

INTERNATIONAL STANDARD

ISO/IEC 6522

Second edition
1992-11-15

Information technology — Programming languages — PL/I general purpose subset

*Technologies de l'information — Langages de programmation —
Sous-ensemble PL/I pour usage général*



Reference number
Documents
AVA Document Sharing Hub

Contents

1. Scope and Overviews 1

1.1 Scope 1

1.2 Referenced Publications 1

1.3 Goals and Rationale 2

1.4 An Informal Guide to the PL/I Definition 3

1.4.1 A Summary of PL/I 3

1.4.2 The Form of the Definition 4

1.4.3 Summary of Chapter Structure 6

1.4.4 Introduction to the Metalanguage 10

1.4.4.1 Tree Concepts 10

1.4.4.2 Syntaxes 11

1.4.4.3 Algorithm Concepts 12

1.5 Relationships between an Implementation and this Definition 14

1.5.1 Flexibilities of Interpretation 15

1.5.1.1 Rejection of Programs 15

1.5.1.2 Quantitative Restrictions 15

1.5.1.3 Addressing Restrictions 15

1.5.1.4 Operating Environment 16

1.5.1.5 Character Sets 16

1.5.1.6 Expression Evaluation 16

1.5.1.7 Interrupts and Assignment 17

1.5.1.8 Input/Output 17

1.5.1.9 On-units 17

1.5.2 Implementation-defined Features 18

1.6 The Metalanguage 20

1.6.1 Trees 21

1.6.1.1 Tree Definitions 21

1.6.1.2 Node Objects 22

1.6.1.2.1 Unique-names 23

1.6.1.2.2 Types 23

1.6.1.3 Node Notation 23

1.6.1.4 Tree Notation 24

1.6.1.4.1 Enumerated-Trees 24

1.6.1.4.2 Forms 25

1.6.1.5 Tree Copies 25

1.6.2 Production Rules 25

1.6.2.1 Production Rules and Syntaxes 25

1.6.2.2 Complete and Partial Trees 26

1.6.2.3 Syntactic-expressions and Syntactic-units 28

1.6.2.4 Use of the Production Rules 28

1.6.2.4.1 Subnode-type Sequence with respect to a Syntax and Category-name 29

1.6.2.4.2 Subnode-type Sequence with respect to a Syntactic-expression 29

1.6.3 Operations 30

1.6.3.1 Nature of an Operation 30

1.6.3.2	Nondeterministic Operations	31
1.6.3.3	Format of an Operation	31
1.6.3.4	Instructions	32
1.6.3.5	Convert	34
1.6.3.6	Additional Notational Conventions	34
1.6.3.7	Arithmetic	34
1.6.3.7.1	Properties of Arithmetic in the Metalanguage	34
1.6.3.7.2	Properties of Arithmetic in PL/I	35
1.6.4	The Processor	36
1.6.5	Mechanization of the Metalanguage	37
1.7	Initialization of the Machine-state	39
1.7.1	The Machine-state	39
1.7.2	Initialization	39
1.7.3	The Top-level Operations	39
1.7.3.1	Define-program	39
1.7.3.2	Translation-phase	40
1.7.3.3	Interpretation-phase	40
2.	Concrete Syntax	41
2.1	Introduction	41
2.2	The Intent of this Definition	41
2.2.1	Concrete and Abstract Syntaxes	41
2.3	Organization of the Concrete Syntax	41
2.4	The High-level Syntax of PL/I	42
2.4.1	Translation Unit	42
2.4.2	Package	42
2.4.3	Procedure	42
2.4.4	Unit	42
2.4.5	Executable Units	42
2.4.6	If Statement	43
2.4.7	Do Group	43
2.4.8	Select Group	43
2.4.9	Begin Block	43
2.4.10	On Unit	44
2.4.10.1	Condition Prefixes	44
2.4.10.2	Statement Name Prefixes	44
2.4.11	Data Declaration	44
2.4.11.1	Dimension Attribute and Dimension Suffix	44
2.4.11.2	Attributes	45
2.4.11.3	Data Attributes	45
2.4.11.4	Environment and Options	46
2.4.11.5	Initial	46
2.4.12	The Package Statement	46
2.4.13	The Procedure Statement	46
2.4.14	The Begin Statement	47
2.4.15	The Do Statement	47
2.4.16	The Select Statement	47
2.4.17	The End Statement	47
2.4.18	Flow of Control Statements	48
2.4.18.1	The Call and Return Statements	48
2.4.18.2	The Go To Statement	48
2.4.18.3	The Leave Statement	48
2.4.18.4	The Null Statement	48
2.4.18.5	The Revert and Signal Statements	48

- 2.4.18.6 The Stop Statement 48
 - 2.4.19 Storage Control Statements 48
 - 2.4.20 Input/Output Statements 49
 - 2.4.20.1 The Open and Close Statements 49
 - 2.4.20.2 Record I/O 49
 - 2.4.20.3 Stream I/O 50
 - 2.4.20.3.1 Stream Input Specification 50
 - 2.4.20.3.2 Stream Output Specification 50
 - 2.4.20.3.3 Format Specification Lists and the Format Statement 50
 - 2.4.21 Expressions 51
 - 2.4.22 Enquiry Functions 52
- 2.5 The Low-level Syntax of PL/I 57
 - 2.5.1 PL/I Text 57
 - 2.5.2 Comment 57
 - 2.5.3 Identifier 57
 - 2.5.4 Arithmetic Constant 57
 - 2.5.5 String Constants and Pictures 58
 - 2.5.6 Include 58
- 2.6 Character Sets 59
 - 2.6.1 Language Character Set 60
 - 2.6.1.1 Letters and Digits 60
 - 2.6.1.2 Special Characters 61
 - 2.6.2 Data Character Set 61
 - 2.6.3 Secondary Case Characters in Numeric and Bit Values 61
- 2.7 Abbreviations 62

3. Abstract Syntax 63

- 3.1 Introduction 63
- 3.2 Abstract Syntax Rules 63
 - 3.2.1 Program 63
 - 3.2.2 Package 63
 - 3.2.3 Procedure 63
 - 3.2.4 Declaration 64
 - 3.2.5 Variable 64
 - 3.2.6 Data-description 64
 - 3.2.7 Data-type 65
 - 3.2.8 Named-constant 66
 - 3.2.9 Entry-point 66
 - 3.2.10 Executable-unit 66
 - 3.2.11 Begin-block 67
 - 3.2.12 Groups 67
 - 3.2.13 Selections 67
 - 3.2.14 On Statement 68
 - 3.2.15 If Statement 68
 - 3.2.16 Flow of Control Statements 68
 - 3.2.17 Storage Statements 69
 - 3.2.18 I/O Statements 69
 - 3.2.19 Record I/O Statements 69
 - 3.2.20 Stream I/O Statements 70
 - 3.2.21 Expression 71
 - 3.2.22 Types of Reference 72
 - 3.2.23 Constant 73
 - 3.2.24 Types of Value 73
 - 3.2.25 Types of Picture 73

4. The Translator	75
4.1 Introduction	75
4.2 Translate	75
4.3 Forming the Translation Unit	76
4.3.1 Low-level-parse	77
4.3.2 High-level-parse	78
4.4 Validation of the Translation Unit	80
4.5 Completion of the External Unit	80
4.5.1 Reorganize	81
4.5.1.1 Complete-options	81
4.5.1.2 Complete-attribute-implications	81
4.5.1.3 Defactor-declarations	82
4.5.2 Construct-explicit-declarations	83
4.5.2.1 Declare-parameters	83
4.5.2.2 Declare-statement-names	84
4.5.2.3 Construct-statement-name-declarations	85
4.5.3 Complete-structure-declarations	87
4.5.3.1 Determine-structure	87
4.5.3.2 Convert-to-logical-levels	88
4.5.3.3 Propagate-alignment	88
4.5.3.4 Find-applicable-declaration	89
4.5.3.5 Find-fully-qualified-name	90
4.5.4 Construct-contextual-declarations	91
4.5.5 Complete-declarations	93
4.5.5.1 Test-attribute-consistency	94
4.5.5.2 Test-invalid-duplicates	96
4.5.5.3 Apply-defaults	96
4.5.5.4 Apply-attribute	98
4.5.5.5 Complete-constructed-entry-declaration	98
4.5.5.6 Test-name-in-description	101
4.5.6 Validate-concrete-declarations	101
4.5.6.1 Check-attribute-completeness-and-delete-attributes	102
4.6 Create-abstract-equivalent-tree	103
4.6.1 Creation of Blocks and Groups	104
4.6.1.1 Create-package	104
4.6.1.2 Create-procedure	105
4.6.1.3 Create-begin-block	106
4.6.1.4 Create-block	106
4.6.1.5 Replace-concrete-designators	107
4.6.1.6 Replace-concrete-precision-designators	107
4.6.1.7 Replace-concrete-named-constant-designators	108
4.6.1.8 Create-group	108
4.6.1.9 Create-selection	109
4.6.1.10 Create-executable-unit-list	110
4.6.1.11 Create-executable-unit	110
4.6.1.12 Create-entry-point	110
4.6.1.13 Create-statement-name	111
4.6.1.14 Create-condition-prefix-list	112
4.6.1.15 Create-condition	112
4.6.1.16 Test-descriptor-extent-expressions	113
4.6.2 Creation of Statements	113
4.6.2.1 Create-assignment-statement	113
4.6.2.2 Data-descriptions Proper for Assignment	114

4.6.2.3	Create-allocation	114
4.6.2.4	Create-format-statement	114
4.6.2.5	Create-remote-format	115
4.6.2.6	Create-format-iteration	115
4.6.2.7	Create-freeing	115
4.6.2.8	Create-if-statement	116
4.6.2.9	Create-balanced-unit	116
4.6.2.10	Create-leave-statement	117
4.6.2.11	Create-on-statement	118
4.6.2.12	Create-open-statement	119
4.6.3	Create-declaration	119
4.6.3.1	Create-named-constant	120
4.6.3.2	Create-variable	120
4.6.3.3	Create-bound-pair-list	121
4.6.3.4	Create-data-description	122
4.6.3.5	Create-data-type	124
4.6.3.6	Create-entry	125
4.6.3.7	Create-refer-option	126
4.6.3.8	Create-identifier	126
4.6.3.9	Create-initial	126
4.6.3.10	Create-array-init	127
4.6.3.11	Create-initial-element	127
4.6.3.12	Create-integer	128
4.6.3.13	Convert-named-literal	128
4.6.3.14	Check-arithmetic-type	130
4.6.3.15	Create-precision	130
4.6.3.16	Create-number-of-digits	130
4.6.3.17	Create-scale-factor	131
4.6.4	Create-expression	131
4.6.5	Create-reference	134
4.6.5.1	Collect-subscripts	137
4.6.5.2	Apply-subscripts	138
4.6.5.3	Create-value-reference	139
4.6.5.4	Trim-dd	139
4.6.5.5	Create-named-constant-reference	139
4.6.5.6	Create-argument-list	140
4.6.5.7	Create-builtin-function-reference	141
4.6.5.8	Create-pseudo-variable-reference	142
4.6.5.9	Create-entry-reference	143
4.6.5.10	Test-matching	144
4.6.6	Create-picture	144
4.6.6.1	Create-numeric-picture	145
4.6.7	Create-constant	147
4.6.8	Create-enquiry-reference	149
4.6.8.1	Create-arithmetic-enquiry-reference	151
4.7	Validation of the External Unit	152
4.7.1	Validate-declaration	152
4.7.2	Validate-automatic-declaration	152
4.7.3	Validate-based-declaration	153
4.7.4	Validate-defined-declaration	154
4.7.5	Validate-parameter-declaration	154
4.7.6	Validate-static-declaration	154
4.7.7	Validate-descriptor	155
4.7.8	Evaluate-restricted-expression	155
4.7.9	Apply-constraints	156

4.7.10 Test-constraints	157
4.7.11 Validate-leave-statements	158
4.8 Validate-program	159
4.8.1 Validate-external-declaration	159
4.8.2 Environment-names	160
5. The PL/I Interpreter	161
5.1 Introduction	161
5.2 The Interpretation-state	161
5.2.1 Directories	161
5.2.2 Block State	161
5.2.3 File Information	163
5.2.4 Storage and Values	164
5.2.5 Generations, Evaluated Data Descriptions, and Evaluated Targets	165
5.2.6 Dataset	165
5.3 Terminology and Definitions	166
5.3.1 Current	166
5.3.2 Block	166
5.4 The Interpret Operation and Interpretation State Initialization	166
5.4.1 Interpret	166
5.4.2 Initialize-interpretation-state	167
5.4.3 Build-file-directory-and-informations	167
5.4.4 Build-fdi	168
5.4.5 Allocate-static-storage-and-build-static-directory	168
5.4.6 Program-epilogue	169
6. Flow Of Control	171
6.1 Introduction	171
6.2 Program Activation and Termination	171
6.2.1 Program Termination	171
6.2.1.1 Execute-stop-statement	172
6.2.1.2 Stop-program	172
6.3 Block Activation and Termination	172
6.3.1 Activate-procedure	172
6.3.1.1 Install-arguments	173
6.3.2 Activate-begin-block	173
6.3.3 Prologue	174
6.3.4 Epilogue	175
6.4 Control within a Block	175
6.4.1 Normal-sequence	176
6.4.1.1 Advance-execution	176
6.4.2 Execute-executable-unit	176
6.4.3 Execute-begin-block	177
6.4.4 Execute-group	177
6.4.4.1 Establish-controlled-group	178
6.4.4.2 Initialize-spec-options	178
6.4.4.3 Test-spec	180
6.4.4.4 Advance-controlled-group	180
6.4.5 Execute-selection	182
6.4.5.1 Test-when-enablement	182
6.4.5.2 Test-when-matching	183
6.4.6 Execute-if-statement	184
6.4.6.1 Establish-truth-value	184
6.4.7 Execute-call-statement	185

