixnum ,x))))
```

—————

66.1.6 defmacro qssub1

— defmacro qssub1 0 —

```
(defmacro qssub1 (x)
  '(the fixnum (1- (the fixnum ,x))))
```

—————

66.1.7 defmacro qsminus

— defmacro qsminus 0 —

```
(defmacro qsminus (x)
  '(the fixnum (minus (the fixnum ,x))))
```

—————

66.1.8 defmacro qsplus

— defmacro qsplus 0 —

```
(defmacro qsplus (x y)
  '(the fixnum (+ (the fixnum ,x) (the fixnum ,y))))
```

—————

66.1.9 defmacro qstimes

— defmacro qstimes 0 —

```
(defmacro qstimes (x y)
  '(the fixnum (* (the fixnum ,x) (the fixnum ,y))))
```

—————

66.1.10 defmacro qsabsval

— defmacro qsabsval 0 —

```
(defmacro qsabsval (x)
  '(the fixnum (abs (the fixnum ,x))))
```

—————

66.1.11 defmacro qsoddp

— defmacro qsoddp 0 —

```
(defmacro qsoddp (x)
  '(oddp (the fixnum ,x)))
```

—————

66.1.12 defmacro qszerop

— defmacro qszerop 0 —

```
(defmacro qszerop (x)
  '(zerop (the fixnum ,x)))
```

—————

66.1.13 defmacro qsmax

— defmacro qsmax 0 —

```
(defmacro qsmax (x y)
  '(the fixnum (max (the fixnum ,x) (the fixnum ,y))))
```

—————

66.1.14 defmacro qsmín

— defmacro qsmín 0 —

```
(defmacro qsmín (x y)
  '(the fixnum (min (the fixnum ,x) (the fixnum ,y))))
```

—————

66.2 Boolean**66.2.1 defun The Boolean = function support**

— defun BooleanEquality 0 —

```
(defun |BooleanEquality| (x y) (if x y (null y)))
```

—————

66.3 IndexedBits**66.3.1 defmacro truth-to-bit**

IndexedBits new function support

— defmacro truth-to-bit —

```
(defmacro truth-to-bit (x) '(cond (,x 1) ('else 0)))
```

—————

66.3.2 defun IndexedBits new function support

— defun bvec-make-full 0 —

```
(defun bvec-make-full (n x)
  (make-array (list n) :element-type 'bit :initial-element x))
```

—————

66.3.3 defmacro bit-to-truth

IndexedBits elt function support

— **defmacro bit-to-truth 0** —

```
(defmacro bit-to-truth (b) '(eq ,b 1))
```

66.3.4 defmacro bvec-elt

IndexedBits elt function support

— **defmacro bvec-elt 0** —

```
(defmacro bvec-elt (bv i) '(sbit ,bv ,i))
```

66.3.5 defmacro bvec-setelt

IndexedBits setelt function support

— **defmacro bvec-setelt** —

```
(defmacro bvec-setelt (bv i x) '(setf (sbit ,bv ,i) ,x))
```

66.3.6 defmacro bvec-size

IndexedBits length function support

— **defmacro bvec-size** —

```
(defmacro bvec-size (bv) '(size ,bv))
```

66.3.7 defun IndexedBits concat function support

— **defun bvec-concat 0** —

```
(defun bvec-concat (bv1 bv2) (concatenate '(vector bit) bv1 bv2))
```

66.3.8 defun IndexedBits copy function support

— defun bvec-copy 0 —

```
(defun bvec-copy (bv) (copy-seq bv))
```

66.3.9 defun IndexedBits = function support

— defun bvec-equal 0 —

```
(defun bvec-equal (bv1 bv2) (equal bv1 bv2))
```

66.3.10 defun IndexedBits < function support

— defun bvec-greater 0 —

```
(defun bvec-greater (bv1 bv2)
  (let ((pos (mismatch bv1 bv2)))
    (cond ((or (null pos) (>= pos (length bv1))) nil)
          ((< pos (length bv2)) (> (bit bv1 pos) (bit bv2 pos)))
          ((find 1 bv1 :start pos) t)
          (t nil))))
```

66.3.11 defun IndexedBits And function support

— defun bvec-and 0 —

```
(defun bvec-and (bv1 bv2) (bit-and  bv1 bv2))
```

66.3.12 defun IndexedBits Or function support

— defun bvec-or 0 —

```
(defun bvec-or (bv1 bv2) (bit-ior  bv1 bv2))
```

66.3.13 defun IndexedBits xor function support

— defun bvec-xor 0 —

```
(defun bvec-xor (bv1 bv2) (bit-xor  bv1 bv2))
```

66.3.14 defun IndexedBits nand function support

— defun bvec-nand 0 —

```
(defun bvec-nand (bv1 bv2) (bit-nand bv1 bv2))
```

66.3.15 defun IndexedBits nor function support

— defun bvec-nor 0 —

```
(defun bvec-nor (bv1 bv2) (bit-nor  bv1 bv2))
```

66.3.16 defun IndexedBits not function support

— defun bvec-not 0 —

```
(defun bvec-not (bv) (bit-not bv))
```

—————

66.4 KeyedAccessFile**66.4.1 defun KeyedAccessFile defstream function support**

This is a simpler interface to RDEFIOSTREAM [rdefiostream p??]

— defun rdefinstream —

```
(defun rdefinstream (&rest fn)
  ;; following line prevents rdefiostream from adding a default filetype
  (unless (rest fn) (setq fn (list (pathname (car fn)))))
  (rdefiostream (list (cons 'file fn) '(mode . input))))
```

—————

66.4.2 defun KeyedAccessFile defstream function support

[rdefiostream p??]

— defun rdefoutstream —

```
(defun rdefoutstream (&rest fn)
  ;; following line prevents rdefiostream from adding a default filetype
  (unless (rest fn) (setq fn (list (pathname (car fn)))))
  (rdefiostream (list (cons 'FILE fn) '(mode . OUTPUT))))
```

—————

66.5 Table

66.5.1 defun Table InnerTable support

We look inside the Key domain given to Table and find if there is an equality predicate associated with the domain. If found then Table will use a HashTable representation, otherwise it will use an AssociationList representation [knownEqualPred p??]

```
[compiledLookup p??]
[Boolean p??]
[bpname p??]
[knownEqualPred p??]
```

— defun hashable —

```
(defun |hashable| (dom)
  (labels (
    (|knownEqualPred| (dom)
      (let ((fun (|compiledLookup| '= '(|Boolean|) $ $) dom)))
      (if fun
        (get (bpname (car fun)) '|SPADreplace|
              nil))))
    (member (|knownEqualPred| dom) '(eq eql equal))))
```

—

66.6 Plot3d

We catch numeric errors and throw a different failure than normal. The trapNumericErrors macro will return a pair of the the form Union(type-of-form, "failed"). This pair is tested for eq-ness so it has to be unique. It lives in the defvar \$numericFailure. The old value of the \$BreakMode variable is saved in a defvar named \$oldBreakMode.

66.6.1 defvar \$numericFailure

This is a failed union branch which is the value returned for numeric failure.

— initvars —

```
(defvar |$numericFailure| (cons 1 "failed"))
```

—

66.6.2 defvar \$oldBreakMode

— **initvars** —

```
(defvar |$oldBreakMode| nil "the old value of the $BreakMode variable")
```

66.6.3 defmacro trapNumericErrors

The following macro evaluates form returning Union(type-of form, "failed"). It is used in the myTrap local function in Plot3d.

— **defmacro trapNumericErrors** —

```
(defmacro |trapNumericErrors| (form)
  '(let ((|$oldBreakMode| |$BreakMode|) (|$BreakMode| 'trapNumerics|) (val))
    (declare (special |$BreakMode| |$numericFailure| |$oldBreakMode|))
    (setq val (catch 'trapNumerics| ,form))
    (if (eq val |$numericFailure|) val (cons 0 val))))
```

66.7 DoubleFloatVector

Double Float Vectors are simple arrays of lisp double-floats made available at the Spad language level. Note that these vectors are 0 based whereas other Spad language vectors are 1-based.

66.7.1 defmacro dlen

DoubleFloatVector Qsize function support

— **defmacro dlen** —

```
(defmacro dlen (v)
  '(length (the (simple-array double-float (*)) ,v)))
```

66.7.2 defmacro make-double-vector

DoubleFloatVector Qnew function support

— **defmacro make-double-vector** —

```
(defmacro make-double-vector (n)
  '(make-array (list ,n) :element-type 'double-float))
```

66.7.3 defmacro make-double-vector1

DoubleFloatVector Qnew1 function support

— **defmacro make-double-vector1** —

```
(defmacro make-double-vector1 (n s)
  '(make-array (list ,n) :element-type 'double-float :initial-element ,s))
```

66.7.4 defmacro delt

DoubleFloatVector Qelt1 function support

— **defmacro delt** —

```
(defmacro delt (v i)
  '(aref (the (simple-array double-float (*)) ,v) ,i))
```

66.7.5 defmacro dsetelt

DoubleFloatVector Qsetelt1 function support

— **defmacro dsetelt** —

```
(defmacro dsetelt (v i s)
  '(setf (aref (the (simple-array double-float (*)) ,v) ,i) ,s))
```

66.8 ComplexDoubleFloatVector

Complex Double Float Vectors are simple arrays of lisp double-floats made available at the Spad language level. Note that these vectors are 0 based whereas other Spad language vectors are 1-based. Complex array is implemented as an array of doubles. Each complex number occupies two positions in the real array.

66.8.1 defmacro make-cdouble-vector

ComplexDoubleFloatVector Qnew function support
— **defmacro make-cdouble-vector** —

```
(defmacro make-cdouble-vector (n)
  '(make-array (list (* 2 ,n)) :element-type 'double-float))
```

66.8.2 defmacro cdelt

ComplexDoubleFloatVector Qelt1 function support
— **defmacro cdelt** —

```
(defmacro CDELTA(ov oi)
  (let ((v (gensym))
        (i (gensym)))
    '(let ((,v ,ov)
          (,i ,oi))
      (cons
        (aref (the (simple-array double-float (*)) ,v) (* 2 ,i))
        (aref (the (simple-array double-float (*)) ,v) (+ (* 2 ,i) 1))))))
```

66.8.3 defmacro cdsetelt

ComplexDoubleFloatVector Qsetelt1 function support
— **defmacro cdsetelt** —

```
(defmacro cdsetelt(ov oi os)
  (let ((v (gensym))
        (i (gensym))
        (s (gensym)))
    '(let ((,v ,ov)
          (,i ,oi)
          (,s ,os))
      (setf (aref (the (simple-array double-float (*)) ,v) (* 2 ,i))
            (aref (the (simple-array double-float (*)) ,s) ,s))))
```

```

      (,i ,oi)
      (,s ,os))
    (setf (aref (the (simple-array double-float (*)) ,v) (* 2 ,i))
      (car ,s))
    (setf (aref (the (simple-array double-float (*)) ,v) (+ (* 2 ,i) 1))
      (cdr ,s))
    ,s)))

```

66.8.4 defmacro cdlen

ComplexDoubleFloatVector Qsize function support

— **defmacro cdlen** —

```

(defmacro cdlen(v)
  '(truncate (length (the (simple-array double-float (*)) ,v)) 2))

```

66.9 DoubleFloatMatrix

66.9.1 defmacro make-double-matrix

DoubleFloatMatrix qnew function support

— **defmacro make-double-matrix** —

```

(defmacro make-double-matrix (n m)
  '(make-array (list ,n ,m) :element-type 'double-float))

```

66.9.2 defmacro make-double-matrix1

DoubleFloatMatrix new function support

— **defmacro make-double-matrix1** —

```

(defmacro make-double-matrix1 (n m s)
  '(make-array (list ,n ,m) :element-type 'double-float
    :initial-element ,s))

```

66.9.3 defmacro daref2

DoubleFloatMatrix qelt function support

— **defmacro daref2** —

```
(defmacro daref2 (v i j)
  '(aref (the (simple-array double-float (* *)) ,v) ,i ,j))
```

66.9.4 defmacro dsetaref2

DoubleFloatMatrix qsetelt! function support

— **defmacro dsetaref2** —

```
(defmacro dsetaref2 (v i j s)
  '(setf (aref (the (simple-array double-float (* *)) ,v) ,i ,j)
    ,s))
```

66.9.5 defmacro danrows

DoubleFloatMatrix nrow function support

— **defmacro danrows** —

```
(defmacro danrows (v)
  '(array-dimension (the (simple-array double-float (* *)) ,v) 0))
```

66.9.6 defmacro dancols

DoubleFloatMatrix ncol function support

— **defmacro dancols** —

```
(defmacro dancols (v)
  '(array-dimension (the (simple-array double-float (* *)) ,v) 1))
```

66.10 ComplexDoubleFloatMatrix

66.10.1 defmacro make-cdouble-matrix

ComplexDoubleFloatMatrix function support

— **defmacro make-cdouble-matrix** —

```
(defmacro make-cdouble-matrix (n m)
  '(make-array (list ,n (* 2 ,m)) :element-type 'double-float))
```

66.10.2 defmacro cdaref2

ComplexDoubleFloatMatrix function support

— **defmacro cdaref2** —

```
(defmacro cdaref2 (ov oi oj)
  (let ((v (gensym))
        (i (gensym))
        (j (gensym)))
    '(let ((,v ,ov)
          (,i ,oi)
          (,j ,oj))
      (cons
        (aref (the (simple-array double-float (* *)) ,v) ,i (* 2 ,j))
        (aref (the (simple-array double-float (* *)) ,v)
              ,i (+ (* 2 ,j) 1))))))
```

66.10.3 defmacro cdsetaref2

ComplexDoubleFloatMatrix function support

— **defmacro cdsetaref2** —

```
(defmacro cdsetaref2 (ov oi oj os)
  (let ((v (gensym))
        (i (gensym))
        (j (gensym))
        (s (gensym)))
    '(let ((,v ,ov)
          (,i ,oi)
          (,j ,oj)
```

```

      (,s ,os))
      (setf (aref (the (simple-array double-float (* *)) ,v) ,i (* 2 ,j))
            (car ,s))
      (setf (aref (the (simple-array double-float (* *)) ,v)
                  ,i (+ (* 2 ,j) 1))
            (cdr ,s))
      ,s)))

```

66.10.4 defmacro cdanrows

ComplexDoubleFloatMatrix function support

— **defmacro cdanrows** —

```

(defmacro cdanrows (v)
  '(array-dimension (the (simple-array double-float (* *)) ,v) 0))

```

66.10.5 defmacro cdancols

ComplexDoubleFloatMatrix function support

— **defmacro cdancols** —

```

(defmacro cdancols (v)
  '(truncate
    (array-dimension (the (simple-array double-float (* *)) ,v) 1) 2))

```

66.11 Integer

66.11.1 defun Integer divide function support

Note that this is defined as a SPADReplace function in Integer so that algebra code that uses the Integer divide function actually inlines a call to this code. The Integer domain contains the line:

```

(PUT (QUOTE |INT;divide;2$R;44|) (QUOTE |SPADreplace|) (QUOTE DIVIDE2))

```

— **defun divide2 0** —


```
(defun divide2 (x y)
  (multiple-value-call #'cons (truncate x y)))
```

66.11.2 defun Integer quo function support

Note that this is defined as a SPADReplace function in Integer so that algebra code that uses the Integer quo function actually inlines a call to this code. The Integer domain contains the line:

```
(PUT (QUOTE |INT;rem;3$;46|) (QUOTE |SPADreplace|) (QUOTE REMAINDER2))
```

Because these are identical except for name we make the symbol-functions equivalent. This was done in the original code for efficiency.

— **defun remainder2 0** —

```
(setf (symbol-function 'remainder2) #'rem)
```

66.11.3 defun Integer quo function support

Note that this is defined as a SPADReplace function in Integer so that algebra code that uses the Integer quo function actually inlines a call to this code. The Integer domain contains the line:

```
(PUT (QUOTE |INT;quo;3$;45|) (QUOTE |SPADreplace|) (QUOTE QUOTIENT2))
```

— **defun quotient2 0** —

```
(defun quotient2 (x y)
  (values (truncate x y)))
```

66.11.4 defun Integer random function support

This is used for calls to random with no arguments. If an argument is supplied to random then the common lisp random function is called directly. This could be lifted up into the spad code.

— **defun random 0** —

```
(defun |random| () (random (expt 2 26)))
```

66.12 IndexCard

66.12.1 defun IndexCard origin function support

```
[dbPart p??]  
[charPosition p??]  
[substring p??]
```

— defun alqlGetOrigin —

```
(defun |alqlGetOrigin| (x)  
  (let (field k)  
    (setq field (|dbPart| x 5 1))  
    (setq k (|charPosition| #\ ( field 2))  
    (substring field 1 (1- k))))
```

66.12.2 defun IndexCard origin function support

```
[dbPart p??]  
[charPosition p??]  
[substring p??]
```

— defun alqlGetParams —

```
(defun |alqlGetParams| (x)  
  (let (field k)  
    (setq field (|dbPart| x 5 1))  
    (setq k (|charPosition| #\ ( field 2))  
    (substring field k nil)))
```

66.12.3 defun IndexCard elt function support

