

Informationsverarbeitung
Programmiersprache PL/I

DIN
66 255

Information processing; programming language PL/I

Zusammenhang mit der von der International Organization for Standardization (ISO) herausgegebenen Internationalen Norm ISO 6160-1979, siehe Abschnitt 3.

Inhalt

	Seite
1 Anwendungsbereich und Zweck	1
2 Allgemeines zur Programmiersprache PL/I	1
3 Umfang der Norm	1
4 Englisch-deutsche Fachwörterliste	2
5 Programmiersprache PL/I	6

1 Anwendungsbereich und Zweck

Die Norm dient dazu, die Programmiersprache PL/I so festzulegen, daß ein weltweiter Austausch von Programmen, die in dieser Sprache abgefaßt sind, zwischen Datenverarbeitungsanlagen ermöglicht wird. Die Norm stellt damit die Grundlage für das Herstellen von Übersetzern und das Ausarbeiten von Anleitungen (Handbücher, Lehrbücher) zur Programmierung in PL/I dar. Die in Abschnitt 4 gegebene englisch-deutsche PL/I-Fachwörterliste soll der Vereinheitlichung im Gebrauch der bei PL/I verwendeten Fachwörter dienen.

Für die Begriffe und die Benennungen dieser Norm wird nur Geltung in bezug auf die hier genormte Programmiersprache, jedoch keine allgemeine (übergreifende) Geltung beansprucht. Entsprechend sind die deutschen Benennungen in der zweisprachigen Wörterliste aufzufassen. Die allgemein verbindlichen Begriffe und Benennungen der Datenverarbeitung sind in Begriffsnormen festgelegt.

2 Allgemeines zur Programmiersprache PL/I

PL/I (Programming Language One) ist eine problemorientierte Programmiersprache für betriebswirtschaftliche, verwaltungstechnische, kaufmännische und technisch-wissenschaftliche Programme und System-Programme.

Die Entwicklung dieser Sprache begann 1963 und führte 1964 zu einer ersten Darstellung. Die neue Programmiersprache erhielt 1965 den Namen PL/I.

Seit 1966 arbeiteten der ECMA-Ausschuß ¹⁾ TC 10 und der ANSI-Ausschuß ²⁾ X3J1 in einem gemeinsamen Projekt an der Normung von PL/I. 1976 wurde die der ISO-Norm zugrunde liegende Norm ANSI X3.53-1976 verabschiedet.

3 Umfang der Norm

Mit dieser Norm wird die Internationale Norm ISO 6160 „Programming Language PL/I“ (1. Ausgabe 1. Juli 1979) in das deutsche Normenwerk übernommen.

Der englische Text umfaßt in neun Kapiteln die Beschreibung der Programmiersprache PL/I. In ihm wird sowohl die Form als auch die Ausführung von in PL/I geschriebenen Programmen festgelegt. Die Sprache wird definiert, indem eine gedachte PL/I-Maschine beschrieben wird, die PL/I-Quellprogramme übersetzt und interpretiert. Das Quellprogramm muß den Regeln genügen, die im Kapitel „Concrete Syntax“ beschrieben sind. Es wird in eine interne Normalform überführt, die im Kapitel „Abstract Syntax“ festgelegt ist. Die Regeln für die Übersetzung enthält das Kapitel „Translator“. Das interne Programm wird von der PL/I-Maschine benutzt, um den Ablauf der Ausführung zu bestimmen. Die Regeln für die Ausführung sind im Kapitel 5 „PL/I-Interpreter“ und in den folgenden dargestellt.

Es ist nicht möglich, einen deutschen Text zu geben, für den eine völlige inhaltliche Übereinstimmung mit dem ISO-Entwurf garantiert werden kann. Wegen des hohen Abstraktionsgrades des Gegenstandes können schon geringfügige Änderungen des Textes beträchtliche Bedeutungsverschiebungen verursachen. Der Zweck der Norm wäre aber verfehlt, wenn sie nicht vollständig mit der ISO-Norm über PL/I übereinstimmen würde.

Daher wird in dieser Norm der englische Originaltext der ISO-Norm wiedergegeben.

1) ECMA European Computer Manufacturers Association
 2) ANSI American National Standards Institute

Fortsetzung Seite 2 bis 211

Normenausschuß Informationsverarbeitung (NI) im DIN Deutsches Institut für Normung e.V.

Nachdruck, auch auszugsweise, nur mit Genehmigung des DIN Deutsches Institut für Normung e. V., Berlin, gestattet.

4 Englisch-deutsche Fachwörterliste

Die Fachwörterliste enthält die Übersetzungen der englischen Fachwörter, die in der ISO-Norm von PL/I benutzt werden. Die Schlüsselwörter sind jedoch nicht übersetzt. Die in Klammern gesetzten Teile von Benennungen dürfen weggelassen werden, wenn keine Mißverständnisse möglich sind.

activate	aktivieren	close, to	abschließen
add builtin function	Addierfunktion	collate builtin function	Sortierfolgefunktion
addr builtin function	Adreßfunktion	colon	Doppelpunkt
after builtin function	Folgenendefunktion	column	Spalte
aggregate	Aggregat	comma	Komma
aligned	ausgerichtet	comment	Kommentar
alignment	Ausrichtung	compatible	verträglich
allocate, to	Speicher zuordnen	complex	komplex
allocation builtin function	Zuordnungsanzahlfunktion	complex builtin function	Komplexfunktion
and symbol	Konjunktionssymbol, Undsymbol	computational	Rechen...
area	Gebiet	concatenation	Verkettung
argument	Argument	condition	Bedingung
arithmetic	arithmetisch	condition name	Bedingungsname
array	Matrix	conjug builtin function	Konjugiertfunktion
assignment	Zuweisung	connected	zusammenhängend
associated data type	assoziiierter Datentyp	constant	Konstante
asterisk	Stern	contextual	kontextuell
attribute	Attribut	control	(Ablauf)steuerung
automatic storage	automatischer Speicher	control format	Steuerformat
base	Basis	controlled group	Zählschleife
based	zeigerbezogen	controlled storage	verwalteter Speicher
before builtin function	Folgenanfangsfunktion	conversion	Konversion
begin block	Beginnblock	copy builtin function	Kopierfunktion
begin statement	Beginnanweisung	copy option	Kopierangabe
binary	dual	credit sign	Kreditzeichen
binary builtin function	Dualfunktion	current	aktuell
bit	Bit		
bit builtin function	Bitfunktion	data	Daten
blank	Leerzeichen	data directed	datengesteuert
block	Block	dataset	(physische) Datei
bool builtin function	Boolefunktion	date builtin function	Datumfunktion
bound	Grenze	debit sign	Debetzeichen
break character	Unterstreichungs(szeichen)	decat builtin function	Zerlegungsfunktion
builtin function	Standardfunktion	decimal	dezimal
by name	gleichnamig	decimal builtin function	Dezimalfunktion
by option	Schrittweitenangabe	declaration	Vereinbarung
		default	vorgegeben, Vorgabe-
		defined	überlagernd
call statement	Prozeduraufruf	delete statement	Löschanweisung
ceil builtin function	Deckefunktion	delimit	Begrenzer
character	Zeichen	derived	abgeleitet
character builtin function	Zeichenfunktion	description	Beschreibung
character set	Zeichenvorrat	descriptor	Deskriptor
character string	Zeichenfolge	digit	Ziffer