6.4.7.1	Entry-references	185
6.4.7.1.1	Evaluate-entry-reference	185
6.4.7.1.2	Establish-argument	186
6.4.8	Execute-goto-statement	187
6.4.8.1	Local-goto	188
6.4.8.2	Trim-group-control	188
6.4.9	Execute-null-statement	189
6.4.10	Execute-return-statement	189
6.4.11	Execute-end-statement	190
6.4.12	Execute-leave-statement	191
6.5	Conditions and Interrupts	191
6.5.1	Conditions	192
6.5.1.1	Raise-condition	192
6.5.1.2	Test-enablement	192
6.5.1.3	Execute-signal-statement	193
6.5.1.4	Evaluate-named-io-condition	194
6.5.2	Interrupts	194
6.5.2.1	Execute-on-statement	194
6.5.2.2	Execute-revert-statement	195
6.5.3	Interrupt	196
6.5.4	System-action	197
6.5.4.1	Comment	197
7.	Storage and Assignment	199
7.1	Introduction	199
7.2	The Generation	200
7.2.1	The Number of Elements in the Storage-index-list of a Generation	200
7.2.2	Correspondence between an Item-data-description and a Basic-value	201
7.2.3	Value of a Generation	202
7.2.4	Value of Storage Index	202
7.3	The Allocation of Storage	203
7.3.1	Execute-allocate-statement	203
7.3.2	Allocate-based-storage	204
7.3.3	Evaluate-in-option	206
7.3.4	Allocate	206
7.3.5	Evaluate-data-description-for-allocation	208
7.3.6	Find-directory-entry	208
7.3.7	Make-allocation-unit	209
7.3.8	Initialize-refer-options	210
7.3.9	Find-block-state-of-declaration	211
7.3.10	Construct-union-list	211
7.4	Initialization	213
7.4.1	Initialize-generation	213
7.4.2	Initialize-scalar-element	214
7.4.3	Initialize-array	214
7.5	The Freeing of Storage	216
7.5.1	Execute-free-statement	216
7.5.2	Free-based-storage	216
7.5.3	Deduce-in-option	218
7.5.4	Free	218
7.6	Assignment	219
7.6.1	The Assignment Statement	219
7.6.2	Target References	220
7.6.2.1	Evaluated Targets	221
7.6.3	The Assignment Operation	222

7.6.3.1 Promote-and-convert	222
7.6.3.2 The Set-storage Operation	223
7.6.3.3 Area-conversion	224
7.6.3.4 Test-union-matching	225
7.6.3.5 Set-union	225
7.6.4 Pseudo-variables	226
7.6.4.1 Exponent-pv	226
7.6.4.2 Onsource-pv	227
7.6.4.3 Pageno-pv	228
7.6.4.4 String-pv	228
7.6.4.5 Substr-pv	229
7.6.4.6 Unspec-pv	230
7.7 Variable-reference	231
7.7.1 Evaluate-variable-reference	231
7.7.1.1 Test Connectedness	232
7.7.1.2 Test String Class	233
7.7.2 Select-based-generation	234
7.7.3 Check-based-reference	234
7.7.4 Overlay-strings	236
7.7.5 Evaluate-data-description-for-reference	237
7.7.6 Select-qualified-reference	238
7.7.7 Select-subscripted-reference	239
7.7.8 Evaluate-defined-reference	241
7.7.9 Evaluate-simply-defined-reference	241
7.7.10 Evaluate-string-overlay-defined-reference	242
7.7.11 Check-simply-defined-reference	242
7.7.12 Extract-slice-of-array	243
7.8 Reference to Named Constant	244
7.8.1 Evaluate-named-constant-reference	244
8. Input/Output	247
8.1 Introduction	247
8.2 Datasets	247
8.2.1 Record Datasets	247
8.2.2 Stream Datasets	247
8.3 Files	248
8.3.1 Record Files	248
8.3.2 Stream Files	248
8.4 I/O Conditions	248
8.4.1 Raise-io-condition	249
8.5 Evaluate-file-option	249
8.6 File Opening and Closing	250
8.6.1 The Open Statement	250
8.6.1.1 Execute-open-statement	250
8.6.1.2 Execute-single-opening	250
8.6.1.3 Open	251
8.6.1.4 Evaluate-title-option	253
8.6.1.5 Evaluate-filename	253
8.6.2 The Close Statement	254
8.6.2.1 Execute-close-statement	254
8.6.2.2 Execute-single-closing	254
8.6.2.3 Close	254
8.7 The Record I/O Statements	255
8.7.1 The Read Statement	255

8.7.1.1	Execute-read-statement	255
8.7.1.2	Read	256
8.7.2	The Write Statement	258
8.7.2.1	Execute-write-statement	258
8.7.2.2	Write	259
8.7.3	The Rewrite Statement	259
8.7.3.1	Execute-rewrite-statement	259
8.7.3.2	Rewrite	260
8.7.4	The Delete Statement	261
8.7.4.1	Execute-delete-statement	261
8.7.4.2	Delete	262
8.7.5	Operations Applicable to Record I/O	263
8.7.5.1	Evaluate-from-option	263
8.7.5.2	Evaluate-into-option	263
8.7.5.3	Evaluate-pointer-set-option	263
8.7.5.4	Evaluate-key-option	264
8.7.5.5	Evaluate-keyfrom-option	264
8.7.5.6	Evaluate-keyto-option	264
8.7.5.7	Evaluate-sizeto-option	265
8.7.5.8	Construct-record	265
8.7.5.9	Insert-record	265
8.7.5.10	Position-file	266
8.7.5.11	Evaluate-size	267
8.7.5.12	Exit-from-io	268
8.7.5.13	Trim-io-control	268
8.8	The Stream I/O Statements	269
8.8.1	The Get Statement	269
8.8.1.1	Execute-get-statement	269
8.8.1.2	Execute-get-file	269
8.8.1.3	Execute-get-string	270
8.8.1.4	Get-list	270
8.8.1.4.1	Parse-list-input	272
8.8.1.4.2	Parsing Categories for List Directed Input	273
8.8.1.5	Get-edit	273
8.8.1.5.1	Execute-input-control-format	274
8.8.1.5.2	Execute-input-data-format	275
8.8.1.5.3	Validate-input-format	277
8.8.1.6	Input-stream-item	278
8.8.1.7	Basic-character-value	279
8.8.1.8	Basic-bit-value	279
8.8.1.9	Input-stream-item-for-edit	280
8.8.2	The Put Statement	280
8.8.2.1	Execute-put-statement	280
8.8.2.2	Execute-put-file	281
8.8.2.3	Execute-put-string	281
8.8.2.4	Put-list	282
8.8.2.5	Put-edit	284
8.8.2.5.1	Execute-output-control-format	284
8.8.2.5.2	Execute-output-data-format	285
8.8.2.5.3	Edit-numeric-output	286
8.8.2.6	Output-string	288
8.8.2.7	Output-string-item	288
8.8.2.8	Output-stream-item	289
8.8.2.9	Tab	289
8.8.2.9.1	Output-tab	290

8.8.2.10 Put-line	290
8.8.2.11 Put-page	291
8.8.3 Operations Applicable to Stream I/O	291
8.8.3.1 Skip	291
8.8.3.2 Evaluate-current-column	292
8.8.3.3 Evaluate-current-line	292
8.8.3.4 Establish-next-data-item	293
8.8.3.4.1 Expand-edd	295
8.8.3.4.2 Expand-generation	295
8.8.3.5 Establish-next-format-item	296
8.8.3.6 Evaluate-format-item	297
8.8.3.6.1 Evaluate-format-expression	297
9. Expressions and Conversion	299
9.1 Introduction	299
9.2 Aggregate Expressions	299
9.2.1 Scalar and Aggregate Types	299
9.2.1.1 Aggregate Type of a Data Description	299
9.2.1.2 Scalar Elements	300
9.2.1.3 Treatment of Scalars	300
9.2.1.4 Compatibility	300
9.2.1.5 Correspondence	300
9.2.1.5.1 Correspondence of Scalar Elements	300
9.2.1.5.2 Correspondence of Data Types	300
9.2.1.5.3 Correspondence of Union Data Descriptions	301
9.2.1.6 Generate-scalar-result	301
9.2.2 Integer Type	302
9.2.2.1 Evaluate-expression-to-integer	302
9.2.3 Derived Data Types	302
9.2.3.1 Derived Base and Scale	303
9.2.3.2 Converted Precision	303
9.2.3.3 Derived String Type	304
9.2.3.4 Further Definitions for Character and Bit Strings	304
9.2.4 Arithmetic Results	305
9.2.4.1 Conditions in Expressions	306
9.2.4.2 Operations Used in the Model of Floating-Point Arithmetic	306
9.2.4.2.1 Chopped-result	306
9.2.4.2.2 Composite-result	307
9.2.4.2.3 Rounded-result	308
9.2.4.2.4 Unconstrained-result	310
9.2.4.2.5 Test-model-number	310
9.2.4.2.6 Highest-positive-model-number	311
9.2.4.2.7 Lowest-positive-model-number	311
9.2.4.2.8 Lower-model-number	311
9.2.4.2.9 Higher-model-number	311
9.2.4.2.10 Model-base	312
9.2.4.2.11 Model-precision	312
9.2.4.2.12 Model-minimum-exponent	312
9.2.4.2.13 Model-maximum-exponent	312
9.2.5 Expressions	313
9.2.6 Value References	313
9.2.7 Constants	314
9.2.8 Parenthesized Expressions	314
9.2.9 Arguments	314

9.2.10 Enquiries	314
9.3 Prefix Operators	317
9.3.1 Prefix Expressions	317
9.3.2 Definition of the Prefix Operators	317
9.3.2.1 Prefix-minus	317
9.3.2.2 Prefix-not	318
9.3.2.3 Prefix-plus	318
9.4 Infix Operators	318
9.4.1 Infix Expressions	319
9.4.2 Definition of the Infix Operators	319
9.4.2.1 Infix-add	319
9.4.2.2 Infix-and	320
9.4.2.3 Infix-and-logical	320
9.4.2.4 Infix-cat	321
9.4.2.4.1 Concatenation of String Values	321
9.4.2.5 Infix-divide	321
9.4.2.6 Infix-eq	322
9.4.2.6.1 Compare	322
9.4.2.6.2 Non-trivial-basic-value-index	326
9.4.2.7 Infix-ge	326
9.4.2.8 Infix-gt	327
9.4.2.9 Infix-le	327
9.4.2.10 Infix-lt	328
9.4.2.11 Infix-multiply	328
9.4.2.12 Infix-ne	329
9.4.2.13 Infix-or	329
9.4.2.14 Infix-or-logical	330
9.4.2.15 Infix-power	330
9.4.2.16 Infix-subtract	331
9.4.2.17 Infix-xor	332
9.5 Built-in Functions	332
9.5.1 Built-in Function Reference	333
9.5.2 Special Terms Defined for Built-in Functions	333
9.5.2.1 Definition of N	333
9.5.2.2 The Arguments p and q	334
9.5.2.2.1 Constraints on p and q	334
9.5.2.2.2 Attributes of Result Determined by p and q	334
9.5.3 Operations Used in Built-in Function Definitions	334
9.5.3.1 Get-established-onvalue	334
9.5.4 Definition of the Built-in Functions	335
9.5.4.1 Abs-bif	335
9.5.4.2 Acos-bif	335
9.5.4.3 Add-bif	336
9.5.4.4 Addr-bif	336
9.5.4.5 Asin-bif	336
9.5.4.6 Atan-bif	337
9.5.4.7 Atand-bif	338
9.5.4.8 Atanh-bif	338
9.5.4.9 Binary-bif	339
9.5.4.10 Bit-bif	339
9.5.4.11 Bool-bif	339
9.5.4.12 Ceil-bif	340
9.5.4.13 Character-bif	341
9.5.4.14 Collate-bif	341
9.5.4.15 Copy-bif	342

9.5.4.16	Cos-bif	342
9.5.4.17	Cosd-bif	343
9.5.4.18	Cosh-bif	343
9.5.4.19	Currentsize-bif	344
9.5.4.20	Date-bif	344
9.5.4.21	Datetime-bif	345
9.5.4.22	Decimal-bif	345
9.5.4.23	Dimension-bif	345
9.5.4.24	Divide-bif	346
9.5.4.25	Edit-bif	347
9.5.4.26	Empty-bif	347
9.5.4.27	Every-bif	348
9.5.4.28	Exp-bif	348
9.5.4.29	Exponent-bif	349
9.5.4.30	Fileopen-bif	349
9.5.4.31	Fixed-bif	350
9.5.4.32	Float-bif	350
9.5.4.33	Floor-bif	350
9.5.4.34	Hbound-bif	351
9.5.4.35	High-bif	352
9.5.4.36	Identical-bif	352
9.5.4.37	Index-bif	353
9.5.4.38	Isochar-bif	354
9.5.4.39	Lbound-bif	355
9.5.4.40	Length-bif	356
9.5.4.41	Lineno-bif	356
9.5.4.42	Log-bif	357
9.5.4.43	Log10-bif	357
9.5.4.44	Log2-bif	357
9.5.4.45	Low-bif	358
9.5.4.46	Max-bif	358
9.5.4.47	Maxlength-bif	359
9.5.4.48	Min-bif	360
9.5.4.49	Mod-bif	361
9.5.4.50	Multiply-bif	362
9.5.4.51	Null-bif	362
9.5.4.52	Offset-bif	362
9.5.4.53	Oncode-bif	363
9.5.4.54	Onfile-bif	363
9.5.4.55	Onkey-bif	363
9.5.4.56	Onsource-bif	364
9.5.4.57	Pageno-bif	364
9.5.4.58	Pointer-bif	365
9.5.4.59	Prod-bif	365
9.5.4.60	Reverse-bif	366
9.5.4.61	Round-bif	366
9.5.4.62	Search-bif	367
9.5.4.63	Sign-bif	368
9.5.4.64	Sin-bif	368
9.5.4.65	Sind-bif	369
9.5.4.66	Sinh-bif	369
9.5.4.67	Some-bif	370
9.5.4.68	Sqrt-bif	370
9.5.4.69	String-bif	371