```
[dbPart p??]  
[substring p??]
```

— **defun alqlGetKindString** —

```
(defun |alqlGetKindString| (x)
  (if (or (char= (elt x 0) #\a) (char= (elt x 0) #\o))
      (substring (|dbPart| x 5 1) 0 1)
      (substring x 0 1))))
```

—————

66.13 OperationsQuery

66.13.1 defun OperationQuery getDatabase function support

This function, called as `getBrowseDatabase(arg)` returns a list of appropriate entries in the browser database. The legal values for `arg` are

- “o” (operations)
- “k” (constructors)
- “d” (domains)
- “c” (categories)
- “p” (packages)

```
[member p1004]
[grepConstruct p??]
[$includeUnexposed? p??]
```

— **defun getBrowseDatabase** —

```
(defun |getBrowseDatabase| (kind)
  (let (|$includeUnexposed?|)
    (declare (special |$includeUnexposed?|))
    (setq |$includeUnexposed?| t)
    (when (|member| kind '("o" "k" "c" "d" "p"))
      (|grepConstruct| "*" (intern kind)))))
```

—————

66.14 Database

66.14.1 defun Database elt function support

[basicMatch? p??]

— defun stringMatches? —

```
(defun |stringMatches?| (pattern subject)
  (when (integerp (|basicMatch?| pattern subject)) t))
```

—————

66.15 FileName

66.15.1 defun FileName filename function implementation

[StringToDir p1027]

— defun fnameMake —

```
(defun |fnameMake| (d n e)
  (if (string= e "") (setq e nil))
  (make-pathname :directory (|StringToDir| d) :name n :type e))
```

—————

66.15.2 defun FileName filename support function

[lastc p??]

— defun StringToDir —

```
(defun |StringToDir| (s)
  (cond
    ((string= s "/" ) '(:root))
    ((string= s "") nil)
    (t
     (let ((lastc (aref s (- (length s) 1))))
       (if (char= lastc #\)
           (pathname-directory (concat s "name.type"))
           (pathname-directory (concat s "/name.type")) ))) ))
```

—————

66.15.3 defun FileName directory function implementation

[DirToString p1028]

— defun fnameDirectory —

```
(defun |fnameDirectory| (f)
  (|DirToString| (pathname-directory f)))
```

—————

66.15.4 defun FileName directory function support

For example, “/” “/u/smwatt” “../src”

— defun DirToString 0 —

```
(defun |DirToString| (d)
  (cond
    ((equal d '(:root)) "/")
    ((null d) "")
    ('t (string-right-trim "/" (namestring (make-pathname :directory d))))))
```

—————

66.15.5 defun FileName name function implementation

— defun fnameName 0 —

```
(defun |fnameName| (f)
  (let ((s (pathname-name f)))
    (if s s "")))
```

—————

66.15.6 defun FileName extension function implementation

— defun fnameType 0 —

```
(defun |fnameType| (f)
  (let ((s (pathname-type f)))
    (if s s "")))
```

66.15.7 defun FileName exists? function implementation

— defun fnameExists? 0 —

```
(defun |fnameExists?| (f)
  (if (probe-file (namestring f)) 't nil))
```

66.15.8 defun FileName readable? function implementation

— defun fnameReadable? 0 —

```
(defun |fnameReadable?| (f)
  (let ((s (open f :direction :input :if-does-not-exist nil)))
    (cond (s (close s) t) (t nil))))
```

66.15.9 defun FileName writeable? function implementation

```
[myWritable? p??]
```

— defun fnameWritable? —

```
(defun |fnameWritable?| (f)
  (|myWritable?| (namestring f)))
```

66.15.10 defun FileName writeable? function support

```
[error p??]
[fnameExists? p1029]
[fnameDirectory p1028]
[writeablep p??]
```

— defun myWritable? —

```
(defun |myWritable?| (s)
  (if (not (stringp s)) (|error| "'myWritable?' requires a string arg."))
  (if (string= s "") (setq s "."))
  (if (not (|fnameExists?| s)) (setq s (|fnameDirectory| s)))
  (if (string= s "") (setq s "."))
  (if (> (|writeablep| s) 0) 't nil) )
```

66.15.11 defun FileName new function implementation

[fnameMake p1027]

— defun fnameNew —

```
(defun |fnameNew| (d n e)
  (if (not (|myWritable?| d))
      nil
      (do ((fn))
          (nil)
          (setq fn (|fnameMake| d (string (gensym n)) e))
          (if (not (probe-file (namestring fn)))
              (return-from |fnameNew| fn) ) ) )
```

66.16 DoubleFloat

These macros wrap their arguments with strong type information in order to optimize doublefloat computations. They are used directly in the DoubleFloat domain (see Volume 10.3).

66.16.1 defmacro DFLessThan

Compute a strongly typed doublefloat comparison See Steele Common Lisp 1990 p293

— defmacro DFLessThan —

```
(defmacro DFLessThan (x y)
  '(< (the double-float ,x) (the double-float ,y)))
```

66.16.2 defmacro DFUnaryMinus

Compute a strongly typed unary doublefloat minus See Steele Common Lisp 1990 p295

— **defmacro DFUnaryMinus** —

```
(defmacro DFUnaryMinus (x)
  '(the double-float (- (the double-float ,x))))
```

—————

66.16.3 defmacro DFMinusp

Compute a strongly typed unary doublefloat test for negative See Steele Common Lisp 1990 p292

— **defmacro DFMinusp** —

```
(defmacro DFMinusp (x)
  '(minusp (the double-float ,x)))
```

—————

66.16.4 defmacro DFZerop

Compute a strongly typed unary doublefloat test for zero See Steele Common Lisp 1990 p292

— **defmacro DFZerop** —

```
(defmacro DFZerop (x)
  '(zerop (the double-float ,x)))
```

—————

66.16.5 defmacro DFAdd

Compute a strongly typed doublefloat addition See Steele Common Lisp 1990 p295

— **defmacro DFAdd** —

```
(defmacro DFAdd (x y)
  '(the double-float (+ (the double-float ,x) (the double-float ,y))))
```

—————

66.16.6 defmacro DFSubtract

Compute a strongly typed doublefloat subtraction See Steele Common Lisp 1990 p295

— **defmacro DFSubtract** —

```
(defmacro DFSubtract (x y)
  '(the double-float (- (the double-float ,x) (the double-float ,y))))
```

—————

66.16.7 defmacro DFMultiply

Compute a strongly typed doublefloat multiplication See Steele Common Lisp 1990 p296

— **defmacro DFMultiply** —

```
(defmacro DFMultiply (x y)
  '(the double-float (* (the double-float ,x) (the double-float ,y))))
```

—————

66.16.8 defmacro DFIntegerMultiply

Compute a strongly typed doublefloat multiplication by an integer. See Steele Common Lisp 1990 p296

— **defmacro DFIntegerMultiply** —

```
(defmacro DFIntegerMultiply (i y)
  '(the double-float (* (the integer ,i) (the double-float ,y))))
```

—————

66.16.9 defmacro DFMax

Choose the maximum of two doublefloats. See Steele Common Lisp 1990 p294

— **defmacro DFMax** —

```
(defmacro DFMax (x y)
  '(the double-float (max (the double-float ,x) (the double-float ,y))))
```

—————

66.16.10 defmacro DFMin

Choose the minimum of two doublefloats. See Steele Common Lisp 1990 p294

— **defmacro DFMin** —

```
(defmacro DFMin (x y)
  '(the double-float (min (the double-float ,x) (the double-float ,y))))
```

—————

66.16.11 defmacro DFEql

Compare two doublefloats for equality, where equality is eq, or numbers of the same type with the same value. See Steele Common Lisp 1990 p105

— **defmacro DFEql** —

```
(defmacro DFEql (x y)
  '(eq (the double-float ,x) (the double-float ,y)))
```

—————

66.16.12 defmacro DFDivide

Divide a doublefloat by a doublefloat See Steele Common Lisp 1990 p296

— **defmacro DFDivide** —

```
(defmacro DFDivide (x y)
  '(the double-float (/ (the double-float ,x) (the double-float ,y))))
```

—————

66.16.13 defmacro DFIntegerDivide

Divide a doublefloat by an integer See Steele Common Lisp 1990 p296

— **defmacro DFIntegerDivide** —

```
(defmacro DFIntegerDivide (x i)
  '(the double-float (/ (the double-float ,x) (the integer ,i))))
```

—————

66.16.14 defmacro DFSqrt

Compute the doublefloat square root of x . The result will be complex if the argument is negative. See Steele Common Lisp 1990 p302

— **defmacro DFSqrt** —

```
(defmacro DFSqrt (x)
  '(sqrt (the double-float ,x)))
```

66.16.15 defmacro DFLogE

Compute the doublefloat log of x with the base e . The result will be complex if the argument is negative. See Steele Common Lisp 1990 p301

— **defmacro DFLogE** —

```
(defmacro DFLogE (x)
  '(log (the double-float ,x)))
```

66.16.16 defmacro DFLog

Compute the doublefloat log of x with a given base b . The result will be complex if x is negative. See Steele Common Lisp 1990 p301

— **defmacro DFLog** —

```
(defmacro DFLog (x b)
  '(log (the double-float ,x) (the fixnum ,b)))
```

66.16.17 defmacro DFIntegerExpt

Compute the doublefloat expt of x with a given integer power i See Steele Common Lisp 1990 p300

— **defmacro DFIntegerExpt** —

```
(defmacro DFIntegerExpt (x i)
  '(the double-float (expt (the double-float ,x) (the integer ,i))))
```

66.16.18 defmacro DFExpt

Compute the doublefloat expt of x with a given power p . The result could be complex if the base is negative and the power is not an integer. See Steele Common Lisp 1990 p300

— **defmacro DFExpt** —

```
(defmacro DFExpt (x p)
  '(expt (the double-float ,x) (the double-float ,p)))
```

—————

66.16.19 defmacro DFExp

Compute the doublefloat exp with power e See Steele Common Lisp 1990 p300

— **defmacro DFExp** —

```
(defmacro DFExp (x)
  '(the double-float (exp (the double-float ,x))))
```

—————

66.16.20 defmacro DFSin

Compute a strongly typed doublefloat sin See Steele Common Lisp 1990 p304

— **defmacro DFSin** —

```
(defmacro DFSin (x)
  '(the double-float (sin (the double-float ,x))))
```

—————

66.16.21 defmacro DFCos

Compute a strongly typed doublefloat cos See Steele Common Lisp 1990 p304

— **defmacro DFCos** —

```
(defmacro DFCos (x)
  '(the double-float (cos (the double-float ,x))))
```

—————

66.16.22 defmacro DFTan

Compute a strongly typed doublefloat tan See Steele Common Lisp 1990 p304

— **defmacro DFTan** —

```
(defmacro DFTan (x)
  '(the double-float (tan (the double-float ,x))))
```

—————

66.16.23 defmacro DFAsin

Compute a strongly typed doublefloat asin. The result is complex if the absolute value of the argument is greater than 1. See Steele Common Lisp 1990 p305

— **defmacro DFAsin** —

```
(defmacro DFAsin (x)
  '(asin (the double-float ,x)))
```

—————

66.16.24 defmacro DFAcos

Compute a strongly typed doublefloat acos. The result is complex if the absolute value of the argument is greater than 1. See Steele Common Lisp 1990 p305

— **defmacro DFAcos** —

```
(defmacro DFAcos (x)
  '(acos (the double-float ,x)))
```

—————

66.16.25 defmacro DFAtan

Compute a strongly typed doublefloat atan See Steele Common Lisp 1990 p305

— **defmacro DFAtan** —

```
(defmacro DFAtan (x)
  '(the double-float (atan (the double-float ,x))))
```

—————

66.16.26 defmacro DFAtan2

Compute a strongly typed doublefloat atan with 2 arguments

$y = 0$	$x > 0$	Positive x-axis	0
$y > 0$	$x > 0$	Quadrant I	$0 < \text{result} < \pi/2$
$y > 0$	$x = 0$	Positive y-axis	$\pi/2$
$y > 0$	$x < 0$	Quadrant II	$\pi/2 < \text{result} < \pi$
$y = 0$	$x < 0$	Negative x-axis	π
$y < 0$	$x < 0$	Quadrant III	$-\pi < \text{result} < -\pi/2$
$y < 0$	$x = 0$	Negative y-axis	$-\pi/2$
$y < 0$	$x > 0$	Quadrant IV	$-\pi/2 < \text{result} < 0$
$y = 0$	$x = 0$	Origin	error

See Steele Common Lisp 1990 p306

— **defmacro DFAtan2** —

```
(defmacro DFAtan2 (y x)
  '(the double-float (atan (the double-float ,x) (the double-float ,y))))
```

—————

66.16.27 defmacro DFSinh

Compute a strongly typed doublefloat sinh

$$(e^z - e^{-z})/2$$

See Steele Common Lisp 1990 p308

— **defmacro DFSinh** —

```
(defmacro DFSinh (x)
  '(the double-float (sinh (the double-float ,x))))
```

—————

66.16.28 defmacro DFCosh

Compute a strongly typed doublefloat cosh

$$(e^z + e^{-z})/2$$

See Steele Common Lisp 1990 p308

— **defmacro DFCosh** —

```
(defmacro DFCosh (x)
  '(the double-float (cosh (the double-float ,x))))
```

66.16.29 defmacro DFTanh

Compute a strongly typed doublefloat tanh

$$(e^z - e^{-z}) / (e^z + e^{-z})$$

See Steele Common Lisp 1990 p308

— **defmacro DFTanh** —

```
(defmacro DFTanh (x)
  '(the double-float (tanh (the double-float ,x))))
```

66.16.30 defmacro DFAsinh

Compute the inverse hyperbolic sin.

$$\log \left(z + \sqrt{1 + z^2} \right)$$

See Steele Common Lisp 1990 p308

— **defmacro DFAsinh** —

```
(defmacro DFAsinh (x)
  '(the double-float (asinh (the double-float ,x))))
```

66.16.31 defmacro DFAcosh

Compute the inverse hyperbolic cos. Note that the acosh function will return a complex result if the argument is less than 1.

$$\log \left(z + (z + 1) \sqrt{(z - 1) / (z + 1)} \right)$$

See Steele Common Lisp 1990 p308

— **defmacro DFAcosh** —

```
(defmacro DFAcosh (x)
  '(acosh (the double-float ,x)))
```

66.16.32 defmacro DFAtanh

Compute the inverse hyperbolic tan. Note that the acosh function will return a complex result if the argument is greater than 1.

$$\log \left((1+z)\sqrt{1/(1-z^2)} \right)$$

See Steele Common Lisp 1990 p308

— **defmacro DFAtanh** —

```
(defmacro DFAtanh (x)
  '(atanh (the double-float ,x)))
```

66.16.33 defun Machine specific float numerator

This is used in the DoubleFloat integerDecode function

— **defun integer-decode-float-numerator 0** —

```
(defun integer-decode-float-numerator (x)
  (integer-decode-float x))
```

66.16.34 defun Machine specific float denominator

This is used in the DoubleFloat integerDecode function

— **defun integer-decode-float-denominator 0** —

```
(defun integer-decode-float-denominator (x)
  (multiple-value-bind (mantissa exponent sign) (integer-decode-float x)
    (declare (ignore mantissa sign)) (expt 2 (abs exponent)))))
```

66.16.35 defun Machine specific float sign

This is used in the DoubleFloat integerDecode function

— **defun integer-decode-float-sign 0** —

```
(defun integer-decode-float-sign (x)
  (multiple-value-bind (mantissa exponent sign) (integer-decode-float x)
    (declare (ignore mantissa exponent)) sign))
```

66.16.36 defun Machine specific float bit length

This is used in the DoubleFloat integerDecode function

— **defun integer-decode-float-exponent 0** —

```
(defun integer-decode-float-exponent (x)
  (multiple-value-bind (mantissa exponent sign) (integer-decode-float x)
    (declare (ignore mantissa sign)) exponent))
```

66.16.37 defun Decode floating-point values

This function is used by DoubleFloat to implement the “mantissa” and “exponent” functions.

— **defun manexp 0** —

```
(defun manexp (u)
  (multiple-value-bind (f e s)
    (decode-float u)
    (cons (* s f) e)))
```

66.16.38 defun The cotangent routine

The cotangent function is defined as

$$\cot(z) = \frac{1}{\tan(z)}$$

— **defun cot 0** —

```
(defun cot (a)
  (if (or (> a 1000.0) (< a -1000.0))
      (/ (cos a) (sin a))
      (/ 1.0 (tan a))))
```

66.16.39 defun The inverse cotangent function

The inverse cotangent (arc-cotangent) function is defined as

$$acot(z) = cot^{-1}(z) = tan^{-1}\left(\frac{1}{z}\right)$$

See Steele Common Lisp 1990 pp305-307

— **defun acot 0** —

```
(defun acot (a)
  (if (> a 0.0)
      (if (> a 1.0)
          (atan (/ 1.0 a))
          (- (/ pi 2.0) (atan a)))
      (if (< a -1.0)
          (- pi (atan (/ -1.0 a)))
          (+ (/ pi 2.0) (atan (- a))))))
```

66.16.40 defun The secant function

$$sec(x) = \frac{1}{cos(x)}$$

— **defun sec 0** —