AVA Documents
Document Sharing Hub

literal	literal	pageno pseudo-variable	Seitennummer-Pseudovariablen
locate statement	Lokalisierungsangabe	pagesize option	Seitenlängenangabe
low builtin function	Niedrigstwertfunktion	parameter	Parameter
lower bound	untere Grenze	perform, to	durchführen
		period	Punkt
match, to	übereinstimmen	picture	Masken
max builtin function	Maximumfunktion	picture scalefactor	Maskenskallierfaktor
maximum length	Maximallänge	pictured	maskiert
member	(Struktur)glied	pointer	(Absolut)zeiger
min builtin function	Minimumfunktion	pointer builtin function	Zeigerfunktion
mode	(Zahlen)modus	pointer set option	Zeigersetzangabe
multiple closure	Mehrfachabschluß	position	Stelle
multiply builtin function	Multiplizierfunktion	power operator	Exponentiationsoperator
multiply symbol	Multiplikationszeichen	precision	Genauigkeit
		precision builtin function	Genauigkeitsfunktion
named	benannt	prefix	Präfix
non-computational	Programmsteuer. . .	prefix operator	Präfixoperator
non-iterative group	einfacher Verbund	primary entry point	Haupteingang
normal sequence	normale Fortsetzung	primitive expression	Primitivausdruck
not symbol	Negationszeichen	procedure	Prozedur
null	Leer. . .	procedure body	Prozedurumpf
null builtin function	Nullzeigerfunktion	procedure function reference	Aufruf einer Funktionsprozedur
number	Zahl	procedure statement	Prozeduranweisung
number of digits	Ziffernanzahl	processor	Prozessor (der PL/I-Maschine)
numeric picture	Zahlenmaske	prod builtin function	Produktfunktion
numeric string	numerische Zeichenfolge	program	Programm
		programmer named condition	benannte Bedingung
offset	Relativzeiger	promotable	überführbar
offset builtin function	Relativzeigerfunktion	pseudo-variable	Pseudovariablen
on statement	Vorsorgeanweisung	put statement	Ablegeanweisung
on system statement	System-Vorsorgeanweisung		
on unit	Vorsorgeeinheit	qualification	Kennzeichnung
onchar builtin function	Zeichenfehlerfunktion	qualifier	Kennzeichner
onchar pseudo-variable	Zeichenfehler-Pseudovariablen		
oncode builtin function	Fehlercodefunktion	radix factor	(Zahlendarstellungs)basis
onfield builtin function	Namensfehlerfunktion	raise, to	auslösen
onfile builtin function	Dateifehlerfunktion	range specification	Bereichsangabe
onkey builtin function	Schlüsselfehlerfunktion	read statement	Leseanweisung
onsource builtin function	Fehlerquellefunktion	real	reell
onsource pseudo-variable	Fehlerquelle-Pseudovariablen	real builtin function	Realteilfunktion
open statement	(Datei)Eröffnungsanweisung	real pseudo-variable	Realteil-Pseudovariablen
open, to	eröffnen	record	(Daten)satz
option	(Ergänzungs)angabe	recursive	rekursiv
or symbol	Disjunktionssymbol, Odersymbol	refer option	Bezugsangabe
output	Ausgabe	reference	Bezugnahme
overflow condition	(Gleitpunkt)Überlaufbedingung	remote format	Verweisformat
		reordering	Umordnung
page option	Seitenvorschubangabe	repeat option	Wiederholangabe
pageno builtin function	Seitennummerfunktion	repetition factor	Wiederholfaktor

replicated string constant result return statement returns descriptor reverse builtin function revert statement rewrite statement right parenthesis round builtin function	Folgenkonstante mit Wiederholung Ergebnis Rückkehranweisung Ergebnisdeskriptor Folgenumkehrfunktion Rücknahmeanweisung Rückschreibanweisung (runde) Klammer zu Rundungsfunktion	stringsize condition structur subroutine reference subscript subscriptrange condition substr builtin function substr pseudo-variable subtract builtin function sum builtin function	Folgenüberlaufbedingung Struktur Unterprogrammaufruf Index Indexgrenzen-Bedingung Teilfolgefunktion Teilfolge-Pseudovariablen Subtrahierfunktion Summierungsfunktion
scalar scale scale factor scale type scope secondary entry point semicolon sentence set option signal statement signed integer simple single quote single statement size condition slash some builtin function special character specification specifier sqrt builtin function state statement statement name static storage stop statement storage storage control statement stream string string builtin function string or picture symbol string overlay defined string pseudo-variable stringrange condition	Skalar Skalierung Skalierfaktor Skaliertyp Gültigkeitsbereich Zusatzeingang Semikolon (Programm)satz Setzangabe Bedingungsaufruf ganze Zahl mit Vorzeichen einfach Apostroph Einzelanweisung Stellenüberlauf Schrägstrich Disjunktionfunktion Sonderzeichen Spezifikation Spezifikationsselement Quadratwurzelfunktion Zustand Anweisung Anweisungsname statischer Speicher Stoppanweisung Speicher Speicherverwaltung (Daten)strom Folge Folgefunktion Folgen- oder -Masken-Symbol folgenüberlagernd Folgen-Pseudovariablen Folgenreihenbedingung	tab target target reference terminate, to time builtin function title option to option translate builtin function transmission true trunc builtin function type unaligned unconditional underflow condition unique unit unspec builtin function unspec pseudo-variable unscripted reference upper bound valid builtin function value variable varying verify builtin function while option word write statement zero suppression zerodivide condition	Tabulator Ziel (Zuweisungs)zielbezugsnahme beenden Tageszeitfunktion Titelangabe Endwertangabe (Zeichen)umsetzungsfunktion Übertragung wahr Abschneidfunktion Type nichtausgerichtet unbedingt (Gleitpunkt)unterschreitungs-Bedingung eindeutig Einheit Interdarstellungsfunktion Interdarstellungs-Pseudovariablen nichtindizierte Bezugsnahme obere Grenze Gültigkeitsfunktion Wert Variable varierend Prüffunktion Solange-Angabe Wort Schreibenanweisung Nullenunterdrückung Nulldivisionsbedingung

5 Programmiersprache PL/I

Aus dem hier folgenden Originaltext sind der Index, die Tabellen und die Beispiele nicht Bestandteil der Norm.

Weitere Normen

DIN 44 300	Informationsverarbeitung; Begriffe
DIN 66 026	Informationsverarbeitung; Programmiersprache ALGOL
DIN 66 027	Informationsverarbeitung; Programmiersprache FORTRAN
DIN 66 028	Informationsverarbeitung; Programmiersprache COBOL
DIN 66 253 Teil 1	(z. Z. noch Entwurf) Informationsverarbeitung; Programmiersprache PEARL, Basic PEARL

CHAPTER 1: SCOPE AND OVERVIEW	1
1.0 Scope	1
1.1 An Informal Guide to the PL/I Definition	1
1.1.1 A Summary of PL/I	1
1.1.2 The Form of the Definition	2
1.1.3 Summary of Chapter Structure	3
1.1.4 Introduction to the Metalanguage	7
1.1.4.1 Tree Concepts	7
1.1.4.2 Syntaxes	8
1.1.4.3 Algorithm Concepts	9
1.1.4.3.1 Relationships between an Implementation and This Definition	11
1.2 Flexibilities of Interpretation	11
1.2.1 Rejection of Programs	11
1.2.1.1 Quantitative Restrictions	12
1.2.1.2 Operating Environment	12
1.2.1.4 Expression Evaluation	12
1.2.1.5 Interrupts and Assignment	13
1.2.1.6 Input/Output	14
1.2.1.7 On-units	14
1.2.2 Implementation-defined features	16
1.3 The Metalanguage	16
1.3.1 Trees	17
1.3.1.1 Tree Definitions	17
1.3.1.2 Node Objects	18
1.3.1.2.1 Unique-names	18
1.3.1.2.2 Types	19
1.3.1.3 Node Notation	19
1.3.1.4 Tree Notation	20
1.3.1.4.1 Enumerated Trees	20
1.3.1.4.2 Forms	21
1.3.1.5 Tree Copies	21
1.3.2 Production Rules	21
1.3.2.1 Production Rules and Syntaxes	22
1.3.2.2 Complete and Partial Trees	23
1.3.2.3 Syntactic-expressions and Syntactic-units	23
1.3.2.4 Application of the Production Rules	24
1.3.3 Operations	24
1.3.3.1 Nature of an Operation	25
1.3.3.2 Nondeterministic Operations	25
1.3.3.3 Format of an Operation	25
1.3.3.4 Instructions	27
1.3.3.5 Convert	28
1.3.3.6 Additional Notational Conventions	28
1.3.3.7 Arithmetic	28
1.3.4 The Processor	29
1.3.5 Mechanization of the Metalanguage	29
1.4 Initialization of the Machine-state	31
1.4.1 The Machine-state	31
1.4.2 Initialization	31
1.4.3 The Top-level Operations	31
1.4.3.1 Define-program	32
1.4.3.2 Translation-phase	32
1.4.3.3 Interpretation-phase	32
CHAPTER 2: CONCRETE SYNTAX	33
2.0 Introduction	33
2.1 The Intent of this Definition	33
2.1.1 Concrete and Abstract Syntaxes	33
2.2 Organization of the Concrete Syntax	33
2.3 The High-level Syntax of PL/I	33
2.3.1 Procedure	33
2.3.2 Unit	33
Contents	1