9.5.4.70	Substr-bif	371
9.5.4.71	Subtract-bif	372
9.5.4.72	Sum-bif	372
9.5.4.73	Tally-bif	373
9.5.4.74	Tan-bif	374
9.5.4.75	Tand-bif	375
9.5.4.76	Tanh-bif	375
9.5.4.77	Time-bif	375
9.5.4.78	Translate-bif	376
9.5.4.79	Trim-bif	377
9.5.4.80	Trunc-bif	378
9.5.4.81	Unspec-bif	379
9.5.4.82	Valid-bif	380
9.5.4.83	Verify-bif	380
9.6	Conversion	381
9.6.1	Conversion of Scalar Values	381
9.6.1.1	Informal Invocation of Convert	384
9.6.1.2	Convert-to-fixed	384
9.6.1.3	Convert-to-float	385
9.6.1.4	Convert-to-bit	386
9.6.1.5	Convert-to-character	387
9.6.1.6	Conversion to Float Decimal	388
9.6.1.7	Conversion from String or Picture to Arithmetic	389
9.6.1.8	Basic Numeric Value of a String	389
9.6.1.9	Evaluate-real-constant	390
9.6.1.10	Validate-offset	391
9.6.2	Numeric Pictures	391
9.6.2.1	Editing Numeric Pictures	392
9.6.2.2	Editing a Numeric Picture Field	393
9.6.2.3	Validity of a Numeric Pictured Value	397
9.6.2.4	Validity of a Field of a Numeric Pictured Value	398
Appendix A.	Features New to This Revision Adopted from X3.53-1976	399
A.1	Lexical Constructs	399
A.2	Attributes and Declarations	399
A.3	Program Structure and Control	400
A.4	Storage Control	400
A.5	Input/Output	401
A.6	Expressions and Evaluation	401
A.7	Pictures	401
A.8	Built-in Functions and Pseudo-variables	401
A.9	Conditions and Exception Handling	403
Appendix B.	Features New to This Revision	405
B.1	Lexical Constructs	405
B.2	Attributes and Declarations	405
B.3	Program Structure and Control	406
B.4	Storage Control	407
B.5	Input/Output	407
B.6	Expressions and Evaluation	408
B.7	Pictures	408
B.8	Built-in Functions and Pseudo-variables	409
B.9	Enquiries	410
B.10	Conditions and Exception Handling	411

Appendix C. Clarifications and Restrictions from X3.74–1981	413
Appendix D. Features of X3.53–1976 not Adopted in This Standard	415
D.1 Program Structure	415
D.1.1 Syntactic Restrictions	415
D.1.2 Semantic Restrictions	415
D.1.3 Features Excluded	415
D.2 Program Control	416
D.2.1 Syntactic Restrictions	416
D.2.2 Semantic Restrictions	416
D.2.3 Features Excluded	416
D.3 Storage Control Statements	416
D.3.1 Syntactic Restrictions	416
D.3.2 Semantic Restrictions	416
D.3.3 Features Excluded	416
D.4 Input/Output	417
D.4.1 Syntactic Restrictions	417
D.4.2 Semantic Restrictions	417
D.4.3 Features Excluded	417
D.5 Attributes and Pictures	417
D.5.1 Syntactic Restrictions	417
D.5.2 Semantic Restrictions	418
D.5.3 Features Excluded	418
D.6 Built-in Functions and Pseudo-variables	418
D.6.1 Syntactic Restrictions	418
D.6.2 Semantic Restrictions	418
D.6.3 Features Excluded	419
D.7 Expressions	419
D.7.1 Syntactic Restrictions	419
D.7.2 Semantic Restrictions	419
D.7.3 Features Excluded	420
D.8 Change in Intermediate Results	420
Appendix E. Incompatible Changes from X3.74–1981	421
Appendix F. Rationale for Major Design Decisions	423
Index	427

Figures

1.	Example of Syntax Brackets.	4
2.	The Prologue Operation.	13
3.	Examples of Syntax.	27
4.	An Example of an Operation Returning a Result.	31
5.	An Example of an Operation Containing a Step-list and a Case-set.	32
6.	Example Showing <operation-list>s of a <machine-state>.	38
7.	The Structure of a Simple <translation-unit>..	53
8.	The Structure of an <if-statement>..	54
9.	The Structure of an <expression>..	55
10.	The Structure of a <do-statement>..	56
11.	Examples of Abstract Text.	74
12.	Table of <bit-values> as a Function of <symbols> and <radix-factors> for CREATE-CONSTANT. .	149
13.	An Example of <generation>s and <basic-value>s of Defined Variables.	204
14.	Table of Converted Precisions as a Function of Target and Source Attributes.	304
15.	An Example of Optional Arguments.	333
16.	Table of Scalar-results as a Function of <bit-values> for BOOL-BIF.	340
17.	An Example of the BOOL Built-in Function.	340
18.	An Example of the COPY Built-in Function.	342
19.	An Example of the DIMENSION Built-in Function.	346
20.	Examples of the INDEX Built-in Function.	354
21.	Some Examples of the MAXLENGTH Built-in Function.	360
22.	Some Examples of the TALLY Built-in Function.	374
23.	An Example of the TRANSLATE Built-in Function.	377
24.	An Example of the TRIM Built-in Function.	378
25.	Examples of the VERIFY Built-in Function.	381
26.	Table of Terminal Nodes of <symbols> as a Function of <suppression-type> for EDIT-NUMERIC-PICTURE-FIELD	397

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 6522 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

This second edition cancels and replaces the first edition (ISO 6522:1985), which has been technically revised.

Annexes A to F of this International Standard are for information only.

Terminology and conventions

The text of American National Standard ANSI X3.74-1987 has been approved for publication, without deviation, as an International Standard. Some terminology and certain conventions are not in accordance with the ISO/IEC Directives Part 3: “Drafting and presentation of International Standards”; attention is especially drawn to the following:

Wherever the word “standard” appears, referring to this International Standard, it should be read as “International Standard”.

Cross-reference

American National Standard	Corresponding International Standard
ANSI X3.134.1	ISO/IEC 4873:1991, <i>Information technology — ISO 8-bit code for information interchange — Structure and rules for implementation</i> .
ANSI X3.4-1986	ISO/IEC 646:1991, <i>Information technology — ISO 7-bit coded character set for information interchange</i> .
ANSI X3.41-1974	ISO 2022:1986, <i>Information processing — ISO 7-bit and 8-bit coded character sets — Coded extension techniques</i> .
ANSI X3.53-1976	ISO 6160:1979, <i>Programming languages — PL/I</i> .

This page intentionally left blank

Information technology — Programming languages — PL/I general purpose subset

1. Scope and Overviews

1.1 Scope

This document defines the computer programming language PL/I General Purpose Subset, referred to in most of this definition simply as “PL/I”. It is intended to serve as an authoritative reference rather than as a tutorial introduction.

The language defined by this document is different from those of previous PL/I standards, although substantially upward compatible, at the source program and semantic level. The differences are summarized in Appendix A. The text that follows defines standard PL/I in terms of an extended reference PL/I language. The extended reference PL/I language contains operations and portions of operations that are not accessed by PL/I programs that conform to this standard. Those features that are implied by the reference language but for which support is not required of conforming processors are retained for convenience of PL/I implementations containing extensions beyond this standard and for definitional compatibility with earlier PL/I standards (see Appendix A).

As explained in more detail in Section 1.5, this standard specifies the syntax and semantics of conforming PL/I programs. It makes no proscriptive statements about the syntax or semantics of non-conforming programs. It defines a conforming processor (or conforming implementation) only in terms of those conforming programs: a conforming processor is one that, subject to the flexibilities and requirements of Section 1.5, gives all conforming programs the interpretations specified by this definition.

The definition is accomplished by specifying a conceptual PL/I machine which translates and interprets intended PL/I programs. Section 1.4 provides a brief introduction to the statements and data types included in the language, to the structure and use of the document, and to the method of definition. The relationship between an actual implementation and the conceptual machine of this document is described in Section 1.5, and the detailed specification of the notation to be used follows in Section 1.6. The main body of the definition is then begun at Section 1.7, and is completed by Chapters 2 through 9.

1.2 Referenced Publications

This standard shall be used in conjunction with the following publications. When the following American National Standards are superseded by a revision approved by the American National Standards Institute, Inc., the revision shall apply.

- ANSI X3.134.1, Information Systems - 8-bit ASCII - Structure and Rules (under development)

Note: For international use, ISO 4873-1984 or its successors should be substituted.

- ANSI X3.4-1986, Information Systems - Coded Character Set - 7-Bit American National Standard Code for Information Interchange

Note: For international use, ISO 646-1983 or its successors should be substituted.

- ANSI X3.41-1974, Information Systems - Code Extension Techniques for Use with the 7-bit Coded Character Set of American National Standard Code for Information Interchange (ASCII)

Note: For international use, ISO 2022-1986 or its successors should be substituted.

- ANSI X3.53-1976 (R 1987), Programming Language PL/I

Note: Identical to ISO 6160-1979.

1.3 Goals and Rationale

The language defined by this document differs from the PL/I language of ANSI X3.53-1976 and ANSI X3.74-1981 by having a few additional features that have been determined to be necessary in the years since the publication of those documents. With the exception of those new features, it remains, like its predecessor, a proper subset of the full PL/I language specified in American National Standard Programming Language PL/I, ANSI X3.53-1976. The differences from those earlier standards are summarized in Appendix A and Appendix B. The language defined here is designed to have the following properties:

1. The subset is a general-purpose programming language that is well suited for use in commercial, scientific, and systems programming application areas.
2. The subset is small enough to achieve widespread implementation yet large enough to achieve widespread usage.
3. The subset is easier to learn and understand, substantially less expensive to implement, and utilizes computer resources more economically than full PL/I (ANSI X3.53-1976).
4. Programs written in the subset are more portable than programs written in the full language.
5. Programs written in the subset are less likely to contain programming errors than programs written in the full language because the subset is both simpler and less permissive than the full language.

Technology now permits the implementation of subsets of PL/I on a variety of medium and small hardware configurations with limited control program software, including most minicomputer and some microprocessor-based systems.

A major benefit of the language defined here is the reduction or elimination of incompatibilities among the various PL/I implementations on such machines.

The language defined here provides a high degree of portability of both programs and programmers among different computers and operating systems. It includes special features for making programs adaptable to differences in environment that go beyond the language itself and has been designed so that programs can be easily converted to and from non-standard PL/I dialects.

1.4 An Informal Guide to the PL/I Definition

1.4.1 A Summary of PL/I

A PL/I program consists of a nonempty set of procedures and a possibly empty set of packages, each of which is written as a sequence of statements. The %INCLUDE construct may be used to include text from other sources during program translation.

All of the statement types are summarized here in groupings which are presented as a means of obtaining an overview of the language. The structure of the document reflects, to some extent, the organization of this list.

Structural:	PACKAGE PROCEDURE BEGIN DO END	
Declarative:	DECLARE DEFINE CONSTANT FORMAT	
Flow of Control:	CALL RETURN IF SELECT LEAVE GO TO Null Statement STOP	
	ON REVERT SIGNAL	 Interrupt Handling
Storage:	ALLOCATE FREE	
	Assignment Statement	
Input/Output:	OPEN CLOSE	
	GET PUT	 Stream I/O
	READ WRITE REWRITE DELETE	 Record I/O

Names may be declared to represent data of the following types, either as single values, or as aggregates in the form of arrays or structures:

Arithmetic
Arithmetic PICTURE
CHARACTER
BIT
AREA
ENTRY
FILE
LABEL
OFFSET
POINTER

Values may be computed by expressions written using a specific set of operators, built-in functions and enquiries, some of which may be applied to aggregates as well as to single values, together with user-defined procedures which, likewise, may operate on aggregate as well as single values, but may only return single values.

1.4.2 The Form of the Definition

The conceptual PL/I machine is a processor which has a set of operations acting on information stored in its memory. The operations are specified as algorithms, and may be viewed as the component parts of one single algorithm, “DEFINE-PROGRAM”, which carries out the entire translation and interpretation process. The information in memory is all held in the form of tree structures.

The definition algorithm operates as follows.

Sequences of symbols which are intended to represent PL/I *packages* and *external procedures* (i.e. procedures not contained in any other procedure or package) are processed by a Translator. Packages and external procedures are collectively called *translation-units*. This processing involves systematically analyzing, transforming, and validating each translation unit. The analysis uses a grammar known as the *Concrete Syntax* to produce the *concrete form* of a translation unit as a tree structure. This is transformed to the *abstract form*, which is a tree satisfying the *Abstract Syntax*, designed to be more convenient for interpretation. Further validation is then carried out on the abstract form. The Translator finally performs some validation of the mutual consistency of the set of translation units which comprise the complete program.

The semantics of the program when applied to given initial *datasets* (i.e. collections of data) are then provided by an Interpreter. The datasets are part of the PL/I machine’s memory, which is a tree satisfying the *Machine-state Syntax*, and it is the sequence of changes in the datasets which constitutes the defined semantics.

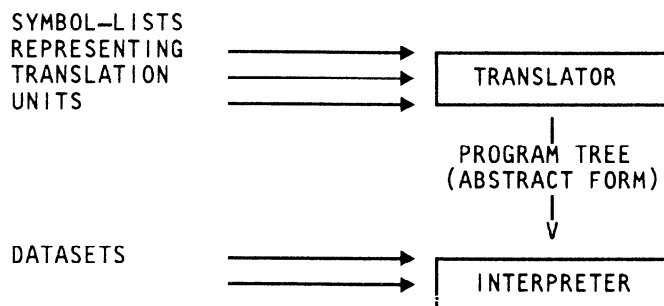
The denotation of the category names of each of the syntaxes includes the following brackets:

⟨...⟩	Concrete syntax
<...>	Abstract syntax
◁...▷	Machine state syntax

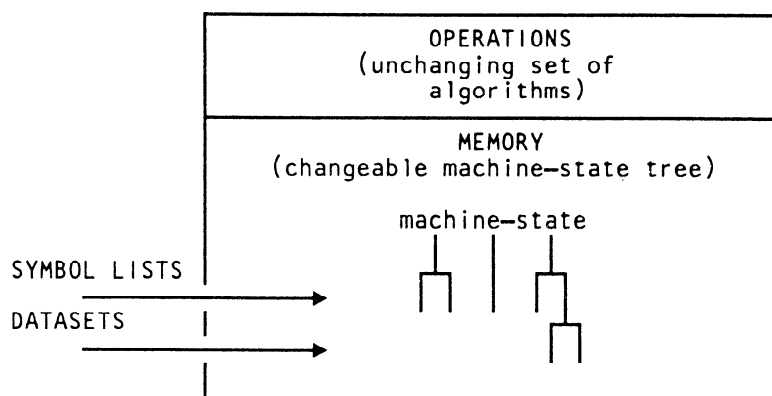
Figure 1. Example of Syntax Brackets.