```
(defun sec (x) (/ 1 (cos x)))
```

66.16.41 defun The inverse secant function

$$asec(x) = acos\left(\frac{1}{x}\right)$$

— **defun asec 0** —

```
(defun asec (x) (acos (/ 1 x)))
```

—————

66.16.42 **defun** The cosecant function

$$csc(x) = \frac{1}{sin(x)}$$

— **defun csc 0** —

```
(defun csc (x) (/ 1 (sin x)))
```

—————

66.16.43 **defun** The inverse cosecant function

$$acsc(x) = \frac{1}{asin(x)}$$

— **defun acsc 0** —

```
(defun acsc (x) (asin (/ 1 x)))
```

—————

66.16.44 **defun** The hyperbolic cosecant function

$$csch(x) = \frac{1}{sinh(x)}$$

— **defun csch 0** —

```
(defun csch (x) (/ 1 (sinh x)))
```

—————

66.16.45 defun The hyperbolic cotangent function

$$\coth(x) = \cosh(x) \operatorname{csch}(x)$$

— **defun coth 0** —

```
(defun coth (x) (* (cosh x) (csch x)))
```

—————

66.16.46 defun The hyperbolic secant function

$$\operatorname{sech}(x) = \frac{1}{\cosh(x)}$$

— **defun sech 0** —

```
(defun sech (x) (/ 1 (cosh x)))
```

—————

66.16.47 defun The inverse hyperbolic cosecant function

$$\operatorname{acsch}(x) = \operatorname{asinh}\left(\frac{1}{x}\right)$$

— **defun acsch 0** —

```
(defun acsch (x) (asinh (/ 1 x)))
```

—————

66.16.48 defun The inverse hyperbolic cotangent function

$$\operatorname{acoth}(x) = \operatorname{atanh}\left(\frac{1}{x}\right)$$

— **defun acoth 0** —

```
(defun acoth (x) (atanh (/ 1 x)))
```

66.16.49 defun The inverse hyperbolic secant function

$$asech(x) = acosh\left(\frac{1}{x}\right)$$

— defun asech 0 —

```
(defun asech (x) (acosh (/ 1 x)))
```

Chapter 67

NRLIB code.lisp support code

67.0.50 defun makeByteWordVec2

— defun makeByteWordVec2 0 —

```
(defun |makeByteWordVec2| (maxelement initialvalue)
  (let ((n (cond ((null initialvalue) 7) ('t maxelement))))
    (make-array (length initialvalue)
      :element-type (list 'mod (1+ n))
      :initial-contents initialvalue)))
```

—————

67.0.51 defmacro spadConstant

— defmacro spadConstant 0 —

```
(defmacro |spadConstant| (dollar n)
  '(spadcall (svref ,dollar (the fixnum ,n))))
```

—————

Chapter 68

Monitoring execution

MONITOR

This file contains a set of function for monitoring the execution of the functions in a file. It constructs a hash table that contains the function name as the key and monitor-data structures as the value

The technique is to use a :cond parameter on trace to call the monitor-incr function to incr the count every time a function is called

```
*monitor-table*                                HASH TABLE
  is the monitor table containing the hash entries
*monitor-nrlibs*                               LIST of STRING
  list of nrlib filenames that are monitored
*monitor-domains*                             LIST of STRING
  list of domains to monitor-report (default is all exposed domains)
monitor-data                                  STRUCTURE
  is the defstruct name of records in the table
  name is the first field and is the name of the monitored function
  count contains a count of times the function was called
  monitorp is a flag that skips counting if nil, counts otherwise
  sourcefile is the name of the file that contains the source code
```

***** SETUP, SHUTDOWN *****

```
monitor-inittable ()                           FUNCTION
  creates the hashtable and sets *monitor-table*
  note that it is called every time this file is loaded
monitor-end ()                                FUNCTION
  unhooks all of the trace hooks
```

***** TRACE, UNTRACE *****


```

monitor-add (name &optional sourcefile)      FUNCTION
    sets up the trace and adds the function to the table
monitor-delete (fn)                          FUNCTION
    untraces a function and removes it from the table
monitor-enable (&optional fn)                FUNCTION
    starts tracing for all (or optionally one) functions that
    are in the table
monitor-disable (&optional fn)               FUNCTION
    stops tracing for all (or optionally one) functions that
    are in the table

***** COUNTING, RECORDING *****

monitor-reset (&optional fn)                 FUNCTION
    reset the table count for the table (or optionally, for a function)
monitor-incr (fn)                            FUNCTION
    increments the count information for a function
    it is called by trace to increment the count
monitor-decr (fn)                            FUNCTION
    decrements the count information for a function
monitor-info (fn)                            FUNCTION
    returns the monitor-data structure for a function

***** FILE IO *****

monitor-write (items file)                   FUNCTION
    writes a list of symbols or structures to a file
monitor-file (file)                          FUNCTION
    will read a file, scan for defuns, monitor each defun
    NOTE: monitor-file assumes that the file has been loaded

***** RESULTS *****

monitor-results ()                           FUNCTION
    returns a list of the monitor-data structures
monitor-untested ()                          FUNCTION
    returns a list of files that have zero counts
monitor-tested (&optional delete)            FUNCTION
    returns a list of files that have nonzero counts
    optionally calling monitor-delete on those functions

***** CHECKPOINT/RESTORE *****

monitor-checkpoint (file)                    FUNCTION
    save the *monitor-table* in a loadable form
monitor-restore (file)                       FUNCTION
    restore a checkpointed file so that everything is monitored

***** ALGEBRA *****

monitor-autoload ()                          FUNCTION
    traces autoload of algebra to monitor corresponding source files

```

NOTE: this requires the /spad/int/algebra directory

monitor-dirname (args) FUNCTION
 expects a list of 1 libstream (loadvol's arglist) and monitors the source
 this is a function called by monitor-autoload

monitor-nrllib (nrllib) FUNCTION
 takes an nrllib name as a string (eg POLY) and returns a list of
 monitor-data structures from that source file

monitor-report () FUNCTION
 generate a report of the monitored activity for domains in
 monitor-domains

monitor-spadfile (name) FUNCTION
 given a spad file, report all nrllibs it creates
 this adds each nrllib name to *monitor-domains* but does not
 trace the functions from those domains

monitor-percent () FUNCTION
 ratio of (functions executed)/(functions traced)

monitor-apropos (str) FUNCTION
 given a string, find all monitored symbols containing the string
 the search is case-insensitive. returns a list of monitor-data items

for example:

suppose we have a file "/u/daly/testmon.lisp" that contains:

```
(defun foo1 () (print 'foo1))
(defun foo2 () (print 'foo2))
(defun foo3 () (foo1) (foo2) (print 'foo3))
(defun foo4 () (print 'foo4))
```

an example session is:

```
; FIRST WE LOAD THE FILE (WHICH INITIS *monitor-table*)
```

```
>(load "/u/daly/monitor.lisp")
Loading /u/daly/monitor.lisp
Finished loading /u/daly/monitor.lisp
T
```

```
; SECOND WE LOAD THE TESTMON FILE
```

```
>(load "/u/daly/testmon.lisp")
T
```

```
; THIRD WE MONITOR THE FILE
```

```
>(monitor-file "/u/daly/testmon.lisp")
monitoring "/u/daly/testmon.lisp"
NIL
```

```
; FOURTH WE CALL A FUNCTION FROM THE FILE (BUMP ITS COUNT)
```

```
>(foo1)
```

```
F001
```

```
F001
```

```
; AND ANOTHER FUNCTION (BUMP ITS COUNT)
>(foo2)

F002
F002

; AND A THIRD FUNCTION THAT CALLS THE OTHER TWO (BUMP ALL THREE)
>(foo3)

F001
F002
F003
F003

; CHECK THAT THE RESULTS ARE CORRECT

>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 2 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F002 COUNT 2 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F003 COUNT 1 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp"))
#S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp"))

; STOP COUNTING CALLS TO F002

>(monitor-disable 'foo2)
NIL

; INVOKE F002 THRU F003

>(foo3)

F001
F002
F003
F003

; NOTICE THAT F001 AND F003 WERE BUMPED BUT NOT F002
>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 3 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F002 COUNT 2 MONITORP NIL SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F003 COUNT 2 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp"))
#S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
```

```

"/u/daly/testmon.lisp"))

; TEMPORARILY STOP ALL MONITORING

>(monitor-disable)
NIL

; CHECK THAT NOTHING CHANGES

>(foo3)

F001
F002
F003
F003

; NO COUNT HAS CHANGED

>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 3 MONITORP NIL SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F002 COUNT 2 MONITORP NIL SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F003 COUNT 2 MONITORP NIL SOURCEFILE
    "/u/daly/testmon.lisp"))
 #S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp"))

; MONITOR ONLY CALLS TO F001

>(monitor-enable 'foo1)
T

; F003 CALLS F001

>(foo3)

F001
F002
F003
F003

; F001 HAS CHANGED BUT NOT F002 OR F003

>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 4 MONITORP T SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F002 COUNT 2 MONITORP NIL SOURCEFILE
    "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F003 COUNT 2 MONITORP NIL SOURCEFILE
    "/u/daly/testmon.lisp"))

```

```

        "/u/daly/testmon.lisp"))
#S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
   "/u/daly/testmon.lisp"))

; MONITOR EVERYBODY

>(monitor-enable)
NIL

; CHECK THAT EVERYBODY CHANGES

>(foo3)

F001
F002
F003
F003

; EVERYBODY WAS BUMPED

>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 5 MONITORP T SOURCEFILE
   "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F002 COUNT 3 MONITORP T SOURCEFILE
   "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F003 COUNT 3 MONITORP T SOURCEFILE
   "/u/daly/testmon.lisp"))
#S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
   "/u/daly/testmon.lisp"))

; WHAT FUNCTIONS WERE TESTED?

>(monitor-tested)
(F001 F002 F003)

; WHAT FUNCTIONS WERE NOT TESTED?

>(monitor-untested)
(F004)

; UNTRACE THE WHOLE WORLD, MONITORING CANNOT RESTART

>(monitor-end)
NIL

; CHECK THE RESULTS

>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 5 MONITORP T SOURCEFILE
   "/u/daly/testmon.lisp")

```

```

#S(MONITOR-DATA NAME F002 COUNT 3 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp")
#S(MONITOR-DATA NAME F003 COUNT 3 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp"))
#S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp"))

; CHECK THAT THE FUNCTIONS STILL WORK

>(foo3)

F001
F002
F003
F003

; CHECK THAT MONITORING IS NOT OCCURRING

>(monitor-results)
(#S(MONITOR-DATA NAME F001 COUNT 5 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F002 COUNT 3 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp")
 #S(MONITOR-DATA NAME F003 COUNT 3 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp"))
 #S(MONITOR-DATA NAME F004 COUNT 0 MONITORP T SOURCEFILE
  "/u/daly/testmon.lisp"))

```

68.0.52 defvar \$*monitor-domains*

— initvars —

```
(defvar *monitor-domains* nil "a list of domains to report")
```

68.0.53 defvar \$*monitor-nrlibs*

— initvars —

```
(defvar *monitor-nrlibs* nil "a list of nrlibs that have been traced")
```

68.0.54 defvar \$*monitor-table*

— initvars —

```
(defvar *monitor-table* nil "a table of all of the monitored data")
```

———

— postvars —

```
(eval-when (eval load)
  (unless *monitor-table* (monitor-inittable)))
```

———

68.0.55 defstruct \$monitor-data

— initvars —

```
(defstruct monitor-data name count monitorp sourcefile)
```

———

68.0.56 defstruct \$libstream

— initvars —

```
(defstruct libstream mode dirname (indextable nil) (indexstream nil))
```

———

68.0.57 defun Initialize the monitor statistics hashtable

```
[*monitor-table* p1054]
```

— defun monitor-inittable 0 —

```
(defun monitor-inittable ()
  "initialize the monitor statistics hashtable"
  (declare (special *monitor-table*))
  (setq *monitor-table* (make-hash-table)))
```

68.0.58 defun End the monitoring process, we cannot restart

[*monitor-table* p1054]

— defun monitor-end 0 —

```
(defun monitor-end ()
  "End the monitoring process. we cannot restart"
  (declare (special *monitor-table*))
  (maphash
   #'(lambda (key value)
       (declare (ignore value))
       (eval '(untrace ,key)))
   *monitor-table*))
```

68.0.59 defun Return a list of the monitor-data structures

[*monitor-table* p1054]

— defun monitor-results 0 —

```
(defun monitor-results ()
  "return a list of the monitor-data structures"
  (let (result)
    (declare (special *monitor-table*))
    (maphash
     #'(lambda (key value)
         (declare (ignore key))
         (push value result))
     *monitor-table*)
    (mapcar #'(lambda (x) (pprint x))
            (sort result #'string-lessp :key #'monitor-data-name))))
```

68.0.60 defun Add a function to be monitored

```
[monitor-delete p1056]
[make-monitor-data p??]
[*monitor-table* p1054]
```

— defun monitor-add 0 —

```
(defun monitor-add (name &optional sourcefile)
  "add a function to be monitored"
  (declare (special *monitor-table*))
  (unless (fboundp name) (load sourcefile))
  (when (gethash name *monitor-table*)
    (monitor-delete name))
  (eval '(trace (,name :cond (progn (monitor-incr ',name) nil))))
  (setf (gethash name *monitor-table*)
    (make-monitor-data
      :name name :count 0 :monitorp t :sourcefile sourcefile))))
```

—————

68.0.61 defun Remove a function being monitored

```
[*monitor-table* p1054]
```

— defun monitor-delete 0 —

```
(defun monitor-delete (fn)
  "Remove a function being monitored"
  (declare (special *monitor-table*))
  (eval '(untrace ,fn))
  (remhash fn *monitor-table*))
```

—————

68.0.62 defun Enable all (or optionally one) function for monitoring

```
[*monitor-table* p1054]
```

— defun monitor-enable 0 —

```
(defun monitor-enable (&optional fn)
  "enable all (or optionally one) function for monitoring"
  (declare (special *monitor-table*))
  (if fn
```

```

(progn
  (eval '(trace (,fn :cond (progn (monitor-incr ',fn) nil))))
  (setf (monitor-data-monitorp (gethash fn *monitor-table*)) t))
(maphash
 #'(lambda (key value)
   (declare (ignore value))
   (eval '(trace (,key :cond (progn (monitor-incr ',key) nil))))
   (setf (monitor-data-monitorp (gethash key *monitor-table*)) t))
 *monitor-table*))

```

68.0.63 defun Disable all (optionally one) function for monitoring

[*monitor-table* p1054]

— defun monitor-disable 0 —

```

(defun monitor-disable (&optional fn)
  "disable all (optionally one) function for monitoring"
  (declare (special *monitor-table*))
  (if fn
    (progn
      (eval '(untrace ,fn))
      (setf (monitor-data-monitorp (gethash fn *monitor-table*)) nil))
    (maphash
     #'(lambda (key value)
       (declare (ignore value))
       (eval '(untrace ,key))
       (setf (monitor-data-monitorp (gethash key *monitor-table*)) nil))
     *monitor-table*)))

```

68.0.64 defun Reset the table count for the table (or a function)

[*monitor-table* p1054]

— defun monitor-reset 0 —

```

(defun monitor-reset (&optional fn)
  "reset the table count for the table (or a function)"
  (declare (special *monitor-table*))
  (if fn
    (setf (monitor-data-count (gethash fn *monitor-table*)) 0)

```

```
(maphash
  #'(lambda (key value)
    (declare (ignore value))
    (setf (monitor-data-count (gethash key *monitor-table*)) 0))
  *monitor-table*))
```

68.0.65 defun Incr the count of fn by 1

```
[*monitor-table* p1054]
```

— defun monitor-incr 0 —

```
(defun monitor-incr (fn)
  "incr the count of fn by 1"
  (let (data)
    (declare (special *monitor-table*))
    (setq data (gethash fn *monitor-table*))
    (if data
      (incf (monitor-data-count data)) ;; change table entry by side-effect
      (warn "~s is monitored but not in table..do (untrace ~s)~%" fn fn)))
```

68.0.66 defun Decr the count of fn by 1

```
[*monitor-table* p1054]
```

— defun monitor-decr 0 —

```
(defun monitor-decr (fn)
  "decr the count of fn by 1"
  (let (data)
    (declare (special *monitor-table*))
    (setq data (gethash fn *monitor-table*))
    (if data
      (decf (monitor-data-count data)) ;; change table entry by side-effect
      (warn "~s is monitored but not in table..do (untrace ~s)~%" fn fn)))
```

68.0.67 defun Return the monitor information for a function

[*monitor-table* p1054]

— defun monitor-info 0 —

```
(defun monitor-info (fn)
  "return the monitor information for a function"
  (declare (special *monitor-table*))
  (gethash fn *monitor-table*))
```

—————

68.0.68 defun Hang a monitor call on all of the defuns in a file