2.3.3	Executable Units	34
2.4	The Middle-level Syntax of PL/I	35
2.4.1	Sentence	35
2.4.2	Statement	36
2.4.3	Prefixes	36
2.4.3.1	Condition Prefixes	36
2.4.3.2	Statement Name Prefixes	36
2.4.4	Data Declaration	37
2.4.4.1	Dimension Attribute and Dimension Suffix	37
2.4.4.2	Attributes	37
2.4.4.3	Data Attributes	38
2.4.4.4	Environment and Options	38
2.4.4.5	Generic	39
2.4.4.6	Initial	39
2.4.4.7	The Default Statement	39
2.4.5	The Procedure Statement	39
2.4.6	The Entry Statement	40
2.4.7	The Begin Statement	40
2.4.8	The Do Statement	40
2.4.9	The End Statement	40
2.4.10	Flow of Control Statements	40
2.4.10.1	The Call and Return Statements	40
2.4.10.2	The Go To Statement	40
2.4.10.3	The Null Statement	40
2.4.10.4	The Revert and Signal Statements	40
2.4.10.5	The Stop Statement	41
2.4.11	Storage Control Statements	41
2.4.12	Input/Output Statements	41
2.4.12.1	The Open and Close Statements	41
2.4.12.2	Record I/O	42
2.4.12.3	Stream I/O	42
2.4.12.3.1	Stream Input Specification	43
2.4.12.3.2	Stream Output Specification	43
2.4.12.3.3	Format Specification Lists and the Format Statement	44
2.4.13	Expressions	47
2.5	The Low-level Syntax of PL/I	47
2.5.1	PL/I Text	47
2.5.2	Comment	47
2.5.3	Identifier	47
2.5.4	Arithmetic Constant	48
2.5.5	String Constants and Pictures	48
2.5.6	Isub	48
2.5.7	Include	48
2.6	Character Sets	48
2.6.1	Language Character Set	49
2.6.1.1	Letters and Digits	49
2.6.1.2	Special Characters	49
2.6.2	Data Character Set	50
2.7	Abbreviations	51
CHAPTER 3:	ABSTRACT SYNTAX	51
3.0	Introduction	51
3.1	Abstract Syntax Rules	51
3.1.1	Program	51
3.1.2	Procedure	51
3.1.3	Declaration	52
3.1.4	Variable	52
3.1.5	Data-description	53
3.1.6	Data-type	53
3.1.7	Named-constant	54
3.1.8	Entry-or-executable-unit	54
3.1.9	Begin-Block	54
3.1.10	Groups	54
3.1.11	On Statement	55
3.1.12	If Statement	55
3.1.13	Flow of Control Statements	55
3.1.14	Storage Statements	56
3.1.15	I/O Statements	56
3.1.16	Record I/O Statements	56
3.1.17	Stream I/O Statements	56

3.1.18 Expression	58
3.1.19 Types of Reference	59
3.1.20 Constant and Isub	60
3.1.21 Types of Value	60
3.1.22 Types of Picture	60
CHAPTER 4: THE TRANSLATOR	
4.0 Introduction	63
4.1 Translate	63
4.2 Forming the Concrete Procedure	64
4.2.1 Low-level-parse	64
4.2.2 Middle-level-parse	65
4.2.3 High-level-parse	66
4.3 Completion of the Concrete Procedure	68
4.3.1 Reorganize	68
4.3.1.1 Complete-options	69
4.3.1.2 Modify-statement-names	70
4.3.1.3 Complete-attribute-implications	71
4.3.1.4 Defector-declarations	71
4.3.2 Construct-explicit-declarations	72
4.3.2.1 Declare-parameters	72
4.3.2.2 Declare-statement-names	73
4.3.2.3 Construct-statement-name-declarations	75
4.3.3 Complete-structure-declarations	76
4.3.3.1 Determine-structure	76
4.3.3.2 Expand-like-attribute	77
4.3.3.3 Convert-to-logical-levels	77
4.3.3.4 Propagate-alignment	78
4.3.3.5 Find-applicable-declaration	78
4.3.3.6 Find-fully-qualified-name	79
4.3.4 Construct-contextual-declarations	80
4.3.5 Construct-implicit-declarations	82
4.3.6 Complete-declarations	82
4.3.6.1 Test-attribute-consistency	83
4.3.6.2 Test-invalid-duplicates	85
4.3.6.3 Append-system-defaults	86
4.3.6.4 Apply-defaults	87
4.3.6.5 Test-default-applicability	88
4.3.6.6 Copy-descriptors	91
4.3.6.7 Test-offset-in-description	92
4.3.6.8 Test-descriptor-extent-expressions	93
4.3.7 Validate-concrete-declarations	94
4.3.7.1 Check-attribute-completeness-and-delete-attributes	95
4.4 Create-abstract-equivalent-tree	95
4.4.1 Creation of Blocks and Groups	96
4.4.1.1 Create-procedure	96
4.4.1.2 Create-begin-block	97
4.4.1.3 Create-block	97
4.4.1.4 Replace-concrete-designators	98
4.4.1.5 Create-group	98
4.4.1.6 Create-entry-or-executable-unit-list	99
4.4.1.7 Create-executable-unit-list	100
4.4.1.8 Create-executable-unit	101
4.4.1.9 Create-entry-point	101
4.4.1.10 Create-statement-name-list	102
4.4.1.11 Create-condition-prefix-list	103
4.4.1.12 Create-condition	104
4.4.2 Creation of Statements	105
4.4.2.1 Create-assignment-statement	106
4.4.2.2 Create-by-name-assignment	106
4.4.2.3 Data-descriptions Proper for Assignment	107
4.4.2.4 Create-by-name-parts-list	108
4.4.2.5 Find-by-name-parts	109
4.4.2.6 Create-allocation	110
4.4.2.7 Create-format-statement	111
4.4.2.8 Create-format-iteration	112
4.4.2.9 Create-freeing	113
4.4.2.10 Create-if-statement	114
4.4.2.11 Create-balanced-unit	115
4.4.2.12 Create-located-statement	116
4.4.2.13 Create-on-statement	117
4.4.3 Create-declaration	120
4.4.3.1 Create-named-constant	121
4.4.3.2 Create-variable	122
4.4.3.3 Create-bound-pair-list	123
4.4.3.4 Create-data-description	124
4.4.3.5 Create-data-type	125
4.4.3.6 Create-entry	126
4.4.3.7 Create-ref-optional	127
4.4.3.8 Create-identifier	128
4.4.3.9 Create-initial-element	129
4.4.4 Create-expression	130
4.4.4.1 Collect-subscripts	131
4.4.4.2 Apply-by-name-parts	132
4.4.4.3 Apply-subscripts	133
4.4.4.4 Create-value-reference	134
4.4.4.5 Trim-dd	135
4.4.4.6 Create-named-constant-reference	136
4.4.4.7 Create-argument-list	137
4.4.4.8 Create-builtin-function-reference	138
4.4.4.9 Create-pseudo-variable-reference	139
4.4.4.10 Test-matching	140
4.4.4.11 Select-generic-alternative	141
4.4.4.12 Test-generic-matching	142
4.4.4.13 Test-generic-aggregation	143
4.4.4.14 Test-generic-description	144
4.4.4.15 Test-generic-precision	145
4.4.6 Create-picture	146
4.4.6.1 Create-numeric-picture	147
4.4.7 Create-constant	148
4.5 Validation of the Abstract Procedure	149
4.5.1 Validate-declaration	150
4.5.2 Validate-automatic-declaration	151
4.5.3 Validate-based-declaration	152
4.5.4 Validate-controlled-declaration	153
4.5.5 Validate-defined-declaration	154
4.5.6 Validate-parameter-declaration	155
4.5.7 Validate-static-declaration	156
4.5.8 Validate-descriptor	157
4.5.9 Evaluate-restricted-expression	158
4.5.10 Apply-constraints	159
4.5.11 Test-constraints	160
4.6 Validate-program	161
4.6.1 Validate-external-declaration	162

CHAPTER 5: THE PL/I INTERPRETER	143
5.0 Introduction	143
5.1 The Interpretation-state	143
5.1.1 Directories	143
5.1.2 Block State	143
5.1.3 File Information	144
5.1.4 Storage and Values	145
5.1.5 Generators, Evaluated Data Descriptions, and Evaluated Targets	146
5.1.6 Dataset	147
5.2 Terminology and Definitions	148
5.2.1 Current	148
5.2.2 Block	148
5.3 The Interpret Operation and the Initialization of the Interpretation State	148
5.3.1 Interpret	149
5.3.2 Initialize-interpretation-state	149
5.3.3 Build-file-directory-and-informations	150
5.3.4 Build-fid	150
5.3.5 Build-controlled-directory	151
5.3.6 Allocate-static-storage-and-build-static-directory	151
5.3.7 Program-epilogue	151