In addition to translating and interpreting all programs which are valid according to this definition, the PL/I machine detects as non-standard all those which violate a requirement involving the words “MUST” or “MUST NOT” in the algorithms performed during translation or interpretation. The implications for an actual implementation are described in Section 1.5.

The method of definition may be seen in outline as follows:



It may also be helpful to visualize the abstract machine on which both the Translator and the Interpreter are “executed”:



The inputs from outside the machine occur at initialization of the Translator and Interpreter; the datasets may change during interpretation. However, there are no outputs defined since the datasets are treated for the purposes of this definition as being a part of the storage of the machine, i.e. as being “on-line” when needed.

1.4.3 Summary of Chapter Structure

1.TOP-LEVEL OF MACHINE-STATE AND OPERATIONS	
2.CONCRETE SYNTAX 3.ABSTRACT SYNTAX 4.TRANSLATOR	
5.INTERPRETATION-STATE + TOP-LEVEL OF INTERPRETER 6.FLOW OF CONTROL 7.STORAGE AND ASSIGNMENT 8.INPUT/OUTPUT 9.EXPRESSIONS	Concerned with the three parts of the interpretation-state Common Subroutines for Chapters 5-8
Appendix A: FEATURES NEW TO THIS REVISION ADOPTED FROM X3.53-1976 Appendix B: FEATURES NEW TO THIS REVISION Appendix C: CLARIFICATIONS AND RESTRICTIONS FROM X3.74-1981 Appendix D: FEATURES OF X3.53-1976 NOT ADOPTED IN THIS STANDARD Appendix E: INCOMPATIBLE CHANGES FROM X3.74-1981 Appendix F: RATIONALE FOR MAJOR DESIGN DECISIONS	

The operations of Chapter 1 serve to drive the Translator and Interpreter.

The operations of the Translator are all contained in Chapter 4, and use the syntaxes of Chapters 2 and 3.

The operations of the Interpreter comprise all the operations in Chapters 5-9. After the initialization in Chapter 5, the relevant operations will be in Chapters 6, 7, or 8 depending on the type of statement being interpreted. All of these chapters invoke operations in Chapter 9 where necessary.

All readers are recommended to acquire a good understanding of Chapter 1 in its entirety. Thereafter, it is possible to read the definition as a systematic whole, or to use the document to locate answers to specific questions by combining an appreciation of the overall structure of the definition with judicious use of the index. To illustrate this latter usage, we consider each chapter in turn together with a sample question answerable from it.

Chapter 2 contains the definition of the Concrete Syntax. The Concrete Syntax consists of rules describing valid forms of PL/I constructs in concrete tree form. The syntax is permissive in the sense that some of the constructs permitted as being syntactically correct may later be found to be meaningless.

QUESTION

Is the following statement correct?

GET LIST (A(I,J) DO I = 1 TO M,N);

ANSWER

The first possibility is that there may be an error according to the Concrete Syntax. The index entries for “get-statement” lead to the High-level Syntax, and study of rules CH115, CH116, CH123, CH124, and CH125 reveals that an extra pair of parentheses is required around the form $\langle \text{input-target-commalist} \rangle$ DO $\langle \text{do-spec} \rangle$. The correct form is:

GET LIST ((A(I,J) DO I = 1 TO M),N);

Chapter 3 contains the definition of the Abstract Syntax. Many parts of the Abstract Syntax description intentionally bear a strong resemblance to the corresponding parts of the Concrete Syntax. Names in the Abstract Syntax have been chosen to resemble those of corresponding parts of the Concrete Syntax in order to make obvious as far as possible the relationship between the syntaxes.

QUESTION

May the KEYTO option on a READ statement specify that the key be assigned to a substring of a variable?

ANSWER

The Concrete Syntax for a $\langle \text{keyto-option} \rangle$ shows merely that a $\langle \text{reference} \rangle$ MUST be specified. However, the Abstract Syntax shows the form of a program after the Translator has completed all declarations, and has thus been able to associate each reference with the appropriate declaration and make more subtle distinctions. The rule A123 for $\langle \text{keyto-option} \rangle$ shows “ $\langle \text{variable-reference} \rangle$ (scalar AND character)”. The parenthesized constraint “(scalar AND character)” shows that it MUST be a single target (not an array or structure) which is character-valued.

$\langle \text{pseudo-variable-reference} \rangle$ s are only allowed in a $\langle \text{target-reference} \rangle$. Since the $\langle \text{keyto-option} \rangle$ does not accept a $\langle \text{target-reference} \rangle$, a substring may not be used. Pseudo-variables are only allowed on the left hand side of an assignment statement.

Chapter 4 defines the Translator whereby each of the individual PL/I program portions (translation units) is translated from the submitted character string form to tree form and appended to the program tree. This process involves the parsing of each translation-unit using the Concrete Syntax to obtain a concrete tree, insertion of missing options and completion of attribute sets in that concrete tree, conversion from that concrete tree to an abstract tree, and, finally, validation of the whole program. Once formed, the abstract tree is not modified.

QUESTION

What file is implied in PUT LIST(X); ?

ANSWER

This seems at first sight as though it might be a semantic question about the $\langle \text{put-statement} \rangle$. However, the Abstract Syntax shows that a $\langle \text{file-option} \rangle$ is required to be present in $\langle \text{put-file} \rangle$ (A130), and this means that if it was absent in the concrete form, it would have been supplied by the Translator if the statement was valid.

In fact, immediately after parsing the input, the Translator completes the concrete procedure in various ways, one of which is to insert the equivalent of FILE (SYSPRINT) into our $\langle$ put-statement $\rangle$ (Step 2 of the operation COMPLETE-OPTIONS, 4.5.1.1).

The reason that it has to be handled early in the Translator is that it will lead to a contextual declaration of the name SYSPRINT if our statement is not within the scope of an existing declaration for SYSPRINT. It is necessary to complete all declarations prior to execution in order to resolve references correctly.

Chapter 5 contains the definitions of the Machine-state Syntax, the initialization of the machine-state tree, and the starting of the interpretation process.

QUESTION

May the first procedure executed in a program have arguments passed to it?

ANSWER

For the initialization of execution, we consult Sections 1.7.3.3 and 5.4.1.

In Step 1 of operation INTERPRETATION-PHASE (Section 1.7.3.3) an $\langle$ entry-value $\rangle$ is obtained from outside the definition. It designates the first procedure in the $\langle$ program $\rangle$ to be activated. The “MUST NOT” condition specifies that the document gives no meaning to a program whose $\langle$ entry-point $\rangle$ designated by the $\langle$ entry-value $\rangle$ contains a $\langle$ parameter-name-list $\rangle$. (See Section 1.6.3.4 for the definition of MUST.)

Additionally, in Step 2 of operation INTERPRET (5.4.1), an $\langle$ evaluated-entry-reference $\rangle$ containing only an $\langle$ entry-value $\rangle$ and not an $\langle$ established-argument-list $\rangle$ (see production rule M22 in 5.2.2) is used to activate the first procedure.

Thus the passing of an argument-list to the first procedure would be an extension beyond the language defined in this document.

Chapter 6 describes the operations of the Interpreter affecting the flow of control through blocks, groups, and statements of the program. Normal flow of control consists of the execution of a list of executable units within a block. The definition also defines calling, parameter/argument matching and result returning, and abnormal flow of control caused by interrupts.

QUESTION

Is it permissible to REVERT a condition for which no ON statement has been executed?

ANSWER

This is a semantic question about interrupt handling. The operation EXECUTE-REVERT-STATEMENT (6.5.2.2) deletes a member from the current $\langle$ established-on-unit-list $\rangle$ if an appropriate one exists, otherwise it performs NORMAL-SEQUENCE to pass on to the next statement. Since the action of EXECUTE-ON-STATEMENT is merely to append $\langle$ established-on-unit $\rangle$ s to such lists, it is clear that EXECUTE-REVERT-STATEMENT is indifferent to the absence of such actions. The answer to the question is that it is permissible and has no effect on the interrupt handling mechanism.

Chapter 7 defines the use of storage including the allocation, freeing and initialization of storage, and the referencing of variables and named constants. The assignment statement, aggregate assignment, and pseudo-variables, are also defined.

QUESTION

Does DECLARE A(5) INITIAL (0); lead to all 5 elements of A being set to 0 when A is allocated?

ANSWER

The allocation and initialization of storage are treated in Chapter 7.

The operation INITIALIZE-ARRAY (7.4.3) iterates over Step 4 while $n \leq nt$. Since nt in this case is 1 (see Step 2) and n is initially 1 and is incremented in Step 4.7, the iteration is performed once only, thus initializing the first element A(1) only.

If it is desired to initialize the entire array with a scalar value, there is a mechanism to do this. The syntax is of the form DCL A(5) INITIAL((*)0); and the operation is carried out in Step 1 of INITIALIZE-ARRAY.

Chapter 8 deals with the transmission of data between external media and internal storage, including the opening and closing of files, stream and record transmission, and interrupts applicable to I/O operations.

QUESTION

Is DECLARE F RECORD; OPEN FILE(F) PRINT; valid?

ANSWER

Although most of the declarative structure is evident in the Abstract Syntax and checked by the Translator, this is not altogether true of file attributes since they may still be incomplete until the file is opened during execution. Therefore their valid combinations have to be tested also at this late stage.

The OPEN operation (Section 8.6.1.3) shows that `<print>` implies `<stream>` in Step 2, and that `<stream>` and `<record>` cause the result `<fail>` to be returned (Step 7) from the attempt to open the file. Thus if the `<open-statement>` is executed, it will lead to the raising of the UNDEFINEDFILE condition when an attempt is made to open the file (see Step 6 of EXECUTE-SINGLE-OPENING, Section 8.6.1.2).

Chapter 9 describes the evaluation of expressions, also conversions between data types, which can take place in expression evaluation, and the built-in functions of PL/I.

QUESTION

Is it possible to add two arrays?

ANSWER

The evaluation of expressions in Chapter 9 begins by considering the general treatment of aggregate operands. Investigation of the INFIX-ADD operation (9.4.2.1) and the ADD-BIF (9.5.4.3) indicates that the operands of the addition operator or the ADD built-in function must be scalars. Thus two arrays may not be added together, except by using an explicit loop to add each element.

Appendix A describes the features of this standard that extend it from its earlier version by adding language constructions present in ANSI X3.53-1976.

Appendix B describes the features of this standard that represent new language — features not present in either the earlier version of this standard ANSI X3.74-1981 or in ANSI X3.53-1976.

Appendix C describes clarifications and restrictions new to this version of the standard. These clarifications and restrictions are more likely to affect processors for the language than practical applications programs written in it.

Appendix D describes the features of ANSI X3.53-1976 that have not been incorporated in this standard and describes a difference in the semantics of the two standards.

Appendix E lists changes to the language described in ANSI X3.74-1981 that are incompatible, i.e., changes that will have to be made to programs conforming to that standard to make them operate correctly with this one.

Appendix F gives the rationale for the major decisions in the design of the language specified in this standard.

1.4.4 Introduction to the Metalanguage

An informal discussion of the main features of the metalanguage is now offered before proceeding to its more rigorous definition in Section 1.6.

1.4.4.1 Tree Concepts

The definition of PL/I deals with three classes of trees (concrete parse, abstract text, and machine-state), and a uniform concept of tree is employed throughout. This is of a tree which is a directed graph with a label (e.g. <procedure>) at each node, and where the subtrees of any node are ordered. (Although the ordering is often irrelevant, it is needed, e.g., in the concrete parse and in lists, and it was decided that the simple uniformity of concept outweighed the advantages of explicitly distinguishing instances where the order was significant.) Moreover, each node implicitly has a unique-name by which it can be “designated” when required. A copy of the tree has the same ordered structure and labelling, but new unique-names to designate its nodes. Equality of trees requires equality of all except the unique-names of nodes, and identity requires that these also match.

The explicit label of a node may be either a grammatical category-name, or some other type of value such as an integer or a designator (i.e. a copy of the unique-name implicitly associated with some node). Thus a single value may be handled as a degenerate tree with only a root-node labelled with this value, and data objects referenced in the metalanguage are uniformly regarded as trees.

The terminology applied to trees is developed from the starting-point of a tree consisting of a node (the *root-node*) and an ordered, possibly empty, set of *immediately contained* trees.

A tree, X, is said to be *contained* in a tree, Y, if X is immediately contained in Y, or if X is immediately contained in some tree contained in Y. X is *simply contained* in Y if and only if it is contained in Y and it is not contained in any tree, Z, contained in Y and having a root-node equal to that of X or of Y. For example, if we refer to an <expression> simply contained in a tree representing some statement in a program, we mean the complete <expression> tree and not some subexpression which might be a tree rooted in <expression> at some lower level within it. Since simple containment is a frequently required concept, the use of any form of possessive phrase not employing the verb “contain” or the noun “component”, is taken to imply simple containment, e.g. “if y has an <expression>” (meaning a tree with root-node labelled <expression>), “the <expression> of y”, or “its <expression>”.

The terminology makes it possible to perform the three essential types of operation on trees, namely, to test them for the presence or type of their subtrees, to select subtrees from them, and to construct or alter them. However, construction can be laborious if phrased as “root-node <a> with two immediate subtrees, the first being with immediate subtrees <c> and <d>, and the second being x” (where x is the name of some tree, meaning that a copy of it is to be made in constructing the new tree). This may therefore be abbreviated as

```

<a>:
  <b>:
    <c>
    <d>;
  x;

```

The indentation is inessential, although helpful with large trees. One or more trailing semicolons at the end of such a constructed tree may optionally be replaced by a period.

1.4.4.2 Syntaxes

Proceeding from consideration of particular trees to classes of trees, we encounter grammars composed of sets of production-rules couched in a slightly extended Backus Naur Form. The interpretation we make of such rules is that they describe the structure of a tree belonging to the class described by the grammar — it is only in the traditional syntactic use of these rules that the sequence of terminal nodes of a tree acquires a special importance as being the sequence of characters representing an utterance in the language.