```
[done p??]
[done p??]
[monitor-add p1056]
```

— defun monitor-file 0 —

```
(defun monitor-file (file)
  "hang a monitor call on all of the defuns in a file"
  (let (expr (package "BOOT"))
    (format t "monitoring ~s~%" file)
    (with-open-file (in file)
      (catch 'done
        (loop
          (setq expr (read in nil 'done))
          (when (eq expr 'done) (throw 'done nil))
          (if (and (consp expr) (eq (car expr) 'in-package))
              (if (and (consp (second expr)) (eq (first (second expr)) 'quote))
                  (setq package (string (second (second expr))))
                  (setq package (second expr)))
              (when (and (consp expr) (eq (car expr) 'defun))
                (monitor-add (intern (string (second expr)) package) file))))))))))
```

—————

68.0.69 defun Return a list of the functions with zero count fields

[*monitor-table* p1054]

— defun monitor-untested 0 —

```
(defun monitor-untested ()
  "return a list of the functions with zero count fields"
  (let (result)
    (declare (special *monitor-table*))
    (maphash
      #'(lambda (key value)
        (if (and (monitor-data-monitorp value) (= (monitor-data-count value) 0))
            (push key result)))
      *monitor-table*)
    (sort result #'string-lessp )))
```

68.0.70 defun Return a list of functions with non-zero counts

[monitor-delete p1056]
 [*monitor-table*] p??]

— defun monitor-tested 0 —

```
(defun monitor-tested (&optional delete)
  "return a list of functions with non-zero counts, optionally deleting them"
  (let (result)
    (declare (special *monitor-table*))
    (maphash
      #'(lambda (key value)
        (when (and (monitor-data-monitorp value)
                    (> (monitor-data-count value) 0))
          (when delete (monitor-delete key))
          (push key result)))
      *monitor-table*)
    (sort result #'string-lessp )))
```

68.0.71 defun Write out a list of symbols or structures to a file

— defun monitor-write 0 —

```
(defun monitor-write (items file)
  "write out a list of symbols or structures to a file"
  (with-open-file (out file :direction :output)
    (dolist (item items)
      (if (symbolp item)
```

```
(format out "~s~%" item)
(format out "~s~50t~s~100t~s~%"
  (monitor-data-sourcefile item)
  (monitor-data-name item)
  (monitor-data-count item))))))
```

68.0.72 defun Save the *monitor-table* in loadable form

```
[*monitor-table* p1054]
[*print-package* p??]
```

— defun monitor-checkpoint 0 —

```
(defun monitor-checkpoint (file)
  "save the *monitor-table* in loadable form"
  (let ((*print-package* t))
    (declare (special *print-package* *monitor-table*))
    (with-open-file (out file :direction :output)
      (format out "(in-package \"BOOT\")~%" )
      (format out "(monitor-inittable)~%" )
      (dolist (data (monitor-results))
        (format out "(monitor-add '~s ~s)~%"
          (monitor-data-name data)
          (monitor-data-sourcefile data))
        (format out "(setf (gethash '~s *monitor-table*)
          (make-monitor-data :name '~s :count ~s :monitorp ~s
            :sourcefile ~s))~%"
          (monitor-data-name data)
          (monitor-data-name data)
          (monitor-data-count data)
          (monitor-data-monitorp data)
          (monitor-data-sourcefile data))))))
```

68.0.73 defun restore a checkpointed file

— defun monitor-restore 0 —

```
(defun monitor-restore (file)
  "restore a checkpointed file"
  (load file))
```

68.0.74 defun Printing help documentation

— defun monitor-help 0 —

```
(defun monitor-help ()
  (format t "~%
  ;;; MONITOR
  ;;;
  ;;; This file contains a set of function for monitoring the execution
  ;;; of the functions in a file. It constructs a hash table that contains
  ;;; the function name as the key and monitor-data structures as the value
  ;;;
  ;;; The technique is to use a :cond parameter on trace to call the
  ;;; monitor-incr function to incr the count every time a function is called
  ;;;
  ;;; *monitor-table*                                HASH TABLE
  ;;;   is the monitor table containing the hash entries
  ;;; *monitor-nrlibs*                                LIST of STRING
  ;;;   list of nrllib filenames that are monitored
  ;;; *monitor-domains*                                LIST of STRING
  ;;;   list of domains to monitor-report (default is all exposed domains)
  ;;; monitor-data                                    STRUCTURE
  ;;;   is the defstruct name of records in the table
  ;;;   name is the first field and is the name of the monitored function
  ;;;   count contains a count of times the function was called
  ;;;   monitorp is a flag that skips counting if nil, counts otherwise
  ;;;   sourcefile is the name of the file that contains the source code
  ;;;
  ;;; ***** SETUP, SHUTDOWN *****
  ;;;
  ;;; monitor-inittable ()                                FUNCTION
  ;;;   creates the hashtable and sets *monitor-table*
  ;;;   note that it is called every time this file is loaded
  ;;; monitor-end ()                                FUNCTION
  ;;;   unhooks all of the trace hooks
  ;;;
  ;;; ***** TRACE, UNTRACE *****
  ;;;
  ;;; monitor-add (name &optional sourcefile)            FUNCTION
  ;;;   sets up the trace and adds the function to the table
  ;;; monitor-delete (fn)                                FUNCTION
  ;;;   untraces a function and removes it from the table
  ;;; monitor-enable (&optional fn)                    FUNCTION
  ;;;   starts tracing for all (or optionally one) functions that
  ;;;   are in the table
  ;;; monitor-disable (&optional fn)                    FUNCTION
```

```

;;; stops tracing for all (or optionally one) functions that
;;; are in the table
;;;
;;; ***** COUNTING, RECORDING *****
;;;
;;; monitor-reset (&optional fn)                FUNCTION
;;; reset the table count for the table (or optionally, for a function)
;;; monitor-incr (fn)                            FUNCTION
;;; increments the count information for a function
;;; it is called by trace to increment the count
;;; monitor-decr (fn)                            FUNCTION
;;; decrements the count information for a function
;;; monitor-info (fn)                            FUNCTION
;;; returns the monitor-data structure for a function
;;;
;;; ***** FILE IO *****
;;;
;;; monitor-write (items file)                   FUNCTION
;;; writes a list of symbols or structures to a file
;;; monitor-file (file)                         FUNCTION
;;; will read a file, scan for defuns, monitor each defun
;;; NOTE: monitor-file assumes that the file has been loaded
;;;
;;; ***** RESULTS *****
;;;
;;; monitor-results ()                          FUNCTION
;;; returns a list of the monitor-data structures
;;; monitor-untested ()                        FUNCTION
;;; returns a list of files that have zero counts
;;; monitor-tested (&optional delete)          FUNCTION
;;; returns a list of files that have nonzero counts
;;; optionally calling monitor-delete on those functions
;;;
;;; ***** CHECKPOINT/RESTORE *****
;;;
;;; monitor-checkpoint (file)                   FUNCTION
;;; save the *monitor-table* in a loadable form
;;; monitor-restore (file)                     FUNCTION
;;; restore a checkpointed file so that everything is monitored
;;;
;;; ***** ALGEBRA *****
;;;
;;; monitor-autoload ()                        FUNCTION
;;; traces autoload of algebra to monitor corresponding source files
;;; NOTE: this requires the /spad/int/algebra directory
;;; monitor-dirname (args)                     FUNCTION
;;; expects a list of 1 libstream (loadvol's arglist) and monitors the source
;;; this is a function called by monitor-autoload
;;; monitor-nrllib (nrllib)                   FUNCTION
;;; takes an nrllib name as a string (eg POLY) and returns a list of

```



```

;;; monitor-data structures from that source file
;;; monitor-report () FUNCTION
;;; generate a report of the monitored activity for domains in
;;; *monitor-domains*
;;; monitor-spadfile (name) FUNCTION
;;; given a spad file, report all nrlibs it creates
;;; this adds each nrlib name to *monitor-domains* but does not
;;; trace the functions from those domains
;;; monitor-percent () FUNCTION
;;; ratio of (functions executed)/(functions traced)
;;; monitor-apropos (str) FUNCTION
;;; given a string, find all monitored symbols containing the string
;;; the search is case-insensitive. returns a list of monitor-data items
") nil)

```

68.0.75 Monitoring algebra files

68.0.76 defun Monitoring algebra code.lsp files

[*monitor-nrlibs* p1053]

— defun monitor-dirname 0 —

```

(defun monitor-dirname (args)
  "expects a list of 1 libstream (loadvol's arglist) and monitors the source"
  (let (name)
    (declare (special *monitor-nrlibs*))
    (setq name (libstream-dirname (car args)))
    (setq name (file-namestring name))
    (setq name (concatenate 'string "/spad/int/algebra/" name "/code.lsp"))
    (when (probe-file name)
      (push name *monitor-nrlibs*)
      (monitor-file name))))

```

68.0.77 defun Monitor autoloading files

— defun monitor-autoload 0 —

```

(defun monitor-autoload ()

```

```
"traces autoload of algebra to monitor corresponding source files"
(trace (vmlisp::loadvol
      :entrycond nil
      :exitcond (progn (monitor-dirname system::arglist) nil))))
```

68.0.78 defun Monitor an nrlib

[*monitor-table* p1054]

— defun monitor-nrlib 0 —

```
(defun monitor-nrlib (nrlib)
  "takes an nrlib name as a string (eg POLY) and returns a list of
  monitor-data structures from that source file"
  (let (result)
    (declare (special *monitor-table*))
    (maphash
      #'(lambda (k v)
          (declare (ignore k))
          (when (string= nrlib
                        (pathname-name (car (last
                                           (pathname-directory (monitor-data-sourcefile v))))))
              (push v result))))
      *monitor-table*)
    result))
```

68.0.79 defun Given a monitor-data item, extract the nrlib name

— defun monitor-libname 0 —

```
(defun monitor-libname (item)
  "given a monitor-data item, extract the nrlib name"
  (pathname-name (car (last
                       (pathname-directory (monitor-data-sourcefile item))))))
```

68.0.80 defun Is this an exposed algebra function?

— defun monitor-exposedp 0 —

```
(defun monitor-exposedp (fn)
  "exposed functions have more than 1 semicolon. given a symbol, count them"
  (> (count #\; (symbol-name fn)) 1))
```

—————

68.0.81 defun Monitor exposed domains

TPDHERE: note that the file `interp.exposed` no longer exists. The exposure information is now in `bookvol5`. This needs to work off the internal exposure list, not the file.

```
[done p??]
[done p??]
[*monitor-domains* p1053]
```

— defun monitor-readinterp 0 —

```
(defun monitor-readinterp ()
  "read interp.exposed to initialize *monitor-domains* to exposed domains.
  this is the default action. adding or deleting domains from the list
  will change the report results"
  (let (skip expr name)
    (declare (special *monitor-domains*))
    (setq *monitor-domains* nil)
    (with-open-file (in "/spad/src/algebra/interp.exposed")
      (read-line in)
      (read-line in)
      (read-line in)
      (read-line in)
      (catch 'done
        (loop
          (setq expr (read-line in nil "done"))
          (when (string= expr "done") (throw 'done nil))
          (cond
            ((string= expr "basic") (setq skip nil))
            ((string= expr "categories") (setq skip t))
            ((string= expr "hidden") (setq skip t))
            ((string= expr "defaults") (setq skip nil)))
          (when (and (not skip) (> (length expr) 58))
            (setq name (subseq expr 58 (length expr)))
            (setq name (string-right-trim '("\space") name))
            (when (> (length name) 0)
              (push name *monitor-domains*))))))))))
```

68.0.82 defun Generate a report of the monitored domains

[monitor-readinterp p1066]
 [*monitor-domains* p1053]

— defun monitor-report 0 —

```
(defun monitor-report ()
  "generate a report of the monitored activity for domains in *monitor-domains*"
  (let (nrlibs nonzero total)
    (declare (special *monitor-domains*))
    (unless *monitor-domains* (monitor-readinterp))
    (setq nonzero 0)
    (setq total 0)
    (maphash
     #'(lambda (k v)
         (declare (ignore k))
         (let (nextlib point)
           (when (> (monitor-data-count v) 0) (incf nonzero))
           (incf total)
           (setq nextlib (monitor-libname v))
           (setq point (member nextlib nrlibs :test #'string= :key #'car))
           (if point
               (setf (cdr (first point)) (cons v (cdr (first point))))
               (push (cons nextlib (list v)) nrlibs))))
      *monitor-table*)
    (format t "~d of ~d (~d percent) tested~%" nonzero total
            (round (/ (* 100.0 nonzero) total)))
    (setq nrlibs (sort nrlibs #'string< :key #'car))
    (dolist (pair nrlibs)
      (let ((exposedcount 0) (testcount 0))
        (when (member (car pair) *monitor-domains* :test #'string=)
          (format t "for library ~s~%" (car pair))
          (dolist (item (sort (cdr pair) #'> :key #'monitor-data-count))
            (when (monitor-exposedp (monitor-data-name item))
              (incf exposedcount)
              (when (> (monitor-data-count item) 0) (incf testcount))
              (format t "~5d ~s~%"
                      (monitor-data-count item)
                      (monitor-data-name item))))
          (if (= exposedcount testcount)
              (format t "~a has all exposed functions tested~%" (car pair))
              (format t "Daly bug:~a has untested exposed functions~%" (car pair))))))
    nil))
```

68.0.83 defun Parse an)abbrev expression for the domain name

— defun monitor-parse 0 —

```
(defun monitor-parse (expr)
  (let (point1 point2)
    (setq point1 (position #\space expr :test #'char=))
    (setq point1 (position #\space expr :start point1 :test-not #'char=))
    (setq point1 (position #\space expr :start point1 :test #'char=))
    (setq point1 (position #\space expr :start point1 :test-not #'char=))
    (setq point2 (position #\space expr :start point1 :test #'char=))
    (subseq expr point1 point2)))
```

68.0.84 defun Given a spad file, report all nrlibs it creates

```
[done p??]
[done p??]
[monitor-parse p1068]
[*monitor-domains* p1053]
```

— defun monitor-spadfile 0 —

```
(defun monitor-spadfile (name)
  "given a spad file, report all nrlibs it creates"
  (let (expr)
    (declare (special *monitor-domains*))
    (with-open-file (in name)
      (catch 'done
        (loop
          (setq expr (read-line in nil 'done))
          (when (eq expr 'done) (throw 'done nil))
          (when (and (> (length expr) 4) (string= (subseq expr 0 4) ")abb"))
            (setq *monitor-domains*
                  (adjoin (monitor-parse expr) *monitor-domains* :test #'string=)))))))
```

68.0.85 defun Print percent of functions tested

[*monitor-table* p1054]

— defun monitor-percent 0 —

```

(defun monitor-percent ()
  "Print percent of functions tested"
  (let (nonzero total)
    (declare (special *monitor-table*))
    (setq nonzero 0)
    (setq total 0)
    (maphash
     #'(lambda (k v)
         (declare (ignore k))
         (when (> (monitor-data-count v) 0) (incf nonzero))
         (incf total))
      *monitor-table*)
    (format t "~d of ~d (~d percent) tested~%" nonzero total
            (round (/ (* 100.0 nonzero) total)))))

```

68.0.86 defun Find all monitored symbols containing the string

[*monitor-table* p1054]

— defun monitor-apropos 0 —

```

(defun monitor-apropos (str)
  "given a string, find all monitored symbols containing the string
  the search is case-insensitive. returns a list of monitor-data items"
  (let (result)
    (maphash
     #'(lambda (k v)
         (when
          (search (string-upcase str)
                  (string-upcase (symbol-name k))
                  :test #'string=)
          (push v result)))
      *monitor-table*)
    result))

```

Chapter 69

The Interpreter

— Interpreter —

```
(setq *print-array* nil)
(setq *print-circle* nil)
(setq *print-pretty* nil)

(in-package "BOOT")
\getchunk{initvars}

;;; level 0 macros

\getchunk{defmacro bit-to-truth 0}
\getchunk{defmacro bvec-elt 0}
\getchunk{defmacro idChar? 0}
\getchunk{defmacro identp 0}
\getchunk{defmacro qsabsval 0}
\getchunk{defmacro qsadd1 0}
\getchunk{defmacro qsdifference 0}
\getchunk{defmacro qslessp 0}
\getchunk{defmacro qsmax 0}
\getchunk{defmacro qsmin 0}
\getchunk{defmacro qsminus 0}
\getchunk{defmacro qsoddp 0}
\getchunk{defmacro qsplus 0}
\getchunk{defmacro qssub1 0}
\getchunk{defmacro qstimes 0}
\getchunk{defmacro qszerop 0}
\getchunk{defmacro spadConstant 0}