CHAPTER 6: FLOW OF CONTROL	153
6.0 Introduction	153
6.1 Program Activation and Termination	153
6.1.1 Program Termination	153
6.1.1.1 Execute-stop-statement	153
6.1.1.2 Stop-program	153
6.2 Block Activation and Termination	154
6.2.1 Activate-procedure	154
6.2.1.1 Install-arguments	155
6.2.2 Activate-begin-block	155
6.2.3 Prologue	156
6.2.4 Epilogue	156
6.3 Control within a Block	157
6.3.1 Normal-sequence	157
6.3.1.1 Advance-execution	157
6.3.2 Execute-executable-unit	157
6.3.3 Execute-begin-block	157
6.3.4 Execute-group	158
6.3.4.1 Establish-controlled-group	158
6.3.4.2 Initialize-spec-options	159
6.3.4.3 Test-spec	161
6.3.4.4 Establish-next-spec	161
6.3.4.5 Test-termination-of-controlled-group	162
6.3.5 Execute-if-statement	163
6.3.5.1 Establish-truth-value	163
6.3.6 Execute-call-statement	163
6.3.6.1 Entry-references	164
6.3.6.1.1 Evaluate-entry-reference	165
6.3.7 Execute-goto-statement	166
6.3.7.1 Local-goto	166
6.3.7.2 Trim-group-control	167
6.3.8 Execute-null-statement	167
6.3.9 Execute-return-statement	167
6.3.10 Execute-end-statement	168
6.4 Conditions and Interrupts	169
6.4.1 Conditions	169
6.4.1.1 Raise-condition	169
6.4.1.2 Test-enablement	170
6.4.1.3 Execute-signal-statement	171
6.4.1.4 Evaluate-named-io-condition	171
6.4.2 Interrupts	171
6.4.2.1 Execute-on-statement	171
6.4.2.2 Execute-revert-statement	172
6.4.3 Interrupt	172
6.4.4 System-action	174
6.4.4.1 Comment	174
CHAPTER 7: STORAGE AND ASSIGNMENT	175
7.0 Introduction	175
7.1 The Generation	175
7.1.1 The Number of Elements in the Storage-index-list of a Generation	175
7.1.2 Correspondence between an Item-data-description and a Basic-value	176
7.1.3 Value of a Generation	177
7.1.4 Value of Storage Index	178
7.2 The Allocation of Storage	180
7.2.1 Execute-allocate-statement	180
7.2.2 Allocate-controlled-storage	181
7.2.3 Allocate-based-storage	181
7.2.4 Evaluate-in-option	182
7.2.5 Allocate	182
7.2.6 Suballocate	183
7.2.7 Evaluate-data-description-for-allocation	184
7.2.8 Find-directory-entry	185
7.2.9 Make-allocation-unit	185
7.2.10 Initialize-ref-er-options	186
7.2.11 Find-block-state-of-declaration	187
7.3 Initialization	188
7.3.1 Initialize-generation	188
7.3.2 Initialize-scalar-element	189

7.3.3 Initialize-array	189
7.4 The Freeing of Storage	191
7.4.1 Execute-free-statement	191
7.4.2 Free-controlled-storage	191
7.4.3 Free-based-storage	192
7.4.4 Deduce-in-option	193
7.4.5 Free	193
7.5 Assignment	194
7.5.1 The Assignment Statement	194
7.5.2 Target References	194
7.5.2.1 Evaluated Targets	195
7.5.3 The Assignment Operation	196
7.5.3.1 Promote-and-convert	197
7.5.3.2 The Set-storage Operation	198
7.5.4 Pseudo-variables	198
7.5.4.1 Imag-pv	199
7.5.4.2 Onchar-pv	200
7.5.4.3 Onsource-pv	200
7.5.4.4 Pageno-pv	201
7.5.4.5 Real-pv	202
7.5.4.6 String-pv	203
7.5.4.7 Substr-pv	204
7.5.4.8 Unspec-pv	205
7.6 Variable-reference	205
7.6.1 Evaluate-variable-reference	205
7.6.1.1 Connected Generations	206
7.6.2 Select-based-generation	207
7.6.3 Check-based-reference	207
7.6.4 Overlay-strings	208
7.6.5 Evaluate-data-description-for-reference	209
7.6.6 Select-qualified-reference	210
7.6.7 Select-subscripted-reference	212
7.6.8 Evaluate-by-name-parts-list	213
7.6.9 Evaluate-defined-reference	213
7.6.10 Evaluate-simply-defined-reference	214
7.6.11 Adjust-bound-pairs	215
7.6.12 Evaluate-isub-defined-reference	215
7.6.13 Expand-list-of-subscript-lists	217
7.6.14 Transform-subscript-list	218
7.6.15 Evaluate-string-overlay-defined-reference	218
7.6.16 Check-simply-defined-reference	219
7.6.17 Extract-slice-of-array	219
7.7 Reference to Named Constant	220
7.7.1 Evaluate-named-constant-reference	220
7.7.2 Search-file-directory	223
CHAPTER 8: INPUT/OUTPUT	225
8.0 Introduction	225
8.1 Datasets	225
8.1.1 Record Datasets	225
8.1.2 Stream Datasets	225
8.2 Files	226
8.2.1 Record Files	226
8.2.2 Stream Files	226
8.3 I/O Conditions	226
8.3.1 Raise-io-condition	227
8.4 Evaluate-file-option	228
8.5 File Opening and Closing	228
8.5.1 The Open Statement	228
8.5.1.1 Execute-open-statement	228
8.5.1.2 Execute-single-opening	229
8.5.1.3 Open	229
8.5.1.4 Evaluate-tab-option	230
8.5.1.5 Evaluate-title-option	231
8.5.1.6 Evaluate-filename	231
8.5.2 The Close Statement	232
8.5.2.1 Execute-close-statement	232
8.5.2.2 Execute-single-closing	232
8.5.2.3 Close	233
8.6 The Record I/O Statements	233

8.6.1 The Read Statement	233
8.6.1.1 Execute-read-statement	233
8.6.1.2 Read	234
8.6.2 The Write Statement	235
8.6.2.1 Execute-write-statement	235
8.6.2.2 Write	236
8.6.3 The Locate Statement	237
8.6.3.1 Execute-locate-statement	237
8.6.4 The Rewrite Statement	238
8.6.4.1 Execute-rewrite-statement	238
8.6.4.2 Rewrite	239
8.6.5 The Delete Statement	240
8.6.5.1 Execute-delete-statement	240
8.6.5.2 Delete	241
8.6.6 Operations Applicable to Record I/O	241
8.6.6.1 Evaluate-info-option	242
8.6.6.2 Evaluate-pointer-set-option	242
8.6.6.3 Evaluate-keyfrom-option	242
8.6.6.4 Evaluate-ignom-option	243
8.6.6.5 Evaluate-keyto-option	243
8.6.6.6 Construct-record	244
8.6.6.7 Insert-record	244
8.6.6.8 Position-file	245
8.6.6.9 Evaluate-size	246
8.6.6.10 Exit-from-io	247
8.6.6.11 Trim-io-control	247
8.7 The Stream I/O Statements	247
8.7.1 The Get Statement	247
8.7.1.1 Execute-get-statement	247
8.7.1.2 Execute-get-file	248
8.7.1.3 Execute-get-string	249
8.7.1.4 Get-list	250
8.7.1.4.1 Parse-list-input	251
8.7.1.4.2 Parsing Categories for List Directed Input	252
8.7.1.5 Get-data	255
8.7.1.5.1 Parse-data-input-name	256
8.7.1.5.2 Parse-data-input-value	257
8.7.1.5.3 Parsing Categories for Data Directed Input	257
8.7.1.6 Get-edit	258
8.7.1.6.1 Execute-input-control-format	259
8.7.1.6.2 Execute-input-data-format	261
8.7.1.6.2.1 Validate-input-format	263
8.7.1.7 Input-stream-item	264
8.7.1.8 Basic-character-value	264
8.7.1.9 Basic-bit-value	264
8.7.1.10 Input-stream-item-for-edit	265
8.7.2 The Put Statement	265
8.7.2.1 Execute-put-statement	266
8.7.2.2 Execute-put-file	267
8.7.2.3 Execute-put-string	267
8.7.2.4 Put-list	268
8.7.2.5 Put-data	270
8.7.2.6 Put-edit	271
8.7.2.6.1 Execute-output-control-format	272
8.7.2.6.2 Execute-output-data-format	273
8.7.2.6.3 Edit-numeric-output	273
8.7.2.7 Output-string	275
8.7.2.8 Output-string-item	276
8.7.2.9 Output-stream-item	277
8.7.2.10 Tab	277
8.7.2.10.1 Output-tab	277
8.7.2.11 Put-line	277
8.7.2.12 Put-page	278
8.7.3 Operations Applicable to Stream I/O	278
8.7.3.1 Skip	278
8.7.3.2 Evaluate-current-column	279
8.7.3.3 Evaluate-current-line	279
8.7.3.4 Establish-next-data-item	280
8.7.3.4.1 Expand-edit	282
8.7.3.4.2 Expand-generation	282