As an illustration, we will construct a tree that conforms to a rule of the form

$$\langle p \rangle ::= \langle q \rangle \mid \{ \langle r\text{-list} \rangle \mid \langle s \rangle \} \bullet \langle t \rangle [\langle u \rangle]$$

A grammatical rule in the metalanguage is rather like a statement in a language which it may be used to define, i.e. a metalanguage rule itself has a grammatical structure which has to be validated and correctly interpreted. We have chosen to escape from circularity or regress at this point by not giving a formal grammar for the metalanguage, but describing in prose how a rule is to be interpreted. So in the above illustration, the rule indicates that any node labelled $\langle p \rangle$ must have subtrees as described by the right-hand side of the rule.

In descending order of precedence there are three types of metalanguage operators:

1. for concatenation (indicated by juxtaposition in the rule),
2. for permutation (“•”), and
3. for a choice between alternatives (“|”).

Expressions within braces are regarded as single operands, and the first stage in using a rule to construct a tree is to partition the syntax expression into subexpressions separated by “|”, the operator of lowest precedence, and to choose one of these alternative subexpressions. Suppose we discard the alternative of having $\langle q \rangle$ alone, and next partition the other alternative by the “•” operator and decide to use the permutation which reverses these partitions to yield

$$\langle t \rangle [\langle u \rangle] \{ \langle r\text{-list} \rangle \mid \langle s \rangle \}$$

The brackets indicate that their contents may be optionally omitted, which we choose to do here, and the braces enclose a syntax expression which is interpreted according to the method just described. Choosing the alternative $\langle r\text{-list} \rangle$ we would then have the tree

```

<p>:
  <t>
  <r-list>;

```

The underlining of t indicates that it always labels a terminal node in the grammar in question, i.e. it does not occur on the left-hand side of any production-rule. $\langle r\text{-list} \rangle$ indicates by convention that it is to have one or more immediate subtrees of type $\langle r \rangle$ beneath it. We then apply the rule for $\langle r \rangle$ separately to each of these. The construction is completed when each terminal node of the tree is either of a type which is always terminal or of a type which has produced an empty set of subtrees.

1.4.4.3 Algorithm Concepts

The operations of the abstract machine are defined by algorithms, which may be compared with the logic or microcode supporting the operation codes of a computer. These operations inspect the $\langle \text{machine-state} \rangle$ or change the $\langle \text{machine-state} \rangle$, or both, i.e. the memory of the abstract machine, which contains all the information directly or indirectly affecting semantics, including some form of the $\langle \text{program} \rangle$ itself. (The metabracquets “ $\langle$ ” and “ $\rangle$ ” are used in the PL/I definition of the machine-state nodes, except for those belonging to the grammar of the abstract program which are distinguished by “ $<$ ” and “ $>$ ” to help the reader.)

An operation of the machine is defined by a sequence of Steps, or a set of mutually exclusive Cases, numbered from 1 to n . The i th Step or Case may itself contain a sequence of Steps or a set of Cases numbered from $i.1$ to $i.m$, and so forth. Each Step or Case specifies actions to be performed, using various informal “statement types” in the metalanguage. A Case begins by stating the predicate that must be satisfied for it to be applicable.

As an example of an operation, consider the definition of the PROLOGUE operation in Figure 2 on page 13. This begins with an introductory paragraph which gives some guidance to the reader (and that is not part of this standard), but the definitive material is not reached until the “Operation:” heading.

Step 1 consists of an *attach* action, meaning that one tree is to be copied as a subtree of another in the position in which it may validly occur according to the syntactic rules governing the type of the latter. The word “current” has been defined precisely with respect to the particular $\langle \text{machine-state} \rangle$ used in defining PL/I - the “current linkage-part” is the linkage-part simply contained in the “block-state” corresponding to the block currently being executed. Step 2 indicates iteration with a *for each* specification applied to the first form of the *perform* action, which refers to one or more Steps in the same operation. A name such as d introduced after a comma is a variable, local to the operation, whose value (v , say) is a designator of the tree mentioned immediately preceding the “ d ”. On subsequent uses of d , it is dereferenced, i.e. it means “the tree designated by v ” except when it is redefined in a way similar to its original definition (as would happen here on the next iteration of “for each”), or by its occurrence after the word “let”. The *let* statement is merely a more explicit syntax for this kind of definition of a local variable, and Step 2.1 contains an instance of this. If d contains $\langle \text{automatic} \rangle$, the predicate of Case 2.1.1 is satisfied and Steps 2.1.1.1 to 2.1.1.3 will be performed in sequence.

The second form of *perform*, which is used to invoke another operation as a subroutine or function, occurs in Step 2.1.1.1. Here it is a function that is invoked “to obtain” a resulting value, which is then given a name. edd is passed as an argument to this operation, whose definition will name a parameter to correspond to it. Argument passing is by reference, i.e. a designator to the argument is passed and becomes the value of the parameter, which is then dereferenced on use, behaving just like a local variable.

The *append* action in Step 2.1.1.2 attaches a tree at the end of a list, and Step 2.1.1.3 exemplifies an *if* statement containing a subroutine call. Steps 3 and 4 consist of the other two tree actions which we use, *delete* and *replace*. When a tree is replaced by another, the first tree is deleted and a copy of the second tree is made with its root at the same position, and having the same implicit name, as had the root of the tree just deleted. Assignment of a value to a variable is permitted in the syntactic form of a *set* action, e.g. “Set tv to $\langle \text{true} \rangle$ ”, but this is only an alternative form of replacement with the usual implied dereferencing of the variable name, meaning “replace the tree designated by the strict designator value of tv by $\langle \text{true} \rangle$ ”.

Other statement types not illustrated here are *go to* (used sparingly), *terminate* an operation at some point other than the end of the last Step, and *return* a value from a function (which also terminates the operation). Values are returned by reference, so that if the result is a tree constructed within the function it MUST be copied back to the caller who will then receive a designator of the copy; values local to an operation are deleted when it terminates.

We have now reached a point at which it is necessary to indicate a mechanization of the metalanguage which will suffice to bolster intuition in its weaker moments. Step 4 provides the cue for this, since it constructs a tree

6.3.3 Prologue

This operation is invoked at the beginning of every block activation to establish the <automatic> and <defined> variables local to that block. The <automatic> variables are initialized if their <declaration>s specify initialization. Any <expression>s evaluated during the prologue, such as in <extent-expression>s or <expression>s in <initial>, are not allowed to reference other <automatic> or <defined> variables local to this block. The operation FIND-DIRECTORY-ENTRY will impose the restriction when it finds a reference to a variable declared in a block for which there exists a <prologue-flag>. The <prologue-flag> is only present while the operation PROLOGUE is active.

Operation: PROLOGUE

- Step 1. Attach a <prologue-flag> to the current <linkage-part>.
- Step 2. For each <declaration>, d, of the current block, that contains <automatic> or <defined>, perform Step 2.1
 - Step 2.1. Let id be the <identifier> immediately contained in d, and let dd be the <data-description> immediately contained in the <variable> of d. Perform EVALUATE-DATA-DESCRIPTION-FOR-ALLOCATION(dd) to obtain an <evaluated-data-description>, edd.
 - Case 2.1.1. d contains <automatic>.
 - Step 2.1.1.1. Perform ALLOCATE(edd) to obtain a <generation>, g.
 - Step 2.1.1.2. Append to the current <automatic-directory-entry-list> an <automatic-directory-entry>: id g.
 - Step 2.1.1.3. If d contains <initial> then perform INITIALIZE-GENERATION (g,d).
 - Case 2.1.2. d contains <defined>.
 - Append to the current <defined-directory-entry-list> a <defined-directory-entry>: id edd.
- Step 3. Delete the <prologue-flag> of the current <linkage-part>.
- Step 4. Replace the current <statement-control> by a
 - <statement-control>:
 - <operation-list>:
 - <operation> for ADVANCE-EXECUTION.

Figure 2. The Prologue Operation.

which is to be placed in the <machine-state> at exactly the point at which it tails off into informality. Of the <operation-list>s in the <machine-state>, just one is said to be “active” and has a conceptual processor associated with it which may execute operations. In particular, the PL/I machine has a current <block-state> which has a <statement-control> where one could mechanize the actions of the metalanguage involved in interpreting the statements of the current block.

The <operation-list> is to be conceived as a push-down stack, where each <operation> will have a subtree which is not formally defined, but contains such information as the name of the operation, a list of its

parameters and local variables with their current values, all trees constructed during execution of the operation or copied back to it from invoked functions, an indication of whereabouts in which Step or Case it is executing its algorithm, how far it has proceeded through a “for each” iteration, and so forth. When a “perform” invokes another operation, this pushes down the stack and becomes the active operation; when an operation terminates, its whole tree (including its local values) is deleted and the stack pops up, activity in the operation now at the head of the stack being resumed immediately after the point in its algorithm at which it was suspended.

To return to the analogy suggested at the beginning of this section, it is as though an abstract processor had a machine cycle which could cause the execution of microcoded operations with their own local memories per invocation, together with the sophistication of the stacking capability.

1.5 Relationships between an Implementation and this Definition

The inputs of the conceptual PL/I machine are one or more $\langle \text{symbol-list} \rangle$ s representing PL/I translation units, an $\langle \text{entry-value} \rangle$, and a $\langle \text{dataset-list} \rangle$. This combination will be referred to as a *program-run*.

The standard definition of PL/I for a particular implementation is completed by defining (not necessarily in the style of this document) the IMPLEMENTATION-DEFINED features listed in Section 1.5.2, together with the representation of a program-run’s elements ($\langle \text{symbol-list} \rangle$ s, $\langle \text{entry-value} \rangle$, and $\langle \text{dataset-list} \rangle$) in the implementation’s operating environment. With this information available, the conceptual PL/I machine gives one or more interpretations to a program-run.

The main purpose of this document is to define the semantics of interpreting valid PL/I program-runs. These semantics are constituted solely by the sequence of changes in the $\langle \text{dataset} \rangle$ s of the $\langle \text{machine-state} \rangle$, and an implementation is free to achieve them by any means. The operations and other parts of the $\langle \text{machine-state} \rangle$ which are the mechanisms used in this document to define the semantics need not be reflected directly in implementation.

An implementation’s interpretation of a program-run conforms to this standard if and only if it conforms to one of the conceptual interpretations as follows:

1. If the conceptual interpretation rejects a program-run (via failure of a MUST test) or if it never completes the translation-phase, then any interpretation by the implementation conforms. In particular, an implementation may or may not reject a program-run at the same point as it is rejected by this standard, or at all.
2. Otherwise, the implementation’s interpretation conforms if it makes the same sequence of changes to datasets as does the conceptual interpretation.
3. The implementation’s interpretation also conforms if it deviates from (1) and (2) only as permitted by the flexibilities of interpretation specified in Section 1.5.1.

Note: This implies that an implementation may provide extensions beyond the language defined in this standard, but is still required to conform for a program not using those extensions just as if the extensions were not available.

1.5.1 Flexibilities of Interpretation

Through use of the terms “optionally”, “implementation-dependent”, and “in any order”, the operation definitions of the conceptual PL/I machine permit most of the flexibility necessary for efficient implementation of PL/I. However, there are some rules that, if given formally with a metalanguage of the kind used here, would require constructions so elaborate as to impede understanding seriously. These rules are given in this section using informal language and making direct reference to possible actions of an implementation.

Note: “implementation-dependent” and IMPLEMENTATION-DEFINED are not the same. IMPLEMENTATION-DEFINED features are given a specific definition, and requirements are imposed on them, in Section 1.5.2.

1.5.1.1 Rejection of Programs

If some part of a program is such that its interpretation would cause the program to be rejected, then an implementation may reject the program even if the conceptual interpretation does not reach the offending part.

1.5.1.2 Quantitative Restrictions

An implementation may make quantitative restrictions not contained in this standard. For example, restrictions may be made in the following contexts:

1. Where syntaxes allow iterative or recursive constructs of arbitrary size, an implementation may restrict the size of these constructs provided the construct is not deleted. In particular, a standard implementation may limit the maximum length of an `<identifier>`, provided it is not less than 31 characters.
2. The quantity of information existing during translation or execution may be restricted.
3. The time permitted for interpretation of a program-run may be bounded.

1.5.1.3 Addressing Restrictions

An implementation may make restrictions on the use or addressability of “unaligned” objects (an UNALIGNED scalar or an aggregate which contains no ALIGNED scalars) in any of three contexts:

1. An argument to the ADDR built-in function.

For example, an implementation might require that a `<reference>` to a variable that has the attribute BIT appearing in an argument to the ADDR built-in function have the attribute ALIGNED.

2. Values transmitted to or from `<dataset>`s in RECORD I/O.

For example, an implementation might require the `<reference>` in a FROM or INTO option, or the `<generation>` to be accessed from the `<reference>` in a SET option, to have the attribute ALIGNED.

3. A `<generation>` and a based `<variable-reference>` used to access the `<generation>` do not have identical `<data-description>`s and are not suitable for based string overlaying (before pruning of the `<data-description>` or the `<variable-reference>`, see Step 3.1.3 of CHECK-BASED-REFERENCE).

For example, an implementation need not support the construction `ADDR(S) -> T` when S and T were declared as follows:

```

DECLARE 1 S,
        2 A BIT UNALIGNED,
        2 B CHARACTER UNALIGNED,
        2 C FIXED BINARY,

        1 T BASED,
        2 A BIT UNALIGNED,
        2 B CHARACTER UNALIGNED;

```

1.5.1.4 Operating Environment

An implementation may make restrictions at the interface with its environment, for example, in the composition of external identifiers, external names, or titles.

An implementation may require appropriate extralingual information in order to execute a program in conformance with this standard.

1.5.1.5 Character Sets

This standard requires that an implementation support primary and secondary cases for letters in the character set (see Section 2.6). Typically, some permutation of the lowercase and uppercase letters in the implementation's character set are used for the primary and secondary case letters of the PL/I character set. However, this standard does not specify the mapping of the primary and secondary case letters (or any other characters in the PL/I character set) on to specific graphic characters nor does it specify their hardware representation (see Section 1.5.2, items 14 and 15). Various operations treat corresponding primary and secondary case letters as equivalent in particular contexts. In addition, a program intended for interchange must not depend on whether or not Step 6.1 of HIGH-LEVEL-PARSE is performed.