;;; above level 0 macros
```



```

\getchunk{defmacro assq}
\getchunk{defmacro bvec-setelt}
\getchunk{defmacro bvec-size}
\getchunk{defmacro cdaref2}
\getchunk{defmacro cdelt}
\getchunk{defmacro cdlen}
\getchunk{defmacro cdancols}
\getchunk{defmacro cdanrows}
\getchunk{defmacro cdsetaref2}
\getchunk{defmacro cdsetelt}
\getchunk{defmacro danrows}
\getchunk{defmacro dancols}
\getchunk{defmacro daref2}
\getchunk{defmacro delt}
\getchunk{defmacro DFAdd}
\getchunk{defmacro DFacos}
\getchunk{defmacro DFacosh}
\getchunk{defmacro DFasin}
\getchunk{defmacro DFasinh}
\getchunk{defmacro DFatan}
\getchunk{defmacro DFatan2}
\getchunk{defmacro DFatanh}
\getchunk{defmacro DFCos}
\getchunk{defmacro DFCosh}
\getchunk{defmacro DFDivide}
\getchunk{defmacro DFEql}
\getchunk{defmacro DFExp}
\getchunk{defmacro DFExpt}
\getchunk{defmacro DFIntegerDivide}
\getchunk{defmacro DFIntegerExpt}
\getchunk{defmacro DFIntegerMultiply}
\getchunk{defmacro DFLessThan}
\getchunk{defmacro DFLog}
\getchunk{defmacro DFLogE}
\getchunk{defmacro DFMax}
\getchunk{defmacro DFMin}
\getchunk{defmacro DFMinusp}
\getchunk{defmacro DFMultiply}
\getchunk{defmacro DFSin}
\getchunk{defmacro DFSinh}
\getchunk{defmacro DFSqrt}
\getchunk{defmacro DFSubtract}
\getchunk{defmacro DFTan}
\getchunk{defmacro DFTanh}
\getchunk{defmacro DFUnaryMinus}
\getchunk{defmacro DFZerop}
\getchunk{defmacro dlen}
\getchunk{defmacro dsetaref2}
\getchunk{defmacro dsetelt}
\getchunk{defmacro funfind}

```

```

\getchunk{defmacro hget}
\getchunk{defmacro make-cdouble-matrix}
\getchunk{defmacro make-cdouble-vector}
\getchunk{defmacro make-double-matrix}
\getchunk{defmacro make-double-matrix1}
\getchunk{defmacro make-double-vector}
\getchunk{defmacro make-double-vector1}
\getchunk{defmacro Rest}
\getchunk{defmacro startsId?}
\getchunk{defmacro trapNumericErrors}
\getchunk{defmacro truth-to-bit}
\getchunk{defmacro while}
\getchunk{defmacro whileWithResult}

;;; layer 0 (all common lisp)

\getchunk{defun acot 0}
\getchunk{defun acoth 0}
\getchunk{defun acsc 0}
\getchunk{defun acsch 0}
\getchunk{defun asec 0}
\getchunk{defun asech 0}
\getchunk{defun axiomVersion 0}

\getchunk{defun BooleanEquality 0}
\getchunk{defun bvec-and 0}
\getchunk{defun bvec-concat 0}
\getchunk{defun bvec-copy 0}
\getchunk{defun bvec-equal 0}
\getchunk{defun bvec-greater 0}
\getchunk{defun bvec-make-full 0}
\getchunk{defun bvec-nand 0}
\getchunk{defun bvec-nor 0}
\getchunk{defun bvec-not 0}
\getchunk{defun bvec-or 0}
\getchunk{defun bvec-xor 0}

\getchunk{defun cleanupLine 0}
\getchunk{defun clearMacroTable 0}
\getchunk{defun concat 0}
\getchunk{defun cot 0}
\getchunk{defun coth 0}
\getchunk{defun createCurrentInterpreterFrame 0}
\getchunk{defun credits 0}
\getchunk{defun csc 0}
\getchunk{defun csch 0}

\getchunk{defun Delay 0}
\getchunk{defun desiredMsg 0}
\getchunk{defun DirToString 0}

```

```

\getchunk{defun divide2 0}
\getchunk{defun dqAppend 0}
\getchunk{defun dqToList 0}
\getchunk{defun dqUnit 0}

\getchunk{defun emptyInterpreterFrame 0}

\getchunk{defun fin 0}
\getchunk{defun findFrameInRing 0}
\getchunk{defun flatten 0}
\getchunk{defun fnameExists? 0}
\getchunk{defun fnameName 0}
\getchunk{defun fnameReadable? 0}
\getchunk{defun fnameType 0}
\getchunk{defun frameExposureData 0}
\getchunk{defun frameHiFiAccess 0}
\getchunk{defun frameHistList 0}
\getchunk{defun frameHistListAct 0}
\getchunk{defun frameHistListLen 0}
\getchunk{defun frameHistoryTable 0}
\getchunk{defun frameHistRecord 0}
\getchunk{defun frameInteractive 0}
\getchunk{defun frameIOIndex 0}
\getchunk{defun frameName 0}
\getchunk{defun frameNames 0}
\getchunk{defun From 0}
\getchunk{defun FromTo 0}

\getchunk{defun get-current-directory 0}
\getchunk{defun getenviron 0}
\getchunk{defun getLinePos 0}
\getchunk{defun getLineText 0}
\getchunk{defun getMsgArgL 0}
\getchunk{defun getMsgKey 0}
\getchunk{defun getMsgKey? 0}
\getchunk{defun getMsgPrefix 0}
\getchunk{defun getMsgPosTagOb 0}
\getchunk{defun getMsgPrefix? 0}
\getchunk{defun getMsgTag 0}
\getchunk{defun getMsgTag? 0}
\getchunk{defun getMsgText 0}
\getchunk{defun getParserMacroNames 0}
\getchunk{defun getPreStL 0}

\getchunk{defun hasOptArgs? 0}

\getchunk{defun incActive? 0}
\getchunk{defun incCommand? 0}
\getchunk{defun incDrop 0}
\getchunk{defun incHandleMessage 0}

```

```

\getchunk{defun inclmsgConsole 0}
\getchunk{defun inclmsgFinSkipped 0}
\getchunk{defun inclmsgPrematureEOF 0}
\getchunk{defun inclmsgCmdBug 0}
\getchunk{defun inclmsgIfBug 0}
\getchunk{defun incPrefix? 0}
\getchunk{defun init-memory-config 0}
\getchunk{defun insertPos 0}
\getchunk{defun integer-decode-float-denominator 0}
\getchunk{defun integer-decode-float-exponent 0}
\getchunk{defun integer-decode-float-sign 0}
\getchunk{defun integer-decode-float-numerator 0}
\getchunk{defun intloopPrefix? 0}
\getchunk{defun isIntegerString 0}

\getchunk{defun keyword 0}
\getchunk{defun keyword? 0}

\getchunk{defun lfcomment 0}
\getchunk{defun lferror 0}
\getchunk{defun lffloat 0}
\getchunk{defun lfid 0}
\getchunk{defun lfinteger 0}
\getchunk{defun lfnegcomment 0}
\getchunk{defun lfrinteger 0}
\getchunk{defun lfspaces 0}
\getchunk{defun lfstring 0}
\getchunk{defun lnCreate 0}
\getchunk{defun lnExtraBlanks 0}
\getchunk{defun lnFileName? 0}
\getchunk{defun lnGlobalNum 0}
\getchunk{defun lnImmediate? 0}
\getchunk{defun lnLocalNum 0}
\getchunk{defun lnPlaceOfOrigin 0}
\getchunk{defun lnSetGlobalNum 0}
\getchunk{defun lnString 0}

\getchunk{defun mac0Define 0}
\getchunk{defun mac0InfiniteExpansion,name 0}
\getchunk{defun make-absolute-filename 0}
\getchunk{defun makeByteWordVec2 0}
\getchunk{defun makeInitialModemapFrame 0}
\getchunk{defun manexp 0}
\getchunk{defun member 0}
\getchunk{defun monitor-add 0}
\getchunk{defun monitor-apropos 0}
\getchunk{defun monitor-autoload 0}
\getchunk{defun monitor-checkpoint 0}
\getchunk{defun monitor-decr 0}
\getchunk{defun monitor-delete 0}

```

```

\getchunk{defun monitor-dirname 0}
\getchunk{defun monitor-disable 0}
\getchunk{defun monitor-enable 0}
\getchunk{defun monitor-end 0}
\getchunk{defun monitor-exposedp 0}
\getchunk{defun monitor-file 0}
\getchunk{defun monitor-help 0}
\getchunk{defun monitor-incr 0}
\getchunk{defun monitor-info 0}
\getchunk{defun monitor-inittable 0}
\getchunk{defun monitor-libname 0}
\getchunk{defun monitor-nrlib 0}
\getchunk{defun monitor-parse 0}
\getchunk{defun monitor-percent 0}
\getchunk{defun monitor-readinterp 0}
\getchunk{defun monitor-report 0}
\getchunk{defun monitor-reset 0}
\getchunk{defun monitor-restore 0}
\getchunk{defun monitor-results 0}
\getchunk{defun monitor-spadfile 0}
\getchunk{defun monitor-tested 0}
\getchunk{defun monitor-untested 0}
\getchunk{defun monitor-write 0}

\getchunk{defun ncError 0}
\getchunk{defun ncloopEscaped 0}
\getchunk{defun ncloopPrefix? 0}
\getchunk{defun ncloopPrintLines 0}
\getchunk{defun nonBlank 0}
\getchunk{defun npAnyNo 0}
\getchunk{defun npboot 0}
\getchunk{defun npEqPeek 0}
\getchunk{defun nplisp 0}
\getchunk{defun npPop1 0}
\getchunk{defun npPop2 0}
\getchunk{defun npPop3 0}
\getchunk{defun npPush 0}

\getchunk{defun opTran 0}

\getchunk{defun packageTran 0}
\getchunk{defun pfAndLeft 0}
\getchunk{defun pfAndRight 0}
\getchunk{defun pfAppend 0}
\getchunk{defun pfApplicationArg 0}
\getchunk{defun pfApplicationOp 0}
\getchunk{defun pfAssignLhsItems 0}
\getchunk{defun pf0AssignLhsItems 0}
\getchunk{defun pfAssignRhs 0}
\getchunk{defun pfBreakFrom 0}

```

```

\getchunk{defun pfCoercetoExpr 0}
\getchunk{defun pfCoercetoType 0}
\getchunk{defun pfCollectBody 0}
\getchunk{defun pfCollectIterators 0}
\getchunk{defun pfDefinitionLhsItems 0}
\getchunk{defun pfDefinitionRhs 0}
\getchunk{defun pfDoBody 0}
\getchunk{defun pfExitCond 0}
\getchunk{defun pfExitExpr 0}
\getchunk{defun pfFirst 0}
\getchunk{defun pfFreeItems 0}
\getchunk{defun pfForinLhs 0}
\getchunk{defun pfForinWhole 0}
\getchunk{defun pfFromdomDomain 0}
\getchunk{defun pfFromdomWhat 0}
\getchunk{defun pfIfCond 0}
\getchunk{defun pfIfElse 0}
\getchunk{defun pfIfThen 0}
\getchunk{defun pfLambdaArgs 0}
\getchunk{defun pfLambdaBody 0}
\getchunk{defun pfLambdaRets 0}
\getchunk{defun pfLiteral? 0}
\getchunk{defun pfLocalItems 0}
\getchunk{defun pfLoopIterators 0}
\getchunk{defun pfMacroLhs 0}
\getchunk{defun pfMacroRhs 0}
\getchunk{defun pfMLambdaArgs 0}
\getchunk{defun pfMLambdaBody 0}
\getchunk{defun pfNotArg 0}
\getchunk{defun pfNovalueExpr 0}
\getchunk{defun pfOrLeft 0}
\getchunk{defun pfOrRight 0}
\getchunk{defun pfParts 0}
\getchunk{defun pfPile 0}
\getchunk{defun pfPretendExpr 0}
\getchunk{defun pfPretendType 0}
\getchunk{defun pfRestrictExpr 0}
\getchunk{defun pfRestrictType 0}
\getchunk{defun pfReturnExpr 0}
\getchunk{defun pfRuleLhsItems 0}
\getchunk{defun pfRuleRhs 0}
\getchunk{defun pfSecond 0}
\getchunk{defun pfSequenceArgs 0}
\getchunk{defun pfSuchthatCond 0}
\getchunk{defun pfTaggedExpr 0}
\getchunk{defun pfTaggedTag 0}
\getchunk{defun pfTree 0}
\getchunk{defun pfTypedId 0}
\getchunk{defun pfTypedType 0}
\getchunk{defun pfTupleParts 0}

```

```

\getchunk{defun pfWhereContext 0}
\getchunk{defun pfWhereExpr 0}
\getchunk{defun pfWhileCond 0}
\getchunk{defun pmDontQuote? 0}
\getchunk{defun poCharPosn 0}
\getchunk{defun poGetLineObject 0}
\getchunk{defun poNopos? 0}
\getchunk{defun poNoPosition 0}
\getchunk{defun poNoPosition? 0}
\getchunk{defun printAsTeX 0}
\getchunk{defun pname 0}

\getchunk{defun qenum 0}
\getchunk{defun qsquotient 0}
\getchunk{defun qsremainder 0}
\getchunk{defun quotient2 0}

\getchunk{defun random 0}
\getchunk{defun rdigit? 0}
\getchunk{defun reclaim 0}
\getchunk{defun remainder2 0}
\getchunk{defun remLine 0}
\getchunk{defun rep 0}
\getchunk{defun resetStackLimits 0}

\getchunk{defun sameUnionBranch 0}
\getchunk{defun satisfiesUserLevel 0}
\getchunk{defun scanCloser? 0}
\getchunk{defun sec 0}
\getchunk{defun sech 0}
\getchunk{defun setCurrentLine 0}
\getchunk{defun setMsgPrefix 0}
\getchunk{defun setMsgText 0}
\getchunk{defun set-restart-hook 0}
\getchunk{defun showMsgPos? 0}
\getchunk{defun StreamNull 0}
\getchunk{defun stripLisp 0}
\getchunk{defun stripSpaces 0}

\getchunk{defun theid 0}
\getchunk{defun thefname 0}
\getchunk{defun theorigin 0}
\getchunk{defun tokPart 0}
\getchunk{defun To 0}
\getchunk{defun Top? 0}
\getchunk{defun trademark 0}

\getchunk{defun zeroOneTran 0}

;;; above level 0

```

```

\getchunk{defun abbQuery}
\getchunk{defun abbreviations}
\getchunk{defun abbreviationsSpad2Cmd}
\getchunk{defun addBinding}
\getchunk{defun addBindingInteractive}
\getchunk{defun addInputLibrary}
\getchunk{defun addNewInterpreterFrame}
\getchunk{defun addOperations}
\getchunk{defun addTraceItem}
\getchunk{defun allConstructors}
\getchunk{defun allOperations}
\getchunk{defun alqlGetOrigin}
\getchunk{defun alqlGetParams}
\getchunk{defun alqlGetKindString}
\getchunk{defun alreadyOpened?}
\getchunk{defun apropos}
\getchunk{defun assertCond}
\getchunk{defun augmentTraceNames}

\getchunk{defun break}
\getchunk{defun breaklet}
\getchunk{defun brightprint}
\getchunk{defun brightprint-0}
\getchunk{defun browse}
\getchunk{defun browseOpen}

\getchunk{defun cacheKeyedMsg}
\getchunk{defun categoryOpen}
\getchunk{defun changeHistListLen}
\getchunk{defun changeToNamedInterpreterFrame}
\getchunk{defun charDigitVal}
\getchunk{defun cleanline}
\getchunk{defun clear}
\getchunk{defun clearCmdAll}
\getchunk{defun clearCmdCompletely}
\getchunk{defun clearCmdExcept}
\getchunk{defun clearCmdParts}
\getchunk{defun clearCmdSortedCaches}
\getchunk{defun clearFrame}
\getchunk{defun clearParserMacro}
\getchunk{defun clearSpad2Cmd}
\getchunk{defun close}
\getchunk{defun closeInterpreterFrame}
\getchunk{defun coerceSpadArgs2E}
\getchunk{defun coerceSpadFunValue2E}
\getchunk{defun coerceTraceArgs2E}
\getchunk{defun coerceTraceFunValue2E}
\getchunk{defun commandAmbiguityError}
\getchunk{defun commandError}

```



```

\getchunk{defun commandErrorIfAmbiguous}
\getchunk{defun commandErrorMessage}
\getchunk{defun commandsForUserLevel}
\getchunk{defun commandUserLevelError}
\getchunk{defun compareposns}
\getchunk{defun compileBoot}
\getchunk{defun compressOpen}
\getchunk{defun constoken}
\getchunk{defun copyright}
\getchunk{defun countCache}

\getchunk{defun DaaseName}
\getchunk{defun decideHowMuch}
\getchunk{defun defiostream}
\getchunk{defun deldatabase}
\getchunk{defun deleteFile}
\getchunk{defun describe}
\getchunk{defun describeFortPersistence}
\getchunk{defun describeInputLibraryArgs}
\getchunk{defun describeOutputLibraryArgs}
\getchunk{defun describeProtectedSymbolsWarning}
\getchunk{defun describeProtectSymbols}
\getchunk{defun describeSetFortDir}
\getchunk{defun describeSetFortTmpDir}
\getchunk{defun describeSetFunctionsCache}
\getchunk{defun describeSetLinkerArgs}
\getchunk{defun describeSetNagHost}
\getchunk{defun describeSetOutputAlgebra}
\getchunk{defun describeSetOutputFormula}
\getchunk{defun describeSetOutputFortran}
\getchunk{defun describeSetOutputHtml}
\getchunk{defun describeSetOutputMathml}
\getchunk{defun describeSetOutputOpenMath}
\getchunk{defun describeSetOutputTex}
\getchunk{defun describeSetStreamsCalculate}
\getchunk{defun describeSpad2Cmd}
\getchunk{defun dewritify}
\getchunk{defun dewritify,dewritifyInner}
\getchunk{defun dewritify,is?}
\getchunk{defun diffAlist}
\getchunk{defun digit?}
\getchunk{defun digitp}
\getchunk{defun disableHist}
\getchunk{defun display}
\getchunk{defun displayCondition}
\getchunk{defun displayExposedConstructors}
\getchunk{defun displayExposedGroups}
\getchunk{defun displayFrameNames}
\getchunk{defun displayHiddenConstructors}
\getchunk{defun displayMacro}

```