8.7.3.4.3 Make-name-and-subscript-list	283
8.7.3.4.4 Expand-name-and-subscript	284
8.7.3.4.5 Subscript-to-comma-subscript	285
8.7.3.4.6 Identifier-to-dotname	285
8.7.3.5 Establish-next-format-item	285
8.7.3.6 Evaluate-format-item	286
8.7.3.6.1 Evaluate-format-expression	287
CHAPTER 9: EXPRESSIONS AND CONVERSION	289
9.0 Introduction	289
9.1 Aggregate Expressions	289
9.1.1 Scalar and Aggregate Types	289
9.1.1.1 Aggregate Type of a Data Description	289
9.1.1.2 Scalar Elements	289
9.1.1.3 Treatment of Scalars	290
9.1.1.4 Compatibility	290
9.1.1.5 Correspondence	291
9.1.1.5.1 Correspondence of Scalar Elements	291
9.1.1.5.2 Correspondence of Data Types	292
9.1.1.6 Generate-aggregate-result	293
9.1.2 Integer Type	295
9.1.2.1 Evaluate-expression-to-integer	295
9.1.3 Derived Data Types	295
9.1.3.1 Derived Base, Scale, and Mode	295
9.1.3.2 Converted Precision	296
9.1.3.3 Derived String Type	297
9.1.3.4 Further Definitions for Character and Bit Strings	297
9.1.4 Arithmetic Results	298
9.1.4.1 Conditions in Expressions	299
9.1.5 Expressions	299
9.1.6 Value References	299
9.1.7 Constants	300
9.1.8 Isubs	300
9.1.9 Parenthesized Expressions	300
9.1.10 Arguments	301
9.2 Prefix Operators	301
9.2.1 Prefix Expressions	301
9.2.2 Definition of the Prefix Operators	301
9.2.2.1 Prefix-minus	302
9.2.2.2 Prefix-plus	302
9.2.2.3 Prefix-not	303
9.3 Infix Operators	303
9.3.1 Infix Expressions	303
9.3.2 Definition of the Infix Operators	304
9.3.2.1 Infix-add	304
9.3.2.2 Infix-and	304
9.3.2.3 Infix-cat	305
9.3.2.3.1 Concatenation of String Values	305
9.3.2.4 Infix-divide	306
9.3.2.5 Infix-eq	306
9.3.2.5.1 Compare	308
9.3.2.6 Infix-ge	308
9.3.2.7 Infix-gt	309
9.3.2.8 Infix-le	309
9.3.2.9 Infix-lt	310
9.3.2.10 Infix-multiply	311
9.3.2.11 Infix-ne	311
9.3.2.12 Infix-or	312
9.3.2.13 Infix-power	313
9.3.2.14 Infix-subtract	314
9.4 Built-in-functions	314
9.4.1 Built-in-function Reference	314
9.4.2 Special Terms Defined for Built-in-functions	314
9.4.2.1 Definition of N	315
9.4.2.2 The Arguments p and q	315
9.4.3 Operations Used in Built-in-function Definitions	316
9.4.3.1 Get-established-onvalue	316
9.4.4 Definition of the Built-in-functions	316
9.4.4.1 Abs-bif	317
9.4.4.2 Acos-bif	317

9.4.4.3 Add-bif	317
9.4.4.4 Addr-bif	318
9.4.4.5 After-bif	318
9.4.4.6 Allocation-bif	319
9.4.4.7 Asin-bif	319
9.4.4.8 Atan-bif	320
9.4.4.9 Atand-bif	321
9.4.4.10 Atanh-bif	321
9.4.4.11 Before-bif	322
9.4.4.12 Binary-bif	322
9.4.4.13 Bit-bif	323
9.4.4.14 Bool-bif	323
9.4.4.15 Ceil-bif	324
9.4.4.16 Character-bif	325
9.4.4.17 Collate-bif	325
9.4.4.18 Complex-bif	326
9.4.4.19 Conjg-bif	326
9.4.4.20 Copy-bif	327
9.4.4.21 Cos-bif	327
9.4.4.22 Cosd-bif	328
9.4.4.23 Cosh-bif	328
9.4.4.24 Date-bif	329
9.4.4.25 Decat-bif	330
9.4.4.26 Decimal-bif	330
9.4.4.27 Dimension-bif	331
9.4.4.28 Divide-bif	331
9.4.4.29 Dot-bif	332
9.4.4.30 Empty-bif	332
9.4.4.31 Erf-bif	333
9.4.4.32 Erfc-bif	333
9.4.4.33 Every-bif	334
9.4.4.34 Exp-bif	334
9.4.4.35 Fixed-bif	335
9.4.4.36 Fixed-bif	335
9.4.4.37 Floor-bif	336
9.4.4.38 Hbound-bif	336
9.4.4.39 High-bif	337
9.4.4.40 Imag-bif	337
9.4.4.41 Index-bif	338
9.4.4.42 Lbound-bif	338
9.4.4.43 Length-bif	339
9.4.4.44 Lineno-bif	339
9.4.4.45 Log-bif	340
9.4.4.46 Log10-bif	340
9.4.4.47 Log2-bif	341
9.4.4.48 Low-bif	341
9.4.4.49 Max-bif	341
9.4.4.50 Min-bif	342
9.4.4.51 Mod-bif	343
9.4.4.52 Multiply-bif	344
9.4.4.53 Null-bif	345
9.4.4.54 Offset-bif	345
9.4.4.55 Onchar-bif	346
9.4.4.56 Oncode-bif	346
9.4.4.57 Onfield-bif	347
9.4.4.58 Onfile-bif	347
9.4.4.59 Onkey-bif	347
9.4.4.60 Onloc-bif	348
9.4.4.61 Onsource-bif	348
9.4.4.62 Pageno-bif	348
9.4.4.63 Pointer-bif	349
9.4.4.64 Precision-bif	349
9.4.4.65 Prod-bif	350
9.4.4.66 Real-bif	350
9.4.4.67 Reverse-bif	351
9.4.4.68 Round-bif	352
9.4.4.69 Sign-bif	352
9.4.4.70 Sin-bif	353
9.4.4.71 Sinc-bif	353
9.4.4.72 Sinh-bif	354
9.4.4.73 Some-bif	354
9.4.4.74 Sqrt-bif	354
9.4.4.75 String-bif	355
9.4.4.76 Substr-bif	355
9.4.4.77 Subtract-bif	356
9.4.4.78 Sum-bif	356
9.4.4.79 Tan-bif	357
9.4.4.80 Tanh-bif	357
9.4.4.81 Tann-bif	358
9.4.4.82 Time-bif	358
9.4.4.83 Translate-bif	359
9.4.4.84 Trunc-bif	360
9.4.4.85 Unspec-bif	361
9.4.4.86 Valid-bif	361
9.4.4.87 Verify-bif	361
9.5 Conversion	363
9.5.1 Conversion of Scalar Values	363
9.5.1.1 Informal Invocation of Convert	366
9.5.1.2 Convert-to-fixed	366
9.5.1.3 Convert-to-float	367
9.5.1.4 Convert-to-bit	368
9.5.1.5 Convert-to-character	369
9.5.1.6 Conversion to Float Decimal	370
9.5.1.7 Conversion from String or Picture to Arithmetic	371
9.5.1.8 Basic Numeric Value of a String	372
9.5.1.9 Evaluate-real-constant	373
9.5.2 Numeric Pictures	374
9.5.2.1 Editing Numeric Pictures	375
9.5.2.2 Editing a Numeric Picture Field	376
9.5.2.3 Validity of a Numeric Pictured Value	380
9.5.2.4 Validity of a Field of a Numeric Pictured Value	381
9.5.3 Character Pictures	382
9.5.3.1 Test-char-pic-char	382

INDEX

Tables

Table 4.1 Concrete Terminals of Significance to Test-generic-description	129
Table 4.2 Table of <bit-values> as a Function of <symbols> and <radix-factors> for Create-constant	136
Table 9.1 Table of Converted Precisions as a function of Target and Source Attributes	236
Table 9.2 Table of Scalar-results as a Function of <bit-values> for Bool-bif	324
Table 9.3 Table of <symbols> as a Function of <suppression-type> for Edit-numeric-picture-field	379
Table 9.4 Table of <symbols> as a function of <numeric-picture-elements> and <character-string-values> for Edit-numeric-picture-field	379

Chapter 1: Scope and Overviews

1.0 Scope

This document defines the computer programming language PL/I. It is intended to serve as an authoritative reference rather than as a tutorial introduction.

The definition is accomplished by specifying a conceptual PL/I machine which translates and interprets intended PL/I programs. Section 1.1 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 1.2, and the detailed specification of the notation to be used follows in Section 1.3. The main body of the definition is then begun at Section 1.4, and is completed by Chapters 2 through 9.

1.1 An Informal Guide to the PL/I Definition

1.1.1 A SUMMARY OF PL/I

A PL/I program consists of a set of procedures, 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 and which may be related to the organization of the document.

Structural:	PROCEDURE ENTRY BEGIN DO END
Declarative:	DECLARE DEFAULT FORMAT
Flow of Control:	CALL RETURN IF GO TO Null Statement STOP
	ON REVERT SIGNAL
Storage:	ALLOCATE FREE Assignment Statement
	Interrupt Handling

Input/Output:	OPEN CLOSE GET } PUT } Stream I/O READ WRITE LOCATE REWRITE DELETE Record I/O Arithmetic } CHARACTER } BIT or PICTURE AREA ENTRY FILE FORMAT LABEL OFFSET POINTER
---------------	---

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:

1.1.2 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 external procedures (i.e. procedures not contained in any other procedure) are processed by a Translator. This processing involves systematically analyzing, transforming, and validating each external procedure. The analysis uses a grammar known as the Concrete Syntax to produce the concrete form of an external procedure 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 external procedures 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.