1.5.1.6 Expression Evaluation

The operations "EVALUATE-EXPRESSION", "EVALUATE-VARIABLE-REFERENCE", and "EVALUATE-TARGET-REFERENCE" define one or more strict orders of evaluation. An implementation may deviate from these strict orders in the following respects:

1. Not all of the interrupts raised by any of the strict evaluations need be raised by an implementation.
2. The order in which interrupts are raised by an implementation may differ from the order in which they are raised by strict evaluation.
3. An implementation may evaluate an <expression> (without an <and-logical> or <or-logical>), <value-reference>, <variable-reference>, or <target-reference> by evaluating its contained <expression>s, <value-reference>s, and <variable-reference>s in any order.

Note: This does not permit the association of operators with their operands to be altered, since the association is fixed in the tree structure of an <expression> by the Translator.

4. If an implementation can determine the result produced by evaluating an <expression>, <variable-reference>, or <target-reference> without evaluating it, the implementation need not perform the evaluation. However, an implementation is required to evaluate any <procedure-function-reference>s which are evaluated in the strict order of evaluation.

An implementation is permitted to ignore the possibility that an evaluation could raise one or more interrupts when determining whether it is required to perform the evaluation.

1.5.1.7 Interrupts and Assignment

The operation “EXECUTE-ASSIGNMENT-STATEMENT” determines a strict order of evaluation. An implementation may deviate from one or more strict orders in the number and order in which it raises interrupts as described in Section 1.5.1.6, parts 1 and 2. When an implementation exceeds one or more of its IMPLEMENTATION-DEFINED limits, it may raise one or more of the following conditions: overflow, underflow, fixedoverflow, zerodivide, size, storage, area, error.

1.5.1.8 Input/Output

1. The <environment> attribute and option are provided for the specification of IMPLEMENTATION-DEFINED information concerned with the manipulation of the <file-value>s and <dataset>s. If present, an <environment> may affect the algorithms described in this standard by either causing (if necessary) the operation EVALUATE-EXPRESSION to be performed or affecting the correspondence between open <file-value>s and <dataset>s. An <environment> option to an OPEN statement, a CLOSE statement, or one of the input-output transmission statements may also affect the semantics of these statements, including the sequence in which values are transmitted, additional or alternate processing to be performed, or default positioning before or after operations. An <expression> appearing in an <environment> is evaluated at an IMPLEMENTATION-DEFINED point. If no <environment> is present, the algorithms work as given.
2. If there is more than one <single-opening> in an <open-statement>, an implementation may deviate from the strict order of execution of the RAISE-IO-CONDITION(<undefinedfile-condition>,fv) operation performed by the EXECUTE-SINGLE-OPENING operation. In particular, an implementation may defer the RAISE-IO-CONDITION operation(s) until all other operations in the statement have been performed. However, when the conditions are raised, it is required that they be raised in the same sequence implied by the strict order of execution.
3. During the execution of any input/output statement which contains a <file-option> an implementation may raise the <transmit-condition>. Continued execution of the program beyond the point where the condition is raised may be undefined depending on the use and validity of the data transmitted by the input/output statement. The <transmit-condition> is raised by performing RAISE-IO-CONDITION(<transmit-condition>,file-value,key) where file-value and key depend on the input/output statement.
4. A <get-string> MUST NOT immediately contain an <expression> that simply contains a <variable-reference> that identifies an <allocation-unit> that is referenced by the evaluation of the <input-specification> of the <get-string>.
5. A <put-string> MUST NOT immediately contain a <variable-reference> that identifies an <allocation-unit> that is referenced by the evaluation of the <output-specification> of the <put-string>.

1.5.1.9 On-units

In order to avoid defining certain “side-effects” of <on-unit>s, and to avoid overly defining the state of the machine upon entry to <on-unit>s, a program MUST satisfy the following constraints not explicitly enforced by the PL/I machine:

1. An <on-unit>, or any of its dynamic descendants, entered for the underflow condition MUST NOT allocate, free, or assign a value to any <allocation-unit> used in the block of interrupt, unless the <on-unit> terminates by executing a <goto-statement>.

Let B be the current $\langle\text{block-state}\rangle$ at the time the interrupt was raised; i.e. the block of interrupt. An $\langle\text{allocation-unit}\rangle$ is *used* in block B if it is referenced by any operation of the PL/I machine while B is the current $\langle\text{block-state}\rangle$.

2. Let B be a $\langle\text{block-state}\rangle$; let E be an $\langle\text{executable-unit}\rangle$ executed while B is the current $\langle\text{block-state}\rangle$. During the execution of E, various $\langle\text{generation}\rangle$ s are returned by invocations of EVALUATE-TARGET-REFERENCE. Let T be the set of $\langle\text{allocation-unit}\rangle$ s designated by these $\langle\text{generation}\rangle$ s, i.e., the targets of a given statement execution.

During the execution of E whenever the INTERRUPT operation is invoked, all $\langle\text{basic-value}\rangle$ s contained in any member of T are set $\langle\text{undefined}\rangle$. If control reaches Step 4 of the INTERRUPT operation, the original $\langle\text{basic-value}\rangle$ s are restored, unless their containing $\langle\text{allocation-unit}\rangle$ was assigned one or more $\langle\text{basic-value}\rangle$ s by the $\langle\text{on-unit}\rangle$ or its dynamic descendants. In the case of such assignment, the $\langle\text{basic-value}\rangle$ s of the $\langle\text{allocation-unit}\rangle$ are set $\langle\text{undefined}\rangle$.

1.5.2 Implementation-defined Features

The PL/I language features listed below are termed IMPLEMENTATION-DEFINED : their specification is regarded as completing the definition of the language for a particular implementation. A brief description of each feature is given, with references in parentheses to the sections of this document where further details can be found.

1. The actual restrictions imposed, if any, under the provisions for restrictions on the size of constructs (item 1 of 1.5.1.2) or in addressing (items 1, 2 and 3 of 1.5.1.3).
2. Whether or not case mapping is performed on $\langle\text{identifier}\rangle$ s (Step 6.1 of 4.3.2).
3. The mapping functions for $\langle\text{external-name}\rangle$ s and $\langle\text{identifier}\rangle$ s to $\langle\text{environment-name}\rangle$ s (4.8.2).
4. Determination of the $\langle\text{dataset-list}\rangle$ passed to the INTERPRET operation (1.7.3.3).
5. Determination of the $\langle\text{entry-value}\rangle$ passed to the INTERPRET operation (1.7.3.3).
6. ENVIRONMENT attribute and option syntax (2.4.11.4) and semantics (Chapter 8 and item 1 in 1.5.1.8).
7. OPTIONS attribute and option syntax (2.4.11.4) and semantics.
8. Consistency requirements for ENVIRONMENT and OPTIONS attributes in EXTERNAL declarations (4.8.1), and for OPTIONS attributes in ENTRY references (6.4.7.1.1).
9. The form of the $\langle\text{text-name}\rangle$ in the INCLUDE construct (2.5.6).
10. Interpretation of UNSPEC pseudo-variable (7.6.4.6).
11. Result of UNSPEC built-in function (9.5.4.81).
12. Length of string returned by TIME built-in function (9.5.4.77) and of the DATETIME built-in function (9.5.4.21).
13. Extralingual characters in data character set (2.5.5 and 2.6.2).
14. Collating sequence, hardware representations, graphic representations, and symbol names of an implementation's character set (2.6 and 9.5.4.14) and the hardware representations of the control character sequences returned by the ISOCHAR built-in function (9.5.4.38).

15. The external hardware representations of the characters specified by $\langle \text{symbol} \rangle$, i.e., the set of characters in which PL/I programs are written (2.6).
16. Representation of currency symbol and digit and sign symbols (9.6.2.2).
17. Default precisions of arithmetic data (4.5.5.3).
18. Maximum $\langle \text{number-of-digits} \rangle$ for each combination of $\langle \text{base} \rangle$ and $\langle \text{scale} \rangle$ (4.6.3.5, 9.2.3.2, 9.6.1.9, and elsewhere).
19. Precision of integer-type (9.2.2).
20. Accuracy and range of floating-point results of expressions and built-in functions (1.6.3.7.2, 9.2.4, and elsewhere).
21. Accuracy and range of ROUND built-in function with floating-point argument (9.5.4.61).
22. Results of numeric conversions (9.6.1.2, 9.6.1.3, and 9.6.1.6).
23. Number of digits in the exponent of a floating-point number (9.6.1.5).
24. Results of the MAXVAL, MINVAL, EPSILON, and RADIX arithmetic-enquiries (9.2.10).
25. Result of the EXPONENT built-in function and effect of the EXPONENT pseudo-variable (9.5.4.29 and 7.6.4.1).
26. Default AREA size (4.5.5.3).
27. Size of an $\langle \text{area-value} \rangle$ passed as a dummy argument (6.4.7.1.2).
28. Use of AREA size specification (7.3.4).
29. Information output when SNAP is specified in ON statement (6.5.3).
30. Form of comment output as standard system action (6.5.4.1).
31. Value returned by ONCODE built-in function (6.5.3).
32. Standard system action for STORAGE condition (6.5.4).
33. Standard system action for ERROR condition (6.5.4).
34. Circumstances in which TRANSMIT condition is raised (1.5.1.8).
35. Circumstances in which the KEY condition is raised (8.6.2.3, 8.7.2.2, and 8.7.5.10).
36. Circumstances under which records are written, or not written, when the RECORD condition is raised and normal return occurs, and the values of those records (8.7.3.2, and 8.7.5.9).
37. Raising of RECORD condition by WRITE and REWRITE statements (8.7.3.2 and 8.7.5.9).
38. Concrete representation of a $\langle \text{dataset} \rangle$ (8.2).
39. The “size” of a $\langle \text{record} \rangle$ (8.2.1).
40. Length of a key (8.2.1).

41. Representation of stream dataset control items (8.2.2).
42. Determination of a <dataset> on file opening (8.6.1.3).
43. Length of file title (8.6.1.4).
44. Default LINESIZE for a STREAM OUTPUT file (8.6.1.3).
45. Default PAGESIZE for a PRINT file (8.6.1.3).
46. Default tab positions for a PRINT file (8.6.1.3).
47. Position of records in a KEYED SEQUENTIAL file (8.7.5.9).
48. Maximum <number-of-digits> used in editing relative to a <fixed-point-format> (8.8.2.5.3).

1.6 The Metalanguage

Following the introductory material in Section 1.4.4, this section now gives a more precise and careful definition of the metalanguage.

The definitive part of this document consists of:

- a set of production-rules
- a set of operations
- constraints
- attribute definitions and argument names
- tables
- definitions of terms

together with the section describing the relationship between an implementation and this mechanized definition.

The metalanguage in which the definition is expressed has three main notational parts, to be presented later in this section:

- a notation for trees, the fundamental type of data in the metalanguage;
- a notation for production-rules, which define classes of trees;
- a notation for operations, which manipulate trees.

Other definitive material follows headings “Constraints:”, “Attributes:”, or “Arguments:”.

At the point where a new term (other than a syntactic category) is defined, the term is italicized to indicate this; subsequent uses of the term are not italicized.

Tables are a part of the definition and are set off from the body of the text. Small tables may not have captions. Larger tables have a figure caption which identifies them as tables. Other figures are examples and are not part of the definition. Introductory paragraphs to syntaxes and operations, and explanatory notes (denoted by “*Note:*”) are likewise without any definitive force.

1.6.1 Trees

Trees are the sole type of data manipulated by the actions of the process defined by this document. All of the internal operations of the process use only trees, all of the inputs to the process from its environment are suitably constructed trees, and all interpretations of the semantics defined by this process are in terms of the tree manipulations performed by it. For uniformity even simple values, such as numbers or characters, are regarded within the process as single node trees.

In a strict mechanization of the process defined by this document, there could in fact be only a single tree used to hold the entire “state” of the process, and all of the trees discussed here would be subtrees of this single tree. Since, however, this document leaves certain informal “gaps” in its tree definitions, it is also possible to regard the process as one that operates on a set of independent trees, one for the “state” and others which are more local to particular phases of the definition.

The general abstract form of trees as employed in this document plus the technical terms used in discussing trees are defined in Section 1.6.1.1. These trees are then made more specific by discussing in Section 1.6.1.2 the basic nature of the objects used in composing tree nodes. In Section 1.6.1.3 and Section 1.6.1.4 the written notations used for individual nodes and then for whole trees are discussed. Section 1.6.1.5 deals with copies of trees.

Following this general section on trees and tree notations, Section 1.6.2 then discusses production-rules, which function in a declarative manner to specify the particular classes of trees used in this document.

1.6.1.1 Tree Definitions

A *node* is an ordered pair of objects, termed the *type* of the node, and the *unique-name* of the node.

A *tree* is a finite set of one or more nodes together with some structuring relationships among these nodes. These relationships are such that:

1. There is a specified node termed the *root-node* of the tree.
2. Excluding this root-node, the remaining nodes (termed the *subnodes* of the tree) are divided up into zero or more disjoint sets, each of which in turn forms a tree. These trees are termed the *immediate subtrees* of the defined tree.
3. There is a specified linear ordering among these immediate subtrees.

A tree, X, is said to be a *subtree* of a given tree, Y, if X is either:

1. an immediate subtree of Y, or
2. a subtree of an immediate subtree of Y.

A subtree, X, of a given tree, Y, is said to be a *simple subtree* of Y if there does not exist a tree, Z, such that:

1. Z is a subtree of Y,
2. X is a subtree of Z, and
3. the root-node of Z has the same type as either the root-node of X or the root-node of Y.

Terminology based on the word “contained” is used consistently as follows: subnodes and subtrees are said to be *contained* in the given tree, and immediate subtrees and simple subtrees are said to be *immediately contained* and *simply contained* respectively. Similarly for the word “component”, i.e. subnodes and subtrees are said to be *components* of the given tree, immediate subtrees and simple subtrees are said to be *immediate components* and *simple components* respectively. The root-nodes of immediate subtrees and simple subtrees are said to be *immediate subnodes* and *simple subnodes* respectively.