```

\getchunk{defun displayMacros}
\getchunk{defun displayMode}
\getchunk{defun displayModemap}
\getchunk{defun displayOperations}
\getchunk{defun displayOperationsFromLisplib}
\getchunk{defun displayParserMacro}
\getchunk{defun displayProperties}
\getchunk{defun displayProperties,sayFunctionDeps}
\getchunk{defun displaySetOptionInformation}
\getchunk{defun displaySetVariableSettings}
\getchunk{defun displaySpad2Cmd}
\getchunk{defun displayType}
\getchunk{defun displayValue}
\getchunk{defun displayWorkspaceNames}
\getchunk{defun domainToGenvar}
\getchunk{defun doSystemCommand}
\getchunk{defun dqConcat}
\getchunk{defun dropInputLibrary}
\getchunk{defun dumbTokenize}

\getchunk{defun edit}
\getchunk{defun editFile}
\getchunk{defun editSpad2Cmd}
\getchunk{defun Else?}
\getchunk{defun Elseif?}
\getchunk{defun enPile}
\getchunk{defun eofp}
\getchunk{defun eqpileTree}
\getchunk{defun erMsgCompare}
\getchunk{defun erMsgSep}
\getchunk{defun erMsgSort}
\getchunk{defun ExecuteInterpSystemCommand}
\getchunk{defun executeQuietCommand}

\getchunk{defun fetchKeyedMsg}
\getchunk{defun fetchOutput}
\getchunk{defun fillerSpaces}
\getchunk{defun filterAndFormatConstructors}
\getchunk{defun filterListOfStrings}
\getchunk{defun filterListOfStringsWithFn}
\getchunk{defun firstTokPosn}
\getchunk{defun fixObjectForPrinting}
\getchunk{defun flattenOperationAlist}
\getchunk{defun float2Sex}
\getchunk{defun fnameDirectory}
\getchunk{defun fnameMake}
\getchunk{defun fnameNew}
\getchunk{defun fnameWritable?}
\getchunk{defun frame}
\getchunk{defun frameEnvironment}

```

```

\getchunk{defun frameSpad2Cmd}
\getchunk{defun functionp}
\getchunk{defun funfind,LAM}

\getchunk{defun genDomainTraceName}
\getchunk{defun gensymInt}
\getchunk{defun getAliasIfTracedMapParameter}
\getchunk{defun getAndSay}
\getchunk{defun getBpiNameIfTracedMap}
\getchunk{defun getBrowseDatabase}
\getchunk{defun getdatabase}
\getchunk{defun getDirectoryList}
\getchunk{defun getFirstWord}
\getchunk{defun getKeyedMsg}
\getchunk{defun getMapSig}
\getchunk{defun getMapSubNames}
\getchunk{defun getMsgCatAttr}
\getchunk{defun getMsgFTTag?}
\getchunk{defun getMsgInfoFromKey}
\getchunk{defun getMsgLitSym}
\getchunk{defun getMsgPos}
\getchunk{defun getMsgPos2}
\getchunk{defun getMsgToWhere}
\getchunk{defun getOption}
\getchunk{defun getPosStL}
\getchunk{defun getPreviousMapSubNames}
\getchunk{defun getProplist}
\getchunk{defun getStFromMsg}
\getchunk{defun getSystemCommandLine}
\getchunk{defun getTraceOption}
\getchunk{defun getTraceOption,hn}
\getchunk{defun getTraceOptions}
\getchunk{defun getWorkspaceNames}

\getchunk{defun handleNoParseCommands}
\getchunk{defun handleParsedSystemCommands}
\getchunk{defun handleTokenSizeSystemCommands}
\getchunk{defun hashable}
\getchunk{defun hasOption}
\getchunk{defun hasPair}
\getchunk{defun help}
\getchunk{defun helpSpad2Cmd}
\getchunk{defun histFileErase}
\getchunk{defun histFileName}
\getchunk{defun histInputFileName}
\getchunk{defun history}
\getchunk{defun historySpad2Cmd}
\getchunk{defun hkeys}
\getchunk{defun hput}

```

```

\getchunk{defun If?}
\getchunk{defun ifCond}
\getchunk{defun importFromFrame}
\getchunk{defun incAppend}
\getchunk{defun incAppend1}
\getchunk{defun incBiteOff}
\getchunk{defun incClassify}
\getchunk{defun incCommandTail}
\getchunk{defun incConsoleInput}
\getchunk{defun incFileInput}
\getchunk{defun incFileName}
\getchunk{defun incIgen}
\getchunk{defun incIgen1}
\getchunk{defun inclFname}
\getchunk{defun incLine}
\getchunk{defun incLine1}
\getchunk{defun inclmsgCannotRead}
\getchunk{defun inclmsgFileCycle}
\getchunk{defun inclmsgPrematureFin}
\getchunk{defun include}
\getchunk{defun include1}
\getchunk{defun inclmsgConActive}
\getchunk{defun inclmsgConStill}
\getchunk{defun inclmsgIfSyntax}
\getchunk{defun inclmsgNoSuchFile}
\getchunk{defun inclmsgSay}
\getchunk{defun incNConsoles}
\getchunk{defun incRenumber}
\getchunk{defun incRenumberItem}
\getchunk{defun incRenumberLine}
\getchunk{defun incRgen}
\getchunk{defun incRgen1}
\getchunk{defun incStream}
\getchunk{defun incString}
\getchunk{defun incZip}
\getchunk{defun incZip1}
\getchunk{defun init-boot/spad-reader}
\getchunk{defun initHist}
\getchunk{defun initHistList}
\getchunk{defun initial-getdatabase}
\getchunk{defun initializeInterpreterFrameRing}
\getchunk{defun initializeSetVariables}
\getchunk{defun initImPr}
\getchunk{defun initroot}
\getchunk{defun initToWhere}
\getchunk{defun insertpile}
\getchunk{defun InterpExecuteSpadSystemCommand}
\getchunk{defun interpFunctionDepAlists}
\getchunk{defun interpOpen}
\getchunk{defun interpret}

```

```

\getchunk{defun interpret1}
\getchunk{defun interpret2}
\getchunk{defun interpretTopLevel}
\getchunk{defun intInterpretPform}
\getchunk{defun intloop}
\getchunk{defun intloopEchoParse}
\getchunk{defun intloopInclude}
\getchunk{defun intloopInclude0}
\getchunk{defun intnplisp}
\getchunk{defun intloopProcess}
\getchunk{defun intloopProcessString}
\getchunk{defun intloopReadConsole}
\getchunk{defun intloopSpadProcess}
\getchunk{defun intloopSpadProcess,interp}
\getchunk{defun intProcessSynonyms}
\getchunk{defun intSayKeyedMsg}
\getchunk{defun ioclear}
\getchunk{defun isDomainOrPackage}
\getchunk{defun isgenvar}
\getchunk{defun isInterpOnlyMap}
\getchunk{defun isListOfIdentifiers}
\getchunk{defun isListOfIdentifiersOrStrings}
\getchunk{defun isSharpVar}
\getchunk{defun isSharpVarWithNum}
\getchunk{defun isSubForRedundantMapName}
\getchunk{defun isTraceGensym}
\getchunk{defun isUncompiledMap}

\getchunk{defun justifyMyType}

\getchunk{defun KeepPart?}

\getchunk{defun lassocSub}
\getchunk{defun lastTokPosn}
\getchunk{defun leader?}
\getchunk{defun leaveScratchpad}
\getchunk{defun letPrint}
\getchunk{defun letPrint2}
\getchunk{defun letPrint3}
\getchunk{defun lfkey}
\getchunk{defun library}
\getchunk{defun line?}
\getchunk{defun lineoftoks}
\getchunk{defun listConstructorAbbreviations}
\getchunk{defun listDecideHowMuch}
\getchunk{defun listOutputter}
\getchunk{defun lnFileName}
\getchunk{defun load}
\getchunk{defun localdatabase}
\getchunk{defun localnrlib}

```

```

\getchunk{defun loopIters2Sex}
\getchunk{defun lotsof}
\getchunk{defun ltrace}

\getchunk{defun macApplication}
\getchunk{defun macExpand}
\getchunk{defun macId}
\getchunk{defun macLambda}
\getchunk{defun macLambda,mac}
\getchunk{defun macLambdaParameterHandling}
\getchunk{defun macMacro}
\getchunk{defun macSubstituteId}
\getchunk{defun macSubstituteOuter}
\getchunk{defun macroExpanded}
\getchunk{defun macWhere}
\getchunk{defun macWhere,mac}
\getchunk{defun macOExpandBody}
\getchunk{defun macOGet}
\getchunk{defun macOGetName}
\getchunk{defun macOInfiniteExpansion}
\getchunk{defun macOMLambdaApply}
\getchunk{defun macOSubstituteOuter}
\getchunk{defun make-appendstream}
\getchunk{defun make-databases}
\getchunk{defun makeFullNamestring}
\getchunk{defun makeHistFileName}
\getchunk{defun makeInputFilename}
\getchunk{defun make-instream}
\getchunk{defun makeLeaderMsg}
\getchunk{defun makeMsgFromLine}
\getchunk{defun make-outstream}
\getchunk{defun makePathname}
\getchunk{defun makeStream}
\getchunk{defun mapLetPrint}
\getchunk{defun mergePathnames}
\getchunk{defun messageprint}
\getchunk{defun messageprint-1}
\getchunk{defun messageprint-2}
\getchunk{defun mkLineList}
\getchunk{defun mkprompt}
\getchunk{defun msgCreate}
\getchunk{defun msgImPr?}
\getchunk{defun msgNoRep?}
\getchunk{defun msgOutputter}
\getchunk{defun msgText}
\getchunk{defun myWritable?}

\getchunk{defun namestring}
\getchunk{defun ncAlist}
\getchunk{defun ncBug}

```

```

\getchunk{defun ncConversationPhase}
\getchunk{defun ncConversationPhase,wrapup}
\getchunk{defun ncEltQ}
\getchunk{defun ncHardError}
\getchunk{defun ncIntLoop}
\getchunk{defun ncloopCommand}
\getchunk{defun ncloopDQlines}
\getchunk{defun ncloopIncFileName}
\getchunk{defun ncloopInclude}
\getchunk{defun ncloopInclude0}
\getchunk{defun ncloopInclude1}
\getchunk{defun ncloopParse}
\getchunk{defun ncParseAndInterpretString}
\getchunk{defun ncPutQ}
\getchunk{defun ncSoftError}
\getchunk{defun ncTag}
\getchunk{defun ncTopLevel}
\getchunk{defun newHelpSpad2Cmd}
\getchunk{defun next}
\getchunk{defun next1}
\getchunk{defun nextInterpreterFrame}
\getchunk{defun nextline}
\getchunk{defun next-lines-clear}
\getchunk{defun npAdd}
\getchunk{defun npADD}
\getchunk{defun npAmpersand}
\getchunk{defun npAmpersandFrom}
\getchunk{defun npAndOr}
\getchunk{defun npAngleBared}
\getchunk{defun npApplication}
\getchunk{defun npApplication2}
\getchunk{defun npArith}
\getchunk{defun npAssign}
\getchunk{defun npAssignment}
\getchunk{defun npAssignVariable}
\getchunk{defun npAtom1}
\getchunk{defun npAtom2}
\getchunk{defun npBacksetElse}
\getchunk{defun npBackTrack}
\getchunk{defun npBDefinition}
\getchunk{defun npBPileDefinition}
\getchunk{defun npBraced}
\getchunk{defun npBracked}
\getchunk{defun npBracketed}
\getchunk{defun npBreak}
\getchunk{defun npBy}
\getchunk{defun npCategory}
\getchunk{defun npCategoryL}
\getchunk{defun npCoerceTo}
\getchunk{defun npColon}

```

```

\getchunk{defun npColonQuery}
\getchunk{defun npComma}
\getchunk{defun npCommaBackSet}
\getchunk{defun npCompMissing}
\getchunk{defun npConditional}
\getchunk{defun npConditionalStatement}
\getchunk{defun npConstTok}
\getchunk{defun npDDInfKey}
\getchunk{defun npDecl}
\getchunk{defun npDef}
\getchunk{defun npDefaultDecl}
\getchunk{defun npDefaultItem}
\getchunk{defun npDefaultItemList}
\getchunk{defun npDefaultValue}
\getchunk{defun npDefinition}
\getchunk{defun npDefinitionItem}
\getchunk{defun npDefinitionlist}
\getchunk{defun npDefinitionOrStatement}
\getchunk{defun npDefn}
\getchunk{defun npDefTail}
\getchunk{defun npDiscrim}
\getchunk{defun npDisjand}
\getchunk{defun npDollar}
\getchunk{defun npDotted}
\getchunk{defun npElse}
\getchunk{defun npEncAp}
\getchunk{defun npEncl}
\getchunk{defun npEnclosed}
\getchunk{defun npEqKey}
\getchunk{defun npExit}
\getchunk{defun npExpress}
\getchunk{defun npExpress1}
\getchunk{defun npExport}
\getchunk{defun npFirstTok}
\getchunk{defun npFix}
\getchunk{defun npForIn}
\getchunk{defun npFree}
\getchunk{defun npFromdom}
\getchunk{defun npFromdom1}
\getchunk{defun npGives}
\getchunk{defun npId}
\getchunk{defun npImport}
\getchunk{defun npInfGeneric}
\getchunk{defun npInfixOp}
\getchunk{defun npInfixOperator}
\getchunk{defun npInfKey}
\getchunk{defun npInline}
\getchunk{defun npInterval}
\getchunk{defun npItem}
\getchunk{defun npItem1}

```



```

\getchunk{defun npIterate}
\getchunk{defun npIterator}
\getchunk{defun npIterators}
\getchunk{defun npLambda}
\getchunk{defun npLeftAssoc}
\getchunk{defun npLet}
\getchunk{defun npLetQualified}
\getchunk{defun npList}
\getchunk{defun npListAndRecover}
\getchunk{defun npListing}
\getchunk{defun npListofFun}
\getchunk{defun npLocal}
\getchunk{defun npLocalDecl}
\getchunk{defun npLocalItem}
\getchunk{defun npLocalItemList}
\getchunk{defun npLogical}
\getchunk{defun npLoop}
\getchunk{defun npMacro}
\getchunk{defun npMatch}
\getchunk{defun npMdef}
\getchunk{defun npMDEF}
\getchunk{defun npMDEFinition}
\getchunk{defun npMissing}
\getchunk{defun npMissingMate}
\getchunk{defun npMoveTo}
\getchunk{defun npName}
\getchunk{defun npNext}
\getchunk{defun npNull}
\getchunk{defun npParened}
\getchunk{defun npParenthesize}
\getchunk{defun npParenthesized}
\getchunk{defun npParse}
\getchunk{defun npPDefinition}
\getchunk{defun npPileBracketed}
\getchunk{defun npPileDefinitionlist}
\getchunk{defun npPileExit}
\getchunk{defun npPower}
\getchunk{defun npPP}
\getchunk{defun npPPf}
\getchunk{defun npPPff}
\getchunk{defun npPPg}
\getchunk{defun npPrefixColon}
\getchunk{defun npPretend}
\getchunk{defun npPrimary}
\getchunk{defun npPrimary1}
\getchunk{defun npPrimary2}
\getchunk{defun npProcessSynonym}
\getchunk{defun npProduct}
\getchunk{defun npPushId}
\getchunk{defun npRelation}

```

```

\getchunk{defun npRemainder}
\getchunk{defun npQualDef}
\getchunk{defun npQualified}
\getchunk{defun npQualifiedDefinition}
\getchunk{defun npQualType}
\getchunk{defun npQualTypelist}
\getchunk{defun npQuiver}
\getchunk{defun npRecoverTrap}
\getchunk{defun npRestore}
\getchunk{defun npRestrict}
\getchunk{defun npReturn}
\getchunk{defun npRightAssoc}
\getchunk{defun npRule}
\getchunk{defun npSCategory}
\getchunk{defun npSDefaultItem}
\getchunk{defun npSegment}
\getchunk{defun npSelector}
\getchunk{defun npSemiBackSet}
\getchunk{defun npSemiListing}
\getchunk{defun npSigDecl}
\getchunk{defun npSigItem}
\getchunk{defun npSigItemList}
\getchunk{defun npSignature}
\getchunk{defun npSignatureDefinee}
\getchunk{defun npSingleRule}
\getchunk{defun npSLocalItem}
\getchunk{defun npSQualTypelist}
\getchunk{defun npStatement}
\getchunk{defun npSuch}
\getchunk{defun npSuchThat}
\getchunk{defun npSum}
\getchunk{defun npsynonym}
\getchunk{defun npSymbolVariable}
\getchunk{defun npSynthetic}
\getchunk{defun npsystem}
\getchunk{defun npState}
\getchunk{defun npTagged}
\getchunk{defun npTerm}
\getchunk{defun npTrap}
\getchunk{defun npTrapForm}
\getchunk{defun npTuple}
\getchunk{defun npType}
\getchunk{defun npTypedForm}
\getchunk{defun npTypedForm1}
\getchunk{defun npTypeStyle}
\getchunk{defun npTypified}
\getchunk{defun npTyping}
\getchunk{defun npTypeVariable}
\getchunk{defun npTypeVariablelist}
\getchunk{defun npVariable}

```