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 1.2.

Within each chapter, the sections are logically organized, and the Table of Contents may be used to obtain an overview of the structure.

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 Middle-level Syntax, and study of rules CW110, 111, 119, 122, and 123 reveals that an extra pair of parentheses is required around the form {input-target-commalist} DO {do-spec}. This is in order to resolve the ambiguity exhibited in this example. The correct form is either

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

or

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

depending on whether the last input value is intended to be assigned to N or to A(N,J).

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

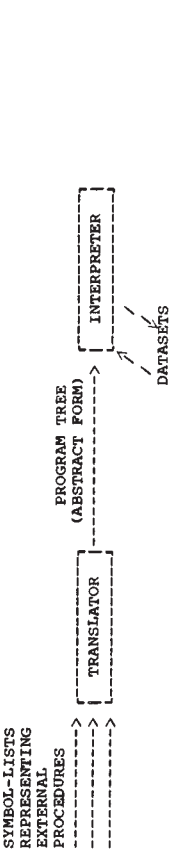
May the KEYTO option on a READ statement specify that the key be assigned to a substring of a variable?

ANSWER

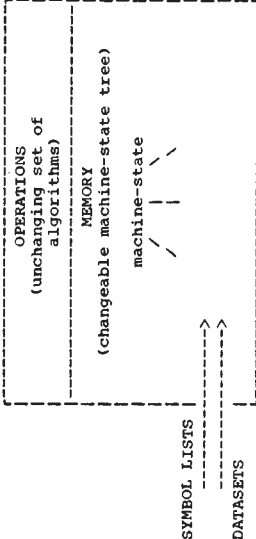
The Concrete Syntax for a {keyto-option} shows merely that a {reference} must be specified. However, the Abstract Syntax shows the form of a program after the Translator has completed all declarations, and has thus been able to associate each reference with the appropriate declaration and make more subtle distinctions. The rule A117 for <keyto-option> shows {target-reference} {scalar & character} ", and A179 shows that this permits a <pseudo-variable-reference>, e.g. the SUBSTR pseudo-variable. The parenthesized constraint "(scalar & character)" shows that it must be a single target (not an array or structure) which is character-valued.

No further restrictions are found by using the index to search the translator and interpreter, so that the answer is: yes, provided the substring is a scalar of character type.

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.

1.1.3 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
	Common Subroutines for Chapters 5-8

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.

Chapter 4 defines the translator whereby each of the individual PL/I program portions (external procedures) is translated from the submitted character string form to tree form and appended to the program tree. This process involves the parsing of each external procedure 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 <put-statement>. However, the Abstract Syntax shows that a <file-option> must be present in <put-file> (A124), and this means that if it was absent in the concrete form, it must 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 {put-statement} (Step 2 of the operation complete-options, Section 4.3.1.1).

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 1.4.3.3 and 5.3.1.

In Step 1 of operation interpretation-phase (Section 1.4.3.3) an <entry-value> is obtained from outside the definition. It designates the first procedure in the <program> to be activated. The "must not" condition specifies that the document gives no meaning to a program whose <entry-point> designated by the <entry-value> contains a <parameter-name-list>. (See Section 1.3.3.4 for the definition of "must".)

Additionally, in Step 2 of operation interpret (Section 5.3.1), an <evaluated-entry-reference> containing only an <entry-value> and not an <established-argument-list> (see production rule M25 in Section 5.1.2) 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 (Section 6.4.2.2) deletes a member from the current <established-on-unit-list> 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 <established-on-unit>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 is treated in Chapter 7.

The operation initialize-array (Section 7.3.3) iterates over Step 4 while n<nt. Since n 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.

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 RECCRD; 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 8.5.1.3) shows that <print> implies <stream> in Step 2, and that <stream> and <record> cause the result <fail> to be returned 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 8.5.1.2).

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

QUESTION

Is it possible to add an array to a structure?

ANSWER

The evaluation of expressions in Chapter 9 begins by considering the general treatment of aggregate operands. The definition of compatibility in Section 9.1.1.4 leads us to Case 3, yielding the answer that the array must be such that an element of it is compatible with the given structure. Applying the test for compatibility again, this would be true if the element is a scalar (Case 1) or a compatible structure (Case 2 and further recursions).

1.1.4 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 1.3.

1.1.4.1 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.

1.1.4.2 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

```
<p> ::= <q> | [<r-list>] <s> * <t> [<u>]
```

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 <p> must have subtrees as described by the right-hand side of the rule. There are three types of metalanguage operator, for concatenation (indicated by juxtaposition in the rule), permutation ("*") and choice between alternatives ("|"). In descending order of precedence. Parenthesized expressions 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 <q> alone, and next partition the other alternative by the "*" operator and decide to use the permutation which reverses these partitions to yield

```
<t> [<u>] [<r-list>] <s>
```

The brackets indicate that their contents may be optionally omitted, which we choose to do here, and the braces enclose a syntax expression which must be interpreted according to the method just described. Choosing the alternative <r-list> 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. <r-list> indicates, by convention, that it is to have one or more immediate subtrees of type <r> beneath it. We then apply the rule for <r> 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.

1.1.4.3 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 <machine-state> or change the <machine-state>, 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 <program> itself. (The metabackets "<" and ">" 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 1.1 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.

6.2.3 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 block, that contains <automatic> or <defined>, perform Step 2.1.
- Step 2. For each <declaration>, d, of the current block, that contains <automatic>