The concept denoted by the term “simply contains” is used so frequently in the sequel that the words themselves are usually elided. Any relational statement between two nodes that implies containment or possession, without using explicitly any form of the words “contain” or “component”, is to be interpreted as implying the simple containment relation. For example, “A simply contains B” may be expressed as “A with a B”, “A has B”, “the B in the A”, or even just “B A”. These abbreviated forms may also be compounded in a single sentence.

Two trees are said to be *equal* if they contain the same number of nodes and if:

1. when they each contain a single node, their respective types are the same, or
2. when they each contain more than one node, the respective types of their root-nodes are the same, they have the same number of immediate subtrees, and their respective immediate subtrees, taken pairwise, are equal.

A tree, X, is said to *immediately follow* a tree, Y, if they are both immediate subtrees of some tree, Z, and if X is next after Y in the linear ordering of the subtrees of Z.

A tree, X, is said to *immediately precede* a tree Y if Y immediately follows X.

A tree, X, is said to *follow* a tree, Y, if any of the following is true:

1. X immediately follows Y.
2. There is a tree Z such that X follows Z and Z follows Y.
3. There is a tree Z such that Z contains X and Z follows Y.
4. There is a tree Z such that Z contains Y and X follows Z.

A tree, X, is said to *precede* a tree, Y, if Y follows X.

Note: The above definitions do not define a linear ordering on all of the subnodes of a tree, just a partial ordering. In particular, any tree which contains another does not either precede or follow the contained tree.

The words *first* and *last* applied to any distinct set of trees have their usual sense of “have none that precede” and “have none that follow”. Although these definitions are not such as to give a unique first or last tree for some sets of trees, “first” and “last” will be applied in this document only to sets such that there is a unique result.

Similarly, the *next* tree following a given tree is defined in the usual sense of “first that follows” and will also be used in contexts where it is unique.

A node, M, contained in a tree, X, is said to *correspond* to a node, N, contained in a tree, Y, if M and N occupy the same ordinal position in the ordered set of immediate subtrees of either:

1. the root-nodes of X and Y, or of
2. corresponding nodes of X and Y.

Note: Chapters 2 (see Section 2.6) and 9 (see Section 9.2.1.5) contain further definitions of “correspond” useful in certain special contexts.

1.6.1.2 Node Objects

The definition of node given in Section 1.6.1.1 leaves undefined the nature of the objects used for node types and node unique-names. The sets from which these objects are selected are limited as described in Section 1.6.1.2.1 and Section 1.6.1.2.2

1.6.1.2.1 Unique-names

The set of objects which may be employed as node unique-names plays two roles. All node unique-names are selected from this set, but in addition, some node types may be selected from this set. Any potentially infinite set of objects which are distinguishable from the other objects used as node types will suffice.

The unique-name component of a node is so called for two reasons. First, at no time do two distinct nodes have equal unique-name components. Second, no node is ever created with a unique-name component equal to that of any node which has ever been created. The unique-name component of a node does not change during the life of the node and thus serves to identify, or designate, the node (see Section 1.6.1.2.2).

1.6.1.2.2 Types

The set of objects which may be employed as node types is the union of the following disjoint sets:

1. The set of *category-names*. This is a finite set of the objects employed in production-rules. This set can be further subdivided into several logically coherent subsets, each with a definite notational convention.
2. The set of real numbers, or possibly some implementation-dependent subset of them which includes the integers. The integers are used throughout for such purposes as indices and ordinals. Real numbers (including possibly integers) are used as the values of arithmetic variables.
3. The set of unique-names, as defined in Section 1.6.1.2.1. A member of this set used as a node type is termed a *designator*. Designators are used explicitly for the purpose of uniquely picking out, or designating, nodes of a tree. A designator (or a tree containing a single designator) *designates* exactly that node which has the same object as unique-name.

Note: This construction, together with the uniqueness rule in Section 1.6.1.2.1, means that it is possible to examine a designator and constructively determine if the potentially designated node has been in fact deleted.

As a general rule, the type of a node does not change during the life of the node. Modifications to the tree occur by removing old nodes and constructing new nodes with new types and new unique-names. The single exception to this is the replace instruction (see Section 1.6.3).

1.6.1.3 Node Notation

Throughout the metalanguage, the unique-names of nodes have only an implicit and essentially invisible function in guaranteeing unique designation and proper subtree distinction. The metalanguage discussion of nodes is always in terms of their types. Particular objects from the set of unique-names are never referred to directly, so no written notation for them is required.

The written notation used for real numbers is just ordinary decimal notation throughout. In the sequel, context is sufficient to distinguish numbers used as node types from numbers used for other purposes.

The written notation used for category-names varies, depending on the logical nature of the use of the particular category-name. These various notations and their meaning is as follows:

1. **Named Categories.** Names formed of lowercase letters, hyphens, and numbers, including surrounding brackets of the form “ $\{$ ” and “ $\}$ ”, “ $<$ ” and “ $>$ ”, or “ $\Leftarrow$ ” and “ $\Rightarrow$ ” are used as the denotation of some category-names. Optionally the name exclusive of the brackets may be underlined. In general, mnemonic English words or abbreviations are chosen to indicate the function of the category-name. In addition, the three types of brackets indicate whether the category-name functions primarily in the concrete, abstract, or interpretation phases of the definitional process respectively. The underlining, if present, indicates that the category-name occurs only as the type of a node that has no components.

2. PL/I Characters. The 83 characters of the PL/I language character set are used as category-names. Two denotations are used. In the great majority of situations, where no confusion is liable to arise, they are denoted by straight-forward individual character denotations. Uppercase or lowercase letters are used for primary and secondary case letters, respectively, while the quoted symbol (i.e. that which is inside the following quotes) “Ø” is used for blank. In situations where confusion might arise, the concrete brackets are used around the straightforward denotations, e.g. <A>, <a> or <A | | >. In some cases, primary and secondary case letters are treated as if they were equivalent, see Section 2.6.3.
3. PL/I Keywords. Certain category-names represent PL/I keywords, i.e. selected sequences of letters or numbers that have particular significance in PL/I. Two denotations are used. In the great majority of situations where no confusion is liable to arise, they are denoted by a straightforward concatenation of the denotations for the individual letters or digits that form these keywords in PL/I, written without intervening spaces on the page. Examples are the quoted symbols “ENTRY” and “FLOAT”. In situations where confusion might arise, the concrete brackets are used around the straightforward denotations. For these keywords, while primary case letters are always used in the metalanguage, secondary case letters are equivalent in the actual keywords, see Section 2.7.

Nodes which have a type of either of the classes (2) or (3) above are said to possess a *concrete-representation*, which is a (non-tree) character string composed from any of the 83 PL/I language characters. For these nodes this concrete-representation is just the simple denotation of the node type, with a blank space substituted for Ø. Any possible subnodes of the category <extralingual-character> are assumed to have a concrete-representation, each of which is different from that of any PL/I character.

Any tree that satisfies the restriction that all of its nodes which contain no components have a type which possesses a concrete-representation, may also be said to possess a *concrete-representation*. This representation is just the character string formed by concatenating, in precedence order, the concrete-representations of these nodes.

1.6.1.4 Tree Notation

1.6.1.4.1 Enumerated-Trees

A particular tree may be completely specified by stating its root-node and describing each subnode in terms of immediate components down to the terminal nodes.

This may be expressed more concisely as an *enumerated-tree* in a notation which specifies:

1. a node type, for the root-node, optionally followed by a comma and a name by which this node can be designated (see Section 1.6.1.3),

optionally followed by
2. a colon, the immediate components of the root-node (which may themselves be enumerated-trees, or may be names designating other trees which are to be copied) which may be enclosed in brackets (“[” and “]”) denoting a component that is to be omitted if and only if its value is <absent>,

and a semicolon.

For example, the tree which consists of a <named-constant> which immediately contains <file> and a <file-description-set>, the latter of which immediately contains <stream> and <input>, can be written as: <named-constant>,nc: <file><file-description-set>: <stream><input>;. Indentation is often used as a visual aid, e.g.

```

<named-constant>,nc:
  <file>
  <file-description-set>:
    <stream>
    <input>;;

```

Semicolons at the end of an enumerated-tree specification may be omitted immediately before a period.

Use of an enumerated-tree in the metalanguage indicates the creation of a local-tree (see Section 1.6.3.1) having the structure and node types indicated, with appropriate copies inserted of the trees to be copied (see (2) above), and with the designators corresponding to the names optionally used as in (1) above set to designate the nodes there created.

A frequently used abbreviation for a particular class of enumerated-trees is to enclose a potential PL/I concrete-representation in double quotes. This is an abbreviation for that tree rooted in $\langle \text{symbol-list} \rangle$ which has this enclosed string as its concrete-representation.

1.6.1.4.2 Forms

Patterns to be searched for in trees may be indicated in the metalanguage by a notation which is the same as that for enumerated-trees, except that the names of trees to be copied may not be included.

Use of such a notation in the metalanguage is always preceded by the word *form*. Its use indicates that a search for, or test of conformance to, the pattern is to be made, yielding true or false, and that the designators corresponding to the names used as in (1) of Section 1.6.1.4.1 are to be set to designate the nodes corresponding to them if and only if the search, or test, returns true. Use of brackets in a form indicates the bracketed components may be either present or absent.

1.6.1.5 Tree Copies

A *copy* of a given tree is constructed as follows:

1. Construct a tree which is equal to the given tree.
2. For each designator node X in the given tree which designates a node Y also in the given tree, change the constructed node which corresponds to X so that it designates the constructed node which corresponds to Y.

1.6.2 Production Rules

The trees actually employed in this document are a limited subset of all the possible trees that could be formed according to the definitions given in Section 1.6.1. Production rules serve as the declarative portion of the metalanguage and do so by specifying restrictions on the forms assumed by the trees used in the definitional process of this document.

1.6.2.1 Production Rules and Syntaxes

A *production-rule* is written with an optional label formed of capital letters and digits, and consists of a category-name, followed by the quoted symbol “::=” and then followed by either a syntactic-expression (see Section 1.6.2.3) or, in a few instances, an English language phrase. Such a production-rule is termed a *defining production-rule* for that category-name written before the “::=”. Within this document, there is at most one defining production-rule for any given category-name.

The basic function of a production-rule is to define a set of possibilities for the number, type(s), and order of the immediate subnodes of a node whose type is the defined category-name. This is done by applying the

definition of a *subnode-type sequence with respect to a syntactic-expression*, given in Section 1.6.2.4.2, to the syntactic-expression of the defining production-rule.

Production-rules are augmented in their function of specifying immediate subnodes by a notational convention used for creating lists of repetitive immediate subnodes. This convention applies to the bracketed category-names whose denotation, exclusive of the brackets, terminates in the quoted symbols “-list” or “-commalist”, and it substitutes for the explicit appearance of a defining production-rule for such category-names (i.e. such category-names have no defining production-rule written in this document).

A *syntax* is any set of production-rules. For example, the set of all of the production-rules in this document is a syntax. Four (disjoint) subsets of the production-rules in this document have enough logical coherence that they have been given names, i.e. the High-level Concrete Syntax, the Low-level Concrete Syntax, the Abstract Syntax, and the Machine-state Syntax. In order to distinguish these syntaxes, the production-rules comprising them have been given numbered labels starting with the quoted symbols “CH”, “CL”, “A”, and “M” respectively. The unlabeled production-rules of this document do not belong to any of these syntaxes.

If a defining production-rule for a category-name occurs in a syntax, then that category-name is said to be *non-terminal with respect to* that syntax. Any category-name that occurs somewhere within the syntactic-expressions of the production-rules of the syntax, but has no defining production-rule in the syntax, is said to be:

1. *non-terminal with respect to* that syntax if its denotation exclusive of the brackets, ends with “-list”, “-commalist”, or “-designator”, and
2. *terminal with respect to* that syntax otherwise.

A category-name that is non-terminal with respect to the syntax composed of all the production-rules occurring in this document is said to be just *non-terminal*; similarly for *terminal*.

The Abstract Syntax additionally allows constraints to be specified for certain category-names. The constraint, written in parentheses after the relevant category-name, is applied by the Translator, or during the interpretation phase, but has no effect on the constitution of a tree specified by the syntax.

1.6.2.2 Complete and Partial Trees

Given a syntax and a category-name, a *complete tree with respect to the given syntax and category-name* is any tree which can be obtained by

1. starting from a node whose type is the given category-name;
2. marking that node “incomplete”;
3. choosing a node marked “incomplete” and completing it as follows:
 - a. attaching immediate subnodes to it in the number, type(s), and order specified by applying the definition of a *subnode-type sequence with respect to a syntax and category-name*, given in Section 1.6.2.4.1, to the given syntax and that category-name which is the type of the chosen node;
 - b. marking as “incomplete” each attached immediate subnode that has a type which is a category-name; and
 - c. removing the “incomplete” mark from the chosen node;
4. repeating (3) as long as some node marked “incomplete” remains in the tree being constructed.

A complete tree with respect to the syntax composed of all the production-rules occurring in this document is said to be just a *complete tree*.

A *partial tree* is any tree which is not a complete tree but which can be obtained by deleting some nodes from some complete tree.

The production-rule

$$\langle \text{bound-pair} \rangle ::= \langle \text{lower-bound} \rangle \langle \text{upper-bound} \rangle \mid \langle \text{asterisk} \rangle$$

defines two possibilities, which may be written as (a) or drawn as (b)

(a) <bound-pair>: <bound-pair>:
 <lower-bound> or <asterisk>;
 <upper-bound>;

(b)

```
graph TD
    A["<bound-pair>"] --> B["<lower-bound>"]
    A --> C["<upper-bound>"]
    D["<bound-pair>"] --> E["<asterisk>"]
```

or

A node whose type is <identifier-list> may have any non-zero number of immediate subnodes of type <identifier>, i.e.

<identifier-list>: <identifier>;	or	<identifier-list>: <identifier> <identifier>;	or	<identifier-list>: <identifier> <identifier> <identifier>;
-------------------------------------	----	---	----	---

and so on.