```

\getchunk{defun npVariablelist}
\getchunk{defun npVariableName}
\getchunk{defun npVoid}
\getchunk{defun npWConditional}
\getchunk{defun npWhile}
\getchunk{defun npWith}
\getchunk{defun npZeroOrMore}

\getchunk{defun oldHistFileName}
\getchunk{defun openOutputLibrary}
\getchunk{defun openserver}
\getchunk{defun operationOpen}
\getchunk{defun optionError}
\getchunk{defun optionUserLevelError}
\getchunk{defun orderBySlotNumber}

\getchunk{defun parseAndInterpret}
\getchunk{defun parseFromString}
\getchunk{defun parseSystemCmd}
\getchunk{defun pathname}
\getchunk{defun pathnameDirectory}
\getchunk{defun pathnameName}
\getchunk{defun pathnameType}
\getchunk{defun pathnameTypeId}
\getchunk{defun patternVarsOf}
\getchunk{defun patternVarsOf1}
\getchunk{defun pcounters}
\getchunk{defun pfAbSynOp}
\getchunk{defun pfAbSynOp?}
\getchunk{defun pfAdd}
\getchunk{defun pfAnd}
\getchunk{defun pfAnd?}
\getchunk{defun pfApplication}
\getchunk{defun pfApplication?}
\getchunk{defun pfApplication2Sex}
\getchunk{defun pfAssign}
\getchunk{defun pfAssign?}
\getchunk{defun pfAttribute}
\getchunk{defun pfBrace}
\getchunk{defun pfBraceBar}
\getchunk{defun pfBracket}
\getchunk{defun pfBracketBar}
\getchunk{defun pfBreak}
\getchunk{defun pfBreak?}
\getchunk{defun pfCharPosn}
\getchunk{defun pfCheckArg}
\getchunk{defun pfCheckMacroOut}
\getchunk{defun pfCheckId}
\getchunk{defun pfCheckItOut}
\getchunk{defun pfCoerceto}

```

```

\getchunk{defun pfCoerceto?}
\getchunk{defun pfCollect}
\getchunk{defun pfCollect?}
\getchunk{defun pfCollect1?}
\getchunk{defun pfCollectArgTran}
\getchunk{defun pfCollectVariable1}
\getchunk{defun pfCollect2Sex}
\getchunk{defun pfCopyWithPos}
\getchunk{defun pfDefinition}
\getchunk{defun pfDefinition?}
\getchunk{defun pfDefinition2Sex}
\getchunk{defun pfDo}
\getchunk{defun pfDo?}
\getchunk{defun pfDocument}
\getchunk{defun pfEnSequence}
\getchunk{defun pfExit}
\getchunk{defun pfExit?}
\getchunk{defun pfExport}
\getchunk{defun pfExpression}
\getchunk{defun pfFileName}
\getchunk{defun pfFix}
\getchunk{defun pfFlattenApp}
\getchunk{defun pfFree}
\getchunk{defun pfFree?}
\getchunk{defun pfForin}
\getchunk{defun pfForin?}
\getchunk{defun pfFromDom}
\getchunk{defun pfFromdom}
\getchunk{defun pfFromdom?}
\getchunk{defun pfGlobalLinePosn}
\getchunk{defun pfHide}
\getchunk{defun pfId}
\getchunk{defun pfId?}
\getchunk{defun pfIdPos}
\getchunk{defun pfIdSymbol}
\getchunk{defun pfIf}
\getchunk{defun pfIf?}
\getchunk{defun pfIfThenOnly}
\getchunk{defun pfImport}
\getchunk{defun pfInline}
\getchunk{defun pfInfApplication}
\getchunk{defun pfIterate}
\getchunk{defun pfIterate?}
\getchunk{defun pfLam}
\getchunk{defun pfLambda}
\getchunk{defun pfLambdaTran}
\getchunk{defun pfLambda?}
\getchunk{defun pfLambda2Sex}
\getchunk{defun pfLeaf}
\getchunk{defun pfLeaf?}

```

```

\getchunk{defun pfLeafPosition}
\getchunk{defun pfLeafToken}
\getchunk{defun pfLhsRule2Sex}
\getchunk{defun pfLinePosn}
\getchunk{defun pfListOf}
\getchunk{defun pfLiteralClass}
\getchunk{defun pfLiteralString}
\getchunk{defun pfLiteral2Sex}
\getchunk{defun pfLocal}
\getchunk{defun pfLocal?}
\getchunk{defun pfLoop}
\getchunk{defun pfLoop1}
\getchunk{defun pfLoop?}
\getchunk{defun pfLp}
\getchunk{defun pfMacro}
\getchunk{defun pfMacro?}
\getchunk{defun pfMapParts}
\getchunk{defun pfMLambda}
\getchunk{defun pfMLambda?}
\getchunk{defun pfname}
\getchunk{defun pfNoPosition}
\getchunk{defun pfNoPosition?}
\getchunk{defun pfNot?}
\getchunk{defun pfNothing}
\getchunk{defun pfNothing?}
\getchunk{defun pfNovalue}
\getchunk{defun pfNovalue?}
\getchunk{defun pfOp2Sex}
\getchunk{defun pfOr}
\getchunk{defun pfOr?}
\getchunk{defun pfParen}
\getchunk{defun pfPretend}
\getchunk{defun pfPretend?}
\getchunk{defun pfPushBody}
\getchunk{defun pfPushMacroBody}
\getchunk{defun pfQualType}
\getchunk{defun pfRestrict}
\getchunk{defun pfRestrict?}
\getchunk{defun pfRetractTo}
\getchunk{defun pfReturn}
\getchunk{defun pfReturn?}
\getchunk{defun pfReturnNoName}
\getchunk{defun pfReturnTyped}
\getchunk{defun pfRhsRule2Sex}
\getchunk{defun pfRule}
\getchunk{defun pfRule?}
\getchunk{defun pfRule2Sex}
\getchunk{defun pfSequence}
\getchunk{defun pfSequence?}
\getchunk{defun pfSequenceToList}

```

```

\getchunk{defun pfSequence2Sex}
\getchunk{defun pfSequence2Sex0}
\getchunk{defun pfSexpr}
\getchunk{defun pfSexpr,strip}
\getchunk{defun pfSourcePosition}
\getchunk{defun pfSourceStok}
\getchunk{defun pfSpread}
\getchunk{defun pfSuch}
\getchunk{defun pfSuchthat}
\getchunk{defun pfSuchthat?}
\getchunk{defun pfSuchThat2Sex}
\getchunk{defun pfSymb}
\getchunk{defun pfSymbol}
\getchunk{defun pfSymbol?}
\getchunk{defun pfSymbolSymbol}
\getchunk{defun pfTagged}
\getchunk{defun pfTagged?}
\getchunk{defun pfTaggedToTyped}
\getchunk{defun pfTaggedToTyped1}
\getchunk{defun pfTransformArg}
\getchunk{defun pfTuple}
\getchunk{defun pfTupleListOf}
\getchunk{defun pfTweakIf}
\getchunk{defun pfTyped}
\getchunk{defun pfTyped?}
\getchunk{defun pfTyping}
\getchunk{defun pfTuple?}
\getchunk{defun pfUnSequence}
\getchunk{defun pfWDec}
\getchunk{defun pfWDeclare}
\getchunk{defun pfWhere}
\getchunk{defun pfWhere?}
\getchunk{defun pfWhile}
\getchunk{defun pfWhile?}
\getchunk{defun pfWith}
\getchunk{defun pfWrong}
\getchunk{defun pfWrong?}
\getchunk{defun pf0ApplicationArgs}
\getchunk{defun pf0DefinitionLhsItems}
\getchunk{defun pf0FlattenSyntacticTuple}
\getchunk{defun pf0ForinLhs}
\getchunk{defun pf0FreeItems}
\getchunk{defun pf0LambdaArgs}
\getchunk{defun pf0LocalItems}
\getchunk{defun pf0LoopIterators}
\getchunk{defun pf0MLambdaArgs}
\getchunk{defun pf0SequenceArgs}
\getchunk{defun pf0TupleParts}
\getchunk{defun pf0WhereContext}
\getchunk{defun pf2Sex}

```

```

\getchunk{defun pf2Sex1}
\getchunk{defun phMacro}
\getchunk{defun phParse}
\getchunk{defun phInterpret}
\getchunk{defun phIntReportMsgs}
\getchunk{defun pileCforest}
\getchunk{defun pileColumn}
\getchunk{defun pileCtree}
\getchunk{defun pileForest}
\getchunk{defun pileForest1}
\getchunk{defun pileForests}
\getchunk{defun pilePlusComment}
\getchunk{defun pilePlusComments}
\getchunk{defun pileTree}
\getchunk{defun poFileName}
\getchunk{defun poGlobalLinePosn}
\getchunk{defun poLinePosn}
\getchunk{defun poPosImmediate?}
\getchunk{defun porigin}
\getchunk{defun posend}
\getchunk{defun posPointers}
\getchunk{defun ppos}
\getchunk{defun pquit}
\getchunk{defun pquitSpad2Cmd}
\getchunk{defun previousInterpreterFrame}
\getchunk{defun printLabelledList}
\getchunk{defun printStatisticsSummary}
\getchunk{defun printStorage}
\getchunk{defun printSynonyms}
\getchunk{defun printTypeAndTime}
\getchunk{defun printTypeAndTimeNormal}
\getchunk{defun printTypeAndTimeSaturn}
\getchunk{defun probeName}
\getchunk{defun processChPosesForOneLine}
\getchunk{defun processInteractive}
\getchunk{defun processInteractive1}
\getchunk{defun processKeyedError}
\getchunk{defun processMsgList}
\getchunk{defun protectedSymbolsWarning}
\getchunk{defun protectedEVAL}
\getchunk{defun processSynonymLine}
\getchunk{defun processSynonymLine,removeKeyFromLine}
\getchunk{defun processSynonyms}
\getchunk{defun protectSymbols}
\getchunk{defun prTraceNames}
\getchunk{defun prTraceNames,fn}
\getchunk{defun pspacers}
\getchunk{defun ptimers}
\getchunk{defun put}
\getchunk{defun putFTText}

```

```

\getchunk{defun punctuation?}
\getchunk{defun putDatabaseStuff}
\getchunk{defun putHist}
\getchunk{defun pvarPredTran}

\getchunk{defun queryClients}
\getchunk{defun queueUpErrors}
\getchunk{defun quit}
\getchunk{defun quitSpad2Cmd}

\getchunk{defun rassocSub}
\getchunk{defun rdefinstream}
\getchunk{defun rdefoutstream}
\getchunk{defun read}
\getchunk{defun /read}
\getchunk{defun readHiFi}
\getchunk{defun readSpadProfileIfThere}
\getchunk{defun readSpad2Cmd}
\getchunk{defun recordAndPrint}
\getchunk{defun recordFrame}
\getchunk{defun recordNewValue}
\getchunk{defun recordNewValue0}
\getchunk{defun recordOldValue}
\getchunk{defun recordOldValue0}
\getchunk{defun redundant}
\getchunk{defun remFile}
\getchunk{defun removeOption}
\getchunk{defun removeTracedMapSigs}
\getchunk{defun removeUndoLines}
\getchunk{defun replaceFile}
\getchunk{defun reportOperations}
\getchunk{defun reportOpsFromLisplib}
\getchunk{defun reportOpsFromLisplib0}
\getchunk{defun reportOpsFromLisplib1}
\getchunk{defun reportOpsFromUnitDirectly}
\getchunk{defun reportOpsFromUnitDirectly0}
\getchunk{defun reportOpsFromUnitDirectly1}
\getchunk{defun reportSpadTrace}
\getchunk{defun reportUndo}
\getchunk{defun reportWhatOptions}
\getchunk{defun reroot}
\getchunk{defun resetCounters}
\getchunk{defun resethashtables}
\getchunk{defun resetInCoreHist}
\getchunk{defun resetSpacers}
\getchunk{defun resetTimers}
\getchunk{defun resetWorkspaceVariables}
\getchunk{defun restart}
\getchunk{defun restart0}
\getchunk{defun restoreHistory}

```



```

\getchunk{defun /rf}
\getchunk{defun /rq}
\getchunk{defun rread}
\getchunk{defun ruleLhsTran}
\getchunk{defun rulePredicateTran}
\getchunk{defun runspad}
\getchunk{defun rwrite}

\getchunk{defun safeWritify}
\getchunk{defun sameMsg?}
\getchunk{defun satisfiesRegularExpressions}
\getchunk{defun saveHistory}
\getchunk{defun saveMapSig}
\getchunk{defun savesystem}
\getchunk{defun sayAllCacheCounts}
\getchunk{defun sayBrightly1}
\getchunk{defun sayCacheCount}
\getchunk{defun sayExample}
\getchunk{defun sayKeyedMsg}
\getchunk{defun sayKeyedMsgLocal}
\getchunk{defun sayMSG}
\getchunk{defun sayMSG2File}
\getchunk{defun sayShowWarning}
\getchunk{defun scanCheckRadix}
\getchunk{defun scanComment}
\getchunk{defun scanDictCons}
\getchunk{defun scanError}
\getchunk{defun scanEsc}
\getchunk{defun scanEscape}
\getchunk{defun scanExponent}
\getchunk{defun scanIgnoreLine}
\getchunk{defun scanInsert}
\getchunk{defun scanKeyTr}
\getchunk{defun scanNegComment}
\getchunk{defun scanNumber}
\getchunk{defun ScanOrPairVec}
\getchunk{defun ScanOrPairVec,ScanOrInner}
\getchunk{defun scanPossFloat}
\getchunk{defun scanPunct}
\getchunk{defun scanPunCons}
\getchunk{defun scanS}
\getchunk{defun scanSpace}
\getchunk{defun scanString}
\getchunk{defun scanKeyTableCons}
\getchunk{defun scanToken}
\getchunk{defun scanTransform}
\getchunk{defun scanW}
\getchunk{defun scanWord}
\getchunk{defun search}
\getchunk{defun searchCurrentEnv}

```

```

\getchunk{defun searchTailEnv}
\getchunk{defun segmentKeyedMsg}
\getchunk{defun selectOption}
\getchunk{defun selectOptionLC}
\getchunk{defun separatePiles}
\getchunk{defun serverReadLine}
\getchunk{defun set}
\getchunk{defun set1}
\getchunk{defun setdatabase}
\getchunk{defun setExpose}
\getchunk{defun setExposeAdd}
\getchunk{defun setExposeAddConstr}
\getchunk{defun setExposeAddGroup}
\getchunk{defun setExposeDrop}
\getchunk{defun setExposeDropConstr}
\getchunk{defun setExposeDropGroup}
\getchunk{defun setFortDir}
\getchunk{defun setFortPers}
\getchunk{defun setFortTmpDir}
\getchunk{defun setFunctionsCache}
\getchunk{defun setHistoryCore}
\getchunk{defun setInputLibrary}
\getchunk{defun setIOindex}
\getchunk{defun setLinkerArgs}
\getchunk{defun setMsgCatLessAttr}
\getchunk{defun setMsgForcedAttr}
\getchunk{defun setMsgForcedAttrList}
\getchunk{defun setMsgUnforcedAttr}
\getchunk{defun setMsgUnforcedAttrList}
\getchunk{defun setNagHost}
\getchunk{defun setOutputAlgebra}
\getchunk{defun setOutputCharacters}
\getchunk{defun setOutputFormula}
\getchunk{defun setOutputFortran}
\getchunk{defun setOutputLibrary}
\getchunk{defun setOutputHtml}
\getchunk{defun setOutputMathml}
\getchunk{defun setOutputOpenMath}
\getchunk{defun setOutputTex}
\getchunk{defun setStreamsCalculate}
\getchunk{defun shortenForPrinting}
\getchunk{defun show}
\getchunk{defun showdatabase}
\getchunk{defun showInOut}
\getchunk{defun showInput}
\getchunk{defun showSpad2Cmd}
\getchunk{defun shut}
\getchunk{defun size}
\getchunk{defun SkipEnd?}
\getchunk{defun SkipPart?}

```