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>: id g. <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.
```

Example 1.1. The Prologue Operation.

As an example of an operation, consider the definition of the prologue operation in Example 1.1. This begins with an introductory paragraph which gives some guidance to the reader, 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 <machine-state> 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 <automatic>, 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 <true>", but this is only an alternative form of replacement with the usual "implicit dereferencing" of the variable name, meaning "replace the tree designated by the strict designator value of tv by <true>".

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 mechanism 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 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.

1.2 Relationships between an Implementation and this Definition

The inputs of the conceptual PL/I machine are one or more {symbol-list}s representing PL/I external procedures, an {entry-value}, and a {dataset-list}. 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 1.2.2, together with the representation of a program-run's elements ({symbol-list}s, {entry-value}, and {dataset-list}) 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 {data-sets} of the {machine-state}, and an implementation is free to achieve them by any means. The operations and other parts of the {machine-state} 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 the 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 the 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 1.2.1.

Note that 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.

1.2.1 FLEXIBILITIES OF INTERPRETATION

Through use of the terms "optionally" 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.

1.2.1.1 Reflection 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.

1.2.1.2 Quantitative Restrictions

An implementation may make quantitative restrictions not contained in the 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.

1.2.1.3 Operating Environment

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

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

1.2.1.4 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}, {value-reference}, {variable-reference}, or {target-reference} by evaluating its contained {expressions}, {value-references}, and {variable-references} in any order. Note that 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, if the result depends on the value returned by a contained {procedure-function-reference}, then this {procedure-function-reference} must be evaluated.
- When determining whether or not an evaluation must be performed, the implementation may ignore the possibility that this evaluation could raise one or more interrupts.

1.2.1.5 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 1.2.1.4, 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, stringsize, stringrange, subscriptrange, storage, area, error.

1.2.1.6 Input/Output

- (1) The <environment> attribute and option are provided for the specification of implementation-defined information concerned with the manipulation of the <file-values> and <datasets>. 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-values> and <datasets>. If no <environment> is present, the algorithms work as given. An <expression> appearing in an <environment> is evaluated at an implementation-defined point.
- (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>,iv) 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. The conditions must be raised in the same order as they would have been raised had the strict order of execution been followed.
- (3) If there is a <copy-option> in a <get-statement>, an implementation may deviate from the strict order of execution of the output-string-item operation. In particular, an implementation may interleave the execution of this operation with the execution of any other operations in the <get-statement> which follow it. If there are two or more output-string operations, they must be executed in the same order with respect to each other as they would have been performed under the strict order of execution, independently of their order with respect to the other operations of the <get-statement>.
- (4) During the execution of any input/output statement which contains a <file-option> or a <copy-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. In the event that an output operation is being executed as a part of file closing during program epilogue and circumstances are such that the <transmit-condition> is to be raised, the implementation may perform an implementation-defined action.
- (5) 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>.
- (6) A <put-string> must not immediately contain a <target-reference> that identifies an <allocation-unit> that is referenced by the evaluation of the <output-specification> of the <put-string>.

1.2.1.7 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, conversion, or string-size 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 <block-state> at the time the interrupt was raised; i.e. the block of interrupt. An <allocation-unit> is used in block B if it is referenced by any operation of the PL/I machine while B is the current <block-state>.

- (2) Let B be a <block-state>; let E be an <executable-unit> executed while B is the current <block-state>. During the execution of E, various <generation>s are returned by invocations of evaluate-target-reference. Let T be the set of <allocation-units> designated by these <generation>s, i.e., the targets of a given statement execution.

During the execution of E whenever the "interrupt" operation is invoked, all <basic-values> contained in any member of T are set <undefined>. If control reaches Step 4 of the "interrupt" operation, the original <basic-values> are restored, unless their containing <allocation-unit> was assigned one or more <basic-values> by the <on-unit> or its dynamic descendants. In the case of such assignment, the <basic-values> of the <allocation-unit> are set <undefined>.

1.2.2 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) Circumstances in which TRANSMIT condition is raised (1.2.1.6).
- (2) Actions performed, instead of raising the RECORD, KEY, or TRANSMIT condition, when an output operation is being executed as a part of a file closing during program epilogue and circumstances are such that, in all other contexts, the condition would be raised (1.2.1.6, 8.5.2.3, and 8.6.6.9).
- (3) Determination of the <dataset-list> passed to the "interpret" operation (1.4.3.3).
- (4) Determination of the <entry-value> passed to the "interpret" operation (1.4.3.3).
- (5) ENVIRONMENT attribute and option syntax (2.4.4.4) and semantics (Chapter 8).
- (6) OPTIONS attribute and option syntax (2.4.4.4) and semantics.
- (7) Extralingual characters in data character set (2.5.5 and 2.6.2).
- (8) The form of the {text-name} in the INCLUDE construct (2.5.7).
- (9) Collating sequence, hardware representations, graphic representations, and symbol names of an implementation's character set (2.6 and 9.4.4.17).
- (10) Default precisions of arithmetic data (4.3.6.3).
- (11) Default AREA size (4.3.6.3).
- (12) Consistency requirements for ENVIRONMENT and OPTIONS attributes in EXTERNAL declarations (4.6.1), and for OPTIONS attributes in ENTRY references (6.3.6.1.1).
- (13) Size of an <area-value> passed as a dummy argument (6.3.6.1.2).

1.3 The Metalanguage

Following the introductory material in Section 1.1.4, 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:".

Tables are enclosed in a frame of straight lines.

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

Examples, which are not part of the definition, appear in a frame with lines at the sides and asterisks at the top and bottom. Introductory paragraphs to syntaxes and operations are likewise without any definitive force.

1.3.1 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 must be 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 1.3.1.1. These trees are then made more specific by discussing in Section 1.3.1.2 the basic nature of the objects used in composing tree nodes. In Sections 1.3.1.3 and 1.3.1.4 the written notations used for individual nodes and then for whole trees are discussed. Section 1.3.1.5 deals with copies of trees.

Following this general section on trees and tree notations, Section 1.3.2 then discusses production-rules, which function in a declarative manner to specify the particular classes of trees used in this document.

(14) Information output when SNAP is specified in ON statement (6.4.3).

(15) Value returned by ONCODE builtin function (6.4.3).

(16) Standard system action for STORAGE condition (6.4.4).

(17) Standard system action for ERROR condition (6.4.4).

(18) Form of comment output as standard system action (6.4.4.1).

(19) Situations when ERROR is raised.

(20) Situations when STORAGE is raised (7.2.5).

(21) Use of AREA size specification (7.2.6).

(22) Interpretation of UNSPEC pseudo-variable (7.5.4.8).

(23) Concrete representation of a <dataset> (8.1).

(24) The "size" of a <record> (8.1.1).

(25) Length of a key (8.1.1).

(26) Representation of stream dataset control items (8.1.2).

(27) Determination of a <dataset> on file opening (8.5.1.3).

(28) Default LINESIZE for a STREAM OUTPUT file (8.5.1.3).

(29) Default PAGESIZE for a PRINT file (8.5.1.3).

(30) Default tab positions for a PRINT file (8.5.1.3).

(31) Length of file title (8.5.1.5).

(32) Circumstances in which the KEY condition is raised (8.5.2.3, 8.6.2.2, 8.6.3.1, and 8.6.6.10).

(33) 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 (8.6.3.1, 8.6.4.2, and 8.6.6.9).

(34) Raising of RECORD condition by WRITE and LOCATE statements (8.6.6.9).

(35) Position of records in a KEYED SEQUENTIAL file (8.6.6.9).

(36) Items output by "PUT DATA," (8.7.2.5).

(37) Maximum <number-of-digits> used in editing relative to a <fixed-point-format> (8.7.2.6.3).

(38) Maximum <number-of-digits> for each combination of <base> and <scale> (4.4.3.5, 9.1.3.2, 9.5.1.9, and elsewhere).

(39) Precision of integer-type (9.1.3.2).

(40) Determination of floating-point results of expressions and builtin functions (9.1.4).

(41) Results of ROUND builtin function with floating-point argument (9.4.4.68).

(42) Length of string returned by TIME builtin function (9.4.4.82).

(43) Result of UNSPEC builtin function (9.4.4.85).

(44) Results of numeric conversions (9.5.1.2, 9.5.1.3, and 9.5.1.6).

(45) Number of digits in the exponent of a floating-point number (9.5.1.5).

(46) Representation of currency symbol and digit and sign symbols (9.5.2.2).

1.3.1.1 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 has 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 that 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 .

(Chapter 9 (see Section 9.1.1.5) contains further definitions of "correspond" useful in certain special contexts.)

1.3.1.2 Node Objects

The definition of node given in Section 1.3.1.1 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 1.3.1.2.1 and 1.3.1.2.2.

1.3.1.2.1 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 1.3.1.2.2).

1.3.1.2.2 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 1.3.1.2.1. 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 that this construction, together with the uniqueness rule in Section 1.3.1.2.1, means that it is possible to examine a designator and constructively determine if the potentially designated node has been in fact deleted.

18

Chapter 1: Scope and Overview 17

1.3.1.4 Tree Notation

1.3.1.4.1 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 1.3.1.3), 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 <data-type> which immediately contains a <non-computational-type> which immediately contains <format> and <local>, can be written as: <data-type>.dt: <non-computational-type>: <format> <local>;. Indentation is often used as a visual aid, e.g.