A node whose type is `<subscript-commalist>` may have any non-zero number of `<subscript>` immediate subnodes, but with nodes whose type is the PL/I character `<, >` interspersed between adjacent ones, i.e.

$$\begin{array}{lll} \langle \text{subscript-commalist} \rangle: & \text{or} & \langle \text{subscript-commalist} \rangle: & \text{or} & \langle \text{subscript-commalist} \rangle: \\ \langle \text{subscript} \rangle; & & \langle \text{subscript} \rangle & & \langle \text{subscript} \rangle \\ & & \langle , \rangle & & \langle , \rangle \\ & & \langle \text{subscript} \rangle; & & \langle \text{subscript} \rangle \\ & & & & \langle , \rangle \\ & & & & \langle \text{subscript} \rangle; \end{array}$$

and so on.

The production-rule

$$\langle \text{repeat-option} \rangle ::= \langle \text{expression} \rangle (\text{scalar})$$

specifies that only <expression>s yielding scalar values (i.e. not aggregate values) are valid immediate subnodes of a <repeat-option>.

Figure 3. Examples of Syntax.

The trees utilized by the definition process of this document are only complete trees or partial trees. Other possible forms of trees are never utilized. Furthermore, it is the usual case that complete trees are utilized, or at least utilized at the interfaces between the various operations of the definition. Although there are a few

specific exceptions, it is a general rule that partial trees occur only in a very local context in the process of building up a complete tree. Several of the “instructions” (see Section 1.6.3.4) of the metalanguage are in fact designed to assist in the process of building complete trees.

Most of the complete trees used in this document are constructed explicitly by particular algorithms, or implicitly by displaying an enumerated tree. In a few cases (notably the PARSE, HIGH-LEVEL-PARSE, and LOW-LEVEL-PARSE operations of Chapter 4), only the existence of complete trees satisfying certain conditions is required, and no algorithm for constructing them is even mentioned.

1.6.2.3 Syntactic-expressions and Syntactic-units

Given a syntax, a *syntactic-expression* is defined to be either a single syntactic-unit, or several syntactic-units any of the adjacent pairs of which is possibly separated by a “|” or a “•”. The symbols are called the or-symbol and the bullet respectively.

Given a syntax, a *syntactic-unit* is defined to be one of the following:

- a single category-name,
- a syntactic-expression enclosed in the brackets “[” and “]”, or
- a syntactic-expression enclosed in the braces “{” and “}”.

1.6.2.4 Use of the Production Rules

In any complete tree with respect to a syntax *s* and a category-name, the immediate subnodes of each node *n* whose type is a category-name *c* must satisfy certain requirements. The requirements dictate the number, type(s) and order of the immediate subnodes. The requirements are determined by *c* and the production-rules of *s*, augmented by a few notational conventions. To obtain those requirements, the production-rules and notational conventions are interpreted according to the definitions in Sections 1.6.2.4.1 and 1.6.2.4.2. In some cases they uniquely prescribe the number, type(s) and order of the immediate subnodes of *n*, as illustrated by the fourth example shown in Figure 3 on page 27. In other cases, as illustrated by the first three examples shown there, they present more than one choice (possibly infinitely many choices) for the number, type(s) and order of the immediate subnodes of *n*.

In any complete tree, each node whose type is an object other than a category-name is constrained to have no immediate subnodes. This constraint follows from the definition of a complete tree.

During the construction (by any algorithm) of a complete tree with respect to a syntax *s* and a category-name, the following two questions arise repeatedly: Given, in the tree being constructed, a node *n* whose type is a category-name *c*, how many immediate subnodes may *n* have, and what may their types be? To answer these questions, one applies the definition in Section 1.6.2.4.1 to *s* and *c* to obtain one or more (possibly empty) sequences of node types, each termed a *subnode-type sequence with respect to the given syntax and category-name*. If *nt[1],...,nt[nsn]*, *nsn* ≥ 0, is any such sequence of node types, then *n* may have *nsn* immediate subnodes of types *nt[1],...,nt[nsn]*, in order.

The first three cases of the definition in Section 1.6.2.4.1 deal with the notational conventions augmenting the production-rules. Case 4.1 of the definition, which deals with production-rules containing a syntactic-expression, appeals to a second definition (shown in Section 1.6.2.4.2). In this case, the answers to the questions posed above are provided indirectly by the second definition. Thus, given a node *n* whose type is a category-name defined by a production-rule containing a syntactic-expression *se*, one applies the definition of Section 1.6.2.4.2 to *se* to obtain one or more (possibly empty) sequences of node types, each termed a *subnode-type sequence with respect to the given syntactic-expression*. If *nt[1],...,nt[nsn]*, *nsn* ≥ 0, is any such sequence of node types, then *n* may have *nsn* immediate subnodes of types *nt[1],...,nt[nsn]*, in order.

1.6.2.4.1 Subnode-type Sequence with respect to a Syntax and Category-name

A subnode-type sequence with respect to a syntax *s* and a category-name *c* is defined as follows:

Case 1. The denotation of *c*, excluding any terminating bracket, ends with “-list”.

A subnode-type sequence with respect to *s* and *c* consists of a sequence of any non-zero number of instances of the category-name whose denotation is obtained by deleting the “-list” from the denotation of *c*.

Case 2. The denotation of *c*, excluding any terminating bracket, ends with “-commalist”.

A subnode-type sequence with respect to *s* and *c* consists of a sequence which:

1. contains any non-zero number, *k*, of instances of the category-name whose denotation is obtained by deleting the “-commalist” from the denotation of *c*, and which
2. contains *k*–1 instances of the category-name $\langle, \rangle$ and which
3. is arranged so that no two instances of the same category-name are adjacent.

Case 3. The denotation of *c*, excluding any terminating bracket, ends with “-designator”.

Let *cdn* be the category-name whose denotation is obtained by deleting the “-designator” from the denotation of *c*. Let *dn* be any existing node whose type is *cdn*. Let *u* be the unique-name of *dn*. Then, a subnode-type sequence with respect to *s* and *c* is a sequence consisting of a single instance of *u*.

Case 4. *c* is defined by a production-rule in *s*.

Case 4.1. The defining production-rule contains a syntactic-expression *se* written following the “::=”.

A subnode-type sequence with respect to *s* and *c* is a subnode-type sequence with respect to *se* (see Section 1.6.2.4.2).

Case 4.2. (Otherwise).

A subnode-type sequence with respect to *s* and *c* is implicit in the English language remark written following the “::=” in the defining production-rule (and may be IMPLEMENTATION-DEFINED).

Case 5. (Otherwise).

c is necessarily terminal with respect to *s*. A subnode-type sequence with respect to *s* and *c* is the empty sequence.

1.6.2.4.2 Subnode-type Sequence with respect to a Syntactic-expression

A subnode-type sequence with respect to a syntactic-expression *se* is defined as follows:

Case 1. *se* is a syntactic-unit.

Case 1.1. The syntactic-unit is a single category-name.

A subnode-type sequence with respect to *se* is a sequence consisting of a single instance of that category-name.

Case 1.2. The syntactic-unit is a syntactic-expression *ise* enclosed in the brackets “[” and “]”.

A subnode-type sequence with respect to *se* is either a subnode-type sequence with respect to *ise* or the empty sequence.

Case 1.3. The syntactic-unit is a syntactic-expression *ise* enclosed in the brackets “{” and “}”.

A subnode-type sequence with respect to *se* is a subnode-type sequence with respect to *ise*.

Case 2. *se* is a sequence of two or more syntactic-units possibly separated by a “|” or a “•”.

Case 2.1. An or-symbol occurs as at least one such separator.

Consider all or-symbols occurring thus in *se* to partition it into a sequence *ise*[1],...,*ise*[*k*], $k \geq 2$, of inner syntactic-expressions. A subnode-type sequence with respect to *se* is a subnode-type sequence with respect to any one of the *ise*[*i*], $i = 1, \dots, k$, chosen arbitrarily.

Case 2.2. A bullet occurs as a separator and an or-symbol does not.

Consider all bullets occurring thus in *se* to partition it into a sequence *ise*[1],...,*ise*[*k*], $k \geq 2$, of inner syntactic-expressions. A subnode-type sequence with respect to *se* is a subnode-type sequence with respect to a syntactic-expression obtained from *se* by rearranging all the *ise*[*i*], $i = 1, \dots, k$, in an arbitrary order and omitting these bullets.

Case 2.3. (Otherwise).

se is necessarily a sequence of syntactic-units optionally separated by blanks. Consider each such syntactic-unit as a syntactic-expression, thereby obtaining from *se* a sequence *ise*[1],...,*ise*[*k*], $k \geq 2$, of inner syntactic-expressions. A subnode-type sequence with respect to *se* is the concatenation, in order, of subnode-type sequences with respect to each of the *ise*[*i*], $i = 1, \dots, k$. (The sequence formed by concatenating sequences consists of the elements of the first sequence, in order, followed by the elements of the second sequence, in order, and so on.)

1.6.3 Operations

The procedural part of the metalanguage provides for the writing of algorithms termed *operations*. These are expressed in a semi-formal programming language which uses the grammatical flexibility of ordinary English prose, while at the same time attaching precise meaning to certain words and phrases, in order that the flow of control and the tree manipulations in the operations be well defined. Completely formal notation is used to describe trees in accordance with their syntactic definitions.

1.6.3.1 Nature of an Operation

An operation, applied to zero or more *operands*, may be performed by the processor (see Section 1.6.4), with the effect of:

1. changing the $\langle \text{machine-state} \rangle$, or
2. changing an operand, or
3. returning a *result*, or
4. any combination of (1), (2), and (3).

Within an operation either *operand-names* or *local-variable-names* may be used for accomplishing this effect. These names serve as designators of trees, which may be either portions of the $\langle \text{machine-state} \rangle$, or *local-trees* created within this or another operation.

When an operation is performed, its operand-names are set to designate the operand trees it has been passed. It then has the following data available to it:

1. The whole $\langle \text{machine-state} \rangle$, which is directly accessible for inspection or change at any time.
2. The operands which it has been passed, which can be inspected or changed.

3. Local-trees local to itself, which it can freely construct, inspect, or modify. These trees are deleted when the operation terminates.

The operation may also apply any operation defined in this document (including itself) to any operands which it may select from among the trees available to it.

Upon completion of its actions, an operation may return a result, which then becomes available to whichever operation applied the given operation to its operands. As with operands, this result may be selected from among the trees available to the operation. In the case that (a portion of) a local-tree local to this operation is selected, this tree is not deleted, but is copied to become a local-tree local to the applying operation: (See Section 1.6.3.4 on “perform”.)

1.6.3.2 Nondeterministic Operations

The phrases “optionally” and “in any order” are used in some operation descriptions. They indicate that the processor is to make a choice each time that part of the operation is executed. In general, then, the conceptual PL/I machine defines a set of possible interpretations for a program.

1.6.3.3 Format of an Operation

The written description of an operation has a format consisting of a heading and a body.

The heading may have three parts:

1. There is always a specification of the form “Operation:” followed by the underlined name of the operation, optionally followed by a parenthesized list of the names used to refer to the operands passed to the operation, the names being separated by commas.
2. For an operation which has operands, the word “where” then precedes a description of the type(s) of tree to which each operand name may refer. Brackets around a node type indicate that the operand is optional, which is an abbreviation for stating that it may alternatively have the value <absent>.
3. For an operation which returns a result, a final part of the heading is of the form “result:” followed by a description of the type(s) of tree which may be returned.

Operation: EVALUATE-IN-OPTION(al, vr)

Where: al is an <allocation>,
 vr is a <variable-reference>.

Result: a <generation> or <fail>.

Figure 4. An Example of an Operation Returning a Result.

The body of an operation consists of either a Step-list or a Case-set. Each Step or Case is a numbered section containing written instruction descriptions of arbitrary complexity, and may itself contain a Step-list or Case-set, numbered with an additional index position and indented to indicate this containment.

Steps are normally performed sequentially.

Each *Case* consists of a condition part followed by an executable part. Within each case-set, the set of condition parts is such that, at any execution point, exactly one of the condition parts is satisfied. (The

conventional condition part “(Otherwise)” is satisfied whenever none of the other condition parts of the case-set is satisfied.) The execution of a case-set consists of executing the executable part of the one case whose condition part is then satisfied.

The normal sequence for Steps and Cases may be modified by explicit instructions using such terms as “go to” or “perform” which are defined in Section 1.6.3.4. The terminology for selecting some actions, either in a defined left-to-right order or in an unspecified order, is “in left-to-right order” and “in any order” respectively. When selection is between two options, “in either order” may be used instead of “in any order”. Optional selection as to whether an action will be carried out is indicated by words such as “optionally perform”.

Operation: EVERY-BIF(rdd,x)

Step 1. Perform EVALUATE-EXPRESSION(x) to obtain an $\langle\text{aggregate-value}\rangle$, u.

Step 2. In any order, convert each scalar-element of u to $\langle\text{bit}\rangle$, to obtain v.

Step 3.

Case 3.1. Every scalar-element of v contains $\langle\text{null-bit-string}\rangle$.

Let r be $\langle\text{zero-bit}\rangle$.

Case 3.2. At least one scalar-element of v which does not contain $\langle\text{null-bit-string}\rangle$ has a $\langle\text{bit-string-value}\rangle$ with at least one $\langle\text{bit-value}\rangle$ containing $\langle\text{zero-bit}\rangle$.

Let r be $\langle\text{zero-bit}\rangle$.

Case 3.3. (Otherwise).

Let r be $\langle\text{one-bit}\rangle$.

Step 4. Return an $\langle\text{aggregate-value}\rangle$ containing a $\langle\text{bit-string-value}\rangle$ containing r.

Figure 5. An Example of an Operation Containing a Step-list and a Case-set.

1.6.3.4 Instructions

An *instruction* is a specification of some action involving the creation, destruction, inspection, or modification of some tree(s), or causing some departure to be made from the normal sequential flow of control through an operation.

Operand-names and local-variable-names strictly denote designators of trees rather than tree values themselves. However, apart from their use in a context where they acquire designator values, are passed as operands, or are returned as results, references to these names are always taken to be an abbreviation for references to the tree designated by the strict value.

The “let” instruction is used to indicate that the named variable is henceforth to reference the specified tree, which may be an existing tree or one newly created, e.g. by use of the enumerated-tree notation or by copying an existing tree. Use of a name for a tree following some description of the root-node and a comma, e.g. in the enumerated-tree notation, is equivalent to use of a “let” instruction. No change to any tree previously designated by the named variable or operand occurs as a result of the “let” instruction.