```

\getchunk{defun Skipping?}
\getchunk{defun spad}
\getchunk{defun spadClosure?}
\getchunk{defun SpadInterpretStream}
\getchunk{defun spadReply}
\getchunk{defun spadReply,printName}
\getchunk{defun spadrread}
\getchunk{defun spadrwrite}
\getchunk{defun spadrwrite0}
\getchunk{defun spad-save}
\getchunk{defun spadStartUpMsgs}
\getchunk{defun spadTrace}
\getchunk{defun spadTraceAlias}
\getchunk{defun spadTrace,g}
\getchunk{defun spadTrace,isTraceable}
\getchunk{defun spadUntrace}
\getchunk{defun specialChar}
\getchunk{defun spleI}
\getchunk{defun spleI1}
\getchunk{defun splitIntoOptionBlocks}
\getchunk{defun squeeze}
\getchunk{defun stackTraceOptionError}
\getchunk{defun startsComment?}
\getchunk{defun startsNegComment?}
\getchunk{defun statisticsInitialization}
\getchunk{defun streamChop}
\getchunk{defun stringMatches?}
\getchunk{defun StringToDir}
\getchunk{defun strpos}
\getchunk{defun strpos1}
\getchunk{defun stupidIsSpadFunction}
\getchunk{defun subMatch}
\getchunk{defun substringMatch}
\getchunk{defun subTypes}
\getchunk{defun summary}
\getchunk{defun syGeneralErrorHere}
\getchunk{defun syIgnoredFromTo}
\getchunk{defun synonym}
\getchunk{defun synonymsForUserLevel}
\getchunk{defun synonymSpad2Cmd}
\getchunk{defun sySpecificErrorAtToken}
\getchunk{defun sySpecificErrorHere}
\getchunk{defun systemCommand}

\getchunk{defun ?t}
\getchunk{defun tabbing}
\getchunk{defun terminateSystemCommand}
\getchunk{defun tersyscommand}
\getchunk{defun thisPosIsEqual}
\getchunk{defun thisPosIsLess}

```

```

\getchunk{defun toFile?}
\getchunk{defun tokConstruct}
\getchunk{defun tokPosn}
\getchunk{defun tokTran}
\getchunk{defun tokType}
\getchunk{defun toScreen?}
\getchunk{defun trace}
\getchunk{defun trace1}
\getchunk{defun traceDomainConstructor}
\getchunk{defun traceDomainLocalOps}
\getchunk{defun tracelet}
\getchunk{defun traceOptionError}
\getchunk{defun /tracereply}
\getchunk{defun traceReply}
\getchunk{defun traceSpad2Cmd}
\getchunk{defun translateTrueFalse2YesNo}
\getchunk{defun translateYesNo2TrueFalse}
\getchunk{defun transOnlyOption}
\getchunk{defun transTraceItem}

\getchunk{defun unAbbreviateKeyword}
\getchunk{defun undo}
\getchunk{defun undoChanges}
\getchunk{defun undoCount}
\getchunk{defun undoFromFile}
\getchunk{defun undoInCore}
\getchunk{defun undoLocalModemapHack}
\getchunk{defun undoSingleStep}
\getchunk{defun undoSteps}
\getchunk{defun unescapeStringsInForm}
\getchunk{defun unsqueeze}
\getchunk{defun untrace}
\getchunk{defun untraceDomainConstructor}
\getchunk{defun untraceDomainConstructor,keepTraced?}
\getchunk{defun untraceDomainLocalOps}
\getchunk{defun untraceMapSubNames}
\getchunk{defun unwritable?}
\getchunk{defun updateCurrentInterpreterFrame}
\getchunk{defun updateFromCurrentInterpreterFrame}
\getchunk{defun updateHist}
\getchunk{defun updateInCoreHist}
\getchunk{defun updateSourceFiles}
\getchunk{defun userLevelErrorMessage}

\getchunk{defun validateOutputDirectory}

\getchunk{defun what}
\getchunk{defun whatCommands}
\getchunk{defun whatConstructors}
\getchunk{defun whatSpad2Cmd}

```

```

\getchunk{defun whatSpad2Cmd,fixpat}
\getchunk{defun whichCat}
\getchunk{defun with}
\getchunk{defun workfiles}
\getchunk{defun workfilesSpad2Cmd}
\getchunk{defun wrap}
\getchunk{defun write-browsedb}
\getchunk{defun write-categorydb}
\getchunk{defun write-compress}
\getchunk{defun writeHiFi}
\getchunk{defun writeHistModesAndValues}
\getchunk{defun writeInputLines}
\getchunk{defun write-interpdb}
\getchunk{defun write-operationdb}
\getchunk{defun write-warmdata}
\getchunk{defun writify}
\getchunk{defun writifyComplain}
\getchunk{defun writify,writifyInner}

\getchunk{defun xlCannotRead}
\getchunk{defun xlCmdBug}
\getchunk{defun xlConActive}
\getchunk{defun xlConsole}
\getchunk{defun xlConStill}
\getchunk{defun xlFileCycle}
\getchunk{defun xlIfBug}
\getchunk{defun xlIfSyntax}
\getchunk{defun xlMsg}
\getchunk{defun xlNoSuchFile}
\getchunk{defun xlOK}
\getchunk{defun xlOK1}
\getchunk{defun xlPrematureEOF}
\getchunk{defun xlPrematureFin}
\getchunk{defun xlSay}
\getchunk{defun xlSkip}
\getchunk{defun xlSkippingFin}

\getchunk{defun yesanswer}

\getchunk{defun zsystemdevelopment}
\getchunk{defun zsystemdevelopment1}
\getchunk{defun zsystemDevelopmentSpad2Cmd}

\getchunk{postvars}

```

Chapter 70

The Global Variables

70.1 Star Global Variables

NAME	SET	USE
<code>eof*</code>	<code>ncTopLevel</code>	
<code>features*</code>		restart
<code>package*</code>		restart
<code>standard-input*</code>		<code>ncIntLoop</code>
<code>standard-output*</code>		<code>ncIntLoop</code>
<code>top-level-hook*</code>	<code>set-restart-hook</code>	

70.1.1 `*eof*`

The `*eof*` variable is set to NIL in `ncTopLevel`.

70.1.2 `*features*`

The `*features*` variable from common lisp is tested for the presence of the `:unix` keyword. Apparently this controls the use of Saturn, a previous Axiom frontend. The Saturn frontend was never released as open source and so this test and the associated variables are probably not used.

70.1.3 `*package*`

The `*package*` variable, from common lisp, is set in restart to the BOOT package where the interpreter lives.

70.1.4 ***standard-input***

The ***standard-input*** common lisp variable is used to set the `curinstream` variable in `ncIntLoop`.

This variable is an argument to `serverReadLine` in the `intloopReadConsole` function.

70.1.5 ***standard-output***

The ***standard-output*** common lisp variable is used to set the `curoutstream` variable in `ncIntLoop`.

70.1.6 ***top-level-hook***

The ***top-level-hook*** common lisp variable contains the name of a function to invoke when an image is started. In our case it is called `restart`. This is the entry point to the Axiom interpreter.

70.2 Dollar Global Variables

NAME	SET	USE
\$boot	ncTopLevel	
coerceFailure		runspad
curinstream	ncIntLoop	
curoutstream	ncIntLoop	
\$currentLine	restart	removeUndoLines
\$dalymode		intloopReadConsole
\$displayStartMsgs		restart
\$e	ncTopLevel	
\$erMsgToss	SpadInterpretStream	
\$fn	SpadInterpretStream	
\$frameRecord	initvars	
	clearFrame	
	undoSteps	undoSteps
	recordFrame	recordFrame
\$HiFiAccess	initHist	historySpad2Cmd
	historySpad2Cmd	
		setHistoryCore
\$HistList	initHist	
\$HistListAct	initHist	
\$HistListLen	initHistList	
\$HistRecord	initHistList	
\$historyDirectory		makeHistFileName
		makeHistFileName
\$historyFileType	initvars	histInputFileName
\$InteractiveFrame	restart	ncTopLevel
	undo	recordFrame
	undoSteps	undoSteps
		reportUndo
\$internalHistoryTable	initvars	
\$interpreterFrameName	initializeInterpreterFrameRing	
\$interpreterFrameRing	initializeInterpreterFrameRing	
\$intRestart		intloop
\$intTopLevel	intloop	
\$IOindex	restart	historySpad2Cmd
	removeUndoLines	undoCount
\$genValue	bookvol5	i-toplev
		i-analy
		i-syscmd
		i-spec1
		i-spec2
		i-map
\$lastPos	SpadInterpretStream	
\$libQuiet	SpadInterpretStream	
\$msgDatabaseName	reroot *	
\$ncMsgList	SpadInterpretStream	
\$newcompErrorCount	SpadInterpretStream	
\$newspad	ncTopLevel	
\$nopus		SpadInterpretStream
\$okToExecuteMachineCode	SpadInterpretStream	
\$oldHistoryFileName	initvars	oldHistFileName
\$options		history
	historySpad2Cmd	historySpad2Cmd
		undo
\$previousBindings	initvars	
	clearFrame	
	recordFrame	recordFrame
\$PrintCompilerMessageIfTrue	spad	

70.2.1 `$boot`

The `$boot` variable is set to `NIL` in `ncTopLevel`.

70.2.2 `coerceFailure`

The `coerceFailure` symbol is a catch tag used in `runspad` to catch an exit from `ncTopLevel`.

70.2.3 `$currentLine`

The `$currentLine` line is set to `NIL` in `restart`. It is used in `removeUndoLines` in the undo mechanism.

70.2.4 `$displayStartMsgs`

The `$displayStartMsgs` variable is used in `restart` but is not set so this is likely a bug.

70.2.5 `$e`

The `$e` variable is set to the value of `$InteractiveFrame` which is set in `restart` to the value of the call to the `makeInitialModemapFrame` function. This function simply returns a copy of the variable `$InitialModemapFrame`.

Thus `$e` is a copy of the variable `$InitialModemapFrame`.

This variable is used in the undo mechanism.

70.2.6 `$erMsgToss`

The `$erMsgToss` variable is set to `NIL` in `SpadInterpretStream`.

70.2.7 `$fn`

The `$fn` variable is set in `SpadInterpretStream`. It is set to the second argument which is a list. It appears that this list has the same structure as an argument to the `LispVM rdefiostream` function.

70.2.8 `$frameRecord`

`$frameRecord = [delta1, delta2, ...]` where `delta(i)` contains changes in the “backwards” direction. Each `delta(i)` has the form `((var . proplist)...)` where `proplist` denotes an ordinary proplist. For example, an entry of the form `((x (value) (mode (Integer))))` indicates that to undo 1 step, `x`’s value is cleared and its mode should be set to `(Integer)`.

A `delta(i)` of the form `(systemCommand . delta)` is a special delta indicating changes due to system commands executed between the last command and the current command. By recording these deltas separately, it is possible to undo to either BEFORE or AFTER the command. These special `delta(i)`s are given ONLY when a system command is given which alters the environment.

Note: `recordFrame('system)` is called before a command is executed, and `recordFrame('normal)` is called after (see `processInteractive1`). If no changes are found for former, no special entry is given.

This is part of the undo mechanism.

70.2.9 \$HiFiAccess

The `$HiFiAccess` is set by `initHist` to T. It is a flag used by the history mechanism to record whether the history function is currently on. It can be reset by using the axiom command

```
)history off
```

It appears that the name means “History File Access”.

The `$HiFiAccess` variable is used by `historySpad2Cmd` to check whether history is turned on. T means it is, NIL means it is not.

70.2.10 \$HistList

The `$HistList` variable is set by `initHistList` to an initial value of NIL elements. The last element of the list is smashed to point to the first element to make the list circular. This is a circular list of length `$HistListLen`.

70.2.11 \$HistListAct

The `$HistListAct` variable is set by `initHistList` to 0. This variable holds the actual number of elements in the history list. This is the number of “undoable” steps.

70.2.12 \$HistListLen

The `$HistListLen` variable is set by `initHistList` to 20. This is the length of a circular list maintained in the variable `$HistList`.

70.2.13 \$HistRecord

The `$HistRecord` variable is set by `initHistList` to NIL. `$HistRecord` collects the input line, all variable bindings and the output of a step, before it is written to the file named by the function `histFileName`.

70.2.14 `$historyFileType`

The `$historyFileType` is set at load time by a call to `initvars` to a value of “axh”. It appears that this is intended to be used as a filetype extension. It is part of the history mechanism. It is used in `makeHistFileName` as part of the history file name.

70.2.15 `$internalHistoryTable`

The `$internalHistoryTable` variable is set at load time by a call to `initvars` to a value of `NIL`. It is part of the history mechanism.

70.2.16 `$interpreterFrameName`

The `$interpreterFrameName` variable, set in `initializeInterpreterFrameRing` to the constant `initial` to indicate that this is the initial (default) frame.

Frames are structures that capture all of the variables defined in a session. There can be multiple frames and the user can freely switch between them. Frames are kept in a ring data structure so you can move around the ring.

70.2.17 `$interpreterFrameRing`

The `$interpreterFrameRing` is set to a pair whose `car` is set to the result of `emptyInterpreterFrame`

70.2.18 `$InteractiveFrame`

The `$InteractiveFrame` is set in `restart` to the value of the call to the `makeInitialModemapFrame` function. This function simply returns a copy of the variable `$InitialModemapFrame`

70.2.19 `$intRestart`

The `$intRestart` variable is used in `intloop` but has no value. This is probably a bug. While the variable’s value is unchanged the system will continually reenter the `SpadInterpretStream` function.

70.2.20 `$intTopLevel`

The `$intTopLevel` is a catch tag. Throwing to this tags which is caught in the `intloop` will restart the `SpadInterpretStream` function.

70.2.21 `$IOindex`

The `$IOindex` index variable is set to 1 in restart. This variable is used in the `historySpad2Cmd` function in the history mechanism. It is set in the `removeUndoLines` function in the undo mechanism.

This is used in the undo mechanism in function `undoCount` to compute the number of undos. You can't undo more actions than have already happened.

70.2.22 `$lastPos`

The `$lastPos` variable is set in `SpadInterpretStream` to the value of the `$npos` variable. Since `$npos` appears to have no value this is likely a bug.

70.2.23 `$libQuiet`

The `$libQuiet` variable is set to the third argument of the `SpadInterpretStream` function. This is passed from `intloop` with the value of `T`. This variable appears to be intended to control the printing of library loading messages which would need to be suppressed if input was coming from a file.

70.2.24 `$msgDatabaseName`

The `$msgDatabaseName` is set to `NIL` in `reroot`.

70.2.25 `$ncMsgList`

The `$ncMsgList` is set to `NIL` in `SpadInterpretStream`.

70.2.26 `$newcompErrorCount`

The `$newcompErrorCount` is set to 0 in `SpadInterpretStream`.

70.2.27 `$newspad`

The `$newspad` is set to `T` in `ncTopLevel`.

70.2.28 `$npos`

The `$npos` variable is used in `SpadInterpretStream` but does not appear to have a value and is likely a bug.

70.2.29 \$oldHistoryFileName

The `$oldHistoryFileName` is set at load time by a call to `initvars` to a value of “last”. It is part of the history mechanism. It is used in the function `oldHistFileName` and `restoreHistory`.

70.2.30 \$okToExecuteMachineCode

The `$okToExecuteMachineCode` is set to T in `SpadInterpretStream`.

70.2.31 \$options

The `$options` variable is tested by the history function. If it is NIL then output the message

```
You have not used the correct syntax for the history command.  
Issue )help history for more information.
```

The `$options` variable is tested in the `historySpad2Cmd` function. It appears to record the options that were given to a `spad` command on the input line. The function `selectOptionLC` appears to take a list off options to scan.

This variable is not yet set and is probably a bug.

70.2.32 \$previousBindings

The `$previousBindings` is a copy of the CAAR `$InteractiveFrame`. This is used to compute the `delta(i)s` stored in `$frameRecord`. This is part of the undo mechanism.

70.2.33 \$PrintCompilerMessageIfTrue

The `$PrintCompilerMessageIfTrue` variable is set to NIL in `spad`.

70.2.34 \$reportUndo

The `$reportUndo` variable is used in `diffAlist`. It was not normally bound but has been set to T in `initvars`. If the variable is set to T then we call `reportUndo`.

It is part of the undo mechanism.

70.2.35 \$spad

The `$spad` variable is set to T in `ncTopLevel`.

70.2.36 \$SpadServer

If an open server is not requested then this variable to T. It has no value before this time (and is thus a bug).

70.2.37 \$SpadServerName

The `$SpadServerName` is passed to the `openServer` function, if the function exists.

70.2.38 \$systemCommandFunction

The `$systemCommandFunction` is set in `SpadInterpretStream` to point to the function `InterpExecuteSpadSystemCommand`.

70.2.39 top_level

The `top_level` symbol is a catch tag used in `runspad` to catch an exit from `ncTopLevel`.

70.2.40 \$quitTag

The `$quitTag` is used as a variable in a catch block. It appears that it can be thrown somewhere below `ncTopLevel`.

70.2.41 \$useInternalHistoryTable

The `$useInternalHistoryTable` variable is set at load time by a call to `initvars` to a value of `NIL`. It is part of the history mechanism.

70.2.42 \$undoFlag

The `$undoFlag` is used in `recordFrame` to decide whether to do undo recording. It is initially set to T in `initvars`. This is part of the undo mechanism.

Bibliography

- [1] Daly, Timothy, "The Axiom Literate Documentation"
<http://axiom.axiom-developer.org/axiom-website/documentation.html>
- [2] Daly, Timothy, "The Axiom Wiki Website"
<http://axiom.axiom-developer.org>

Chapter 71

Index