```

<data-type>.dt:
  <non-computational-type>:
    <format>
    <local>;

```

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 1.3.3.1) 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 {symbol-list} which has this enclosed string as its concrete-representation.

1.3.1.4.2 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 1.3.1.4.1 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.

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 1.3.3).

1.3.1.3 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 lower case letters, hyphens, and numbers, including surrounding brackets of the form "f" and "f", "<" and ">", or "<" and ">" 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 57 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 straightforward individual character denotations. Capital letters are used for the letters, while the quoted symbol (i.e. that which is inside the following quotes) "B" is used for blank. In situations where confusion might arise, the concrete brackets are used around the straightforward denotations, e.g. f, f.
- (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 "LINE" and "FLOAT". In situations where confusion might arise, the concrete brackets are used around the straightforward denotations.

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 57 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 B. 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.

1.3.1.5 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.

1.3.2 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 1.3.1. 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.

1.3.2.1 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 1.3.2.3) 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, types, and order of the immediate subnodes of a node whose type is the defined category-name. This is done by interpreting the syntactic-expression of the production-rule according to the algorithm given in Section 1.3.2.4.

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. Five (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 Middle-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", "Cm", "Cl", "A", and "M" respectively. The unlabelled 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.

The production-rule

A17. $\langle \text{bound-pair} \rangle ::= \langle \text{lower-bound} \rangle \langle \text{upper-bound} \rangle \mid \langle \text{asterisk} \rangle$
defines two possibilities, which may be written as (a) or drawn as (b)

(a) $\langle \text{bound-pair} \rangle:$
 $\langle \text{lower-bound} \rangle$ or
 $\langle \text{upper-bound} \rangle;$

(b) $\langle \text{bound-pair} \rangle$
or
 $\langle \text{lower-bound} \rangle \quad \langle \text{upper-bound} \rangle$
 $\langle \text{asterisk} \rangle$

A node whose type is $\langle \text{identifier-list} \rangle$ may have any non-zero number of immediate subnodes of type $\langle \text{identifier} \rangle$, i.e.

$\langle \text{identifier-list} \rangle:$ or $\langle \text{identifier-list} \rangle:$ or $\langle \text{identifier-list} \rangle:$
 $\langle \text{identifier} \rangle$ $\langle \text{identifier} \rangle$ $\langle \text{identifier} \rangle$
 $\langle \text{identifier} \rangle$ $\langle \text{identifier} \rangle$ $\langle \text{identifier} \rangle$
 $\langle \text{identifier} \rangle;$ $\langle \text{identifier} \rangle;$ $\langle \text{identifier} \rangle;$

and so on.

A node whose type is $\langle \text{parameter-commalist} \rangle$ may have any non-zero number of $\langle \text{parameter} \rangle$ immediate subnodes, but with nodes whose type is the PL/I character $\{, \}$ interspersed between adjacent ones, i.e.

$\langle \text{parameter-commalist} \rangle:$ or $\langle \text{parameter-commalist} \rangle:$ or $\langle \text{parameter-commalist} \rangle:$
 $\{ \text{parameter} \}$ $\{ \text{parameter} \}$ $\{ \text{parameter} \}$
 $\{ \text{parameter} \}$ $\{ \text{parameter} \}$ $\{ \text{parameter} \}$
 $\{ \text{parameter} \};$ $\{ \text{parameter} \};$ $\{ \text{parameter} \};$

and so on.

The production-rule

A67. $\langle \text{repeat-option} \rangle ::= \langle \text{expression} \rangle (\text{scalar})$

specifies that only $\langle \text{expression} \rangle$ s yielding scalar values (i.e. not aggregate values) are valid immediate subnodes of a $\langle \text{repeat-option} \rangle$.

Example 1.2. Examples of Syntax.

1.3.2.2 Complete and Partial Trees

Given a syntax, a complete tree with respect to that syntax is any tree which can be obtained by starting from a node of a given type, and repeatedly attaching subnodes to the nodes of the tree being developed according to the algorithm of Section 1.3.2.4, until an interpretation has been obtained for every node of the tree. 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.

Case 4. There is in the given syntax a defining production-rule for the given category-name.

An interpretation is an interpretation of the syntactic-expression written following the ":-" in the defining production-rule.

Case 5. (Otherwise).

The given category-name is a terminal with respect to the given syntax; the interpretation is the empty set.

An interpretation of a syntactic-expression is defined as follows:

Case 1. The syntactic-expression is a syntactic-unit.

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

An interpretation consists of the ordered set containing just this single category-name.

Case 1.2. The syntactic-unit is a syntactic-expression enclosed in the brackets "[" and "]."

An interpretation consists either of an interpretation of the enclosed syntactic-expression, or of the empty set.

Case 1.3. The syntactic-unit is a syntactic-expression enclosed in the braces "{" and "}."

An interpretation consists of an interpretation of the enclosed syntactic-expression.

Case 2. The syntactic-expression 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 a separator.

Consider all or-symbols occurring thus in the given syntactic-expression to partition it into a sequence of inner syntactic-expressions. An interpretation is one of any of these inner syntactic-expressions chosen arbitrarily.

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

Consider all bullets occurring thus in the given syntactic-expression to partition it into a sequence of inner syntactic-expressions. An interpretation is the same as one of a syntactic-expression formed by arranging these inner syntactic-expressions in an arbitrary order and omitting these bullets.

Case 2.3. (Otherwise).

(The syntactic-expression is a sequence of syntactic-expressions optionally separated by blanks.) An interpretation consists of the concatenation, in order, of interpretations of the syntactic-expressions of the sequence.

1.3.3 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.

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 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 1.3.3.4) of the metalanguage are in fact designed to assist in the process of building complete trees.

1.3.2.3 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 "}."

1.3.2.4 Application of the Production Rules

Given a syntax and a category-name, the algorithm shown just below obtains a (possibly empty) ordered set of category-names, termed here an interpretation with respect to the given syntax of the given category-name. In the process of constructing a complete tree with respect to the given syntax, any such ordered set may then be used as the corresponding types of an ordered set of immediate subnodes connected to any node whose type is the given category-name.

An interpretation of a category-name is defined as follows:

Case 1. The denotation of the given category-name, excluding any terminating bracket, ends with "-list".

An interpretation consists of an ordered set of any non-zero number of instances of that category-name whose denotation is obtained by deleting the "-list" from the denotation of the given category-name.

Case 2. The denotation of the given category-name, excluding any terminating bracket, ends with "-commalist".

An interpretation consists of an ordered set which:

(1) contains any non-zero number, n , of instances of the category-name whose denotation is obtained by deleting the "-commalist" from the denotation of the given category-name, and which

(2) contains $n-1$ instances of the category-name $\{, \}$, and which

(3) is arranged so that no two instances of the same category-name are adjacent.

Case 3. The denotation of the given category-name, excluding any terminating bracket, ends with "-designator".

An interpretation is a single member of the set of unique-names. If the category-name is of the form "x-designator", the unique-name must be of that of a node of type "x".

1.3.3.1 Nature of an Operation

An operation, applied to zero or more operands, may be performed by the processor (see Section 1.3.4), with the effect of:

- (1) changing the machine-state, 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 machine-state, 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 machine-state, 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 1.3.4 on "perform".)

1.3.3.2 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.

1.3.3.3 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.

Example 1.3. 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 1.3.3.4. 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 aggregate-value, u.

Step 2. In any order, convert each scalar-element of u to bit, to obtain v.

Step 3.

Case 3.1. Every scalar-element of v that does not contain a null-bit-string has a bit-string-value with every bit-value containing one-bit.

Let r be one-bit.

Case 3.2. (otherwise).

Let r be zero-bit.

Step 4. Return an aggregate-value containing a bit-string-value containing r.

Example 1.4. An Example of an Operation Containing a Step-list and a Case-set.

The "must" instruction specifies a test to be performed. If the test is not satisfied, the program (when combined with the particular initial entry-point and datasets if the interpretation-phase has begun) is rejected by the standard, and the conceptual PL/I machine stops. This is the only sense in which "must" and "must not" are used in operation descriptions.

1.3.3.5 Convert

Convert is an exception to the general rules for naming and performing (see Section 1.3.3.4). Use of this operation is generally specified in an informal style, e.g. "convert the value of the <expression> to integer-type". Details are given in Section 9.5.1.1.

1.3.3.6 Additional Notational Conventions

Common mathematical symbols are used with their usual meaning. In addition, the following notational conventions are used:

- denotes multiplication;
- / denotes division;
- ↑ denotes exponentiation;
- $\sum$ denotes iterated addition (summation); the result is taken as zero if the iteration range is empty;
- $\prod$ denotes iterated multiplication (product); the result is taken as one if the iteration range is empty;
- () denotes subscripting in the metalanguage as a notational convenience used with local names;
- π_i denotes the mathematical constant of that name;
- e denotes the mathematical constant of that name;
- i denotes the square root of -1;
- ceil(x) denotes the smallest integer larger than or equal to x;
- floor(x) denotes the largest integer smaller than or equal to x;
- min(x,y) denotes the value of x if x ≤ y, otherwise the value of y;
- max(x,y) denotes the value of x if x ≥ y, otherwise the value of y;
- log(x) denotes the natural logarithm of x;
- |x| denotes the absolute value of x.

1.3.3.7 Arithmetic

In the metalanguage, an arithmetic expression denotes the exact mathematical value. The situations in which the result may be approximated by the PL/I machine are defined in such operations as arithmetic-result.

In general the distinction between numbers and trees containing them is ignored. For example, if z is a tree of the form

```

<basic-value>;
<complex-value>;
<real-number>, x
<real-number>, y;;

```

then the arithmetic expression "z+1" denotes the complex number (x+1)+iy.

1.3.3.4 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.

In contrast, the "replace" instruction is used to indicate that the tree referenced by the named variable or operand is to be replaced with the specified tree. The replacement occurs exactly at the root of the referenced tree, and the named variable or operand henceforth references the replacement.

The "append" instruction, as in "append b to c", indicates that b is to be attached as the last immediate subtree of c. c may be any existing tree, or if it is uniquely specified, may be non-existent. This latter case causes the construction of the (single) node c. In order to make the specification of the potentially missing node c unique, the notation illustrated by "append b to c in d" can be employed to indicate that c may be missing and is to be constructed as a simple component of d.

The "attach" instruction, as in "attach b to c", indicates that b is to be attached to c as a component of c. If b can be an immediate component of c, then it is attached as an immediate component. Otherwise, there will be a unique way that a node of type b can be a simple component of c, and exactly the minimal necessary nodes which are both contained by c and also contain b are created so as to attach b as a simple component.

The "delete" instruction, as in "delete b" indicates that b is to be removed from its containing tree (and discarded). In addition, if b is a mandatory component of the tree which immediately contained b, say c, then the "delete" instruction is applied to c (i.e., c is discarded and the process continues with the tree which immediately contained c).

The "perform" instruction either calls another operation, possibly passing operands, possibly receiving a result, or calls for the execution of some steps with an operation out of the normal sequence. In calling other operations, references to trees specified by the argument list are passed to the named operation as operands (any missing arguments are given the value <absent> in the called operation). If the operation returns a result, then the term "to obtain" is used to indicate the obtained result. When other steps in an operation are performed, control returns to the instruction following the "perform" instruction.

The "go to" instruction indicates that the normal sequence of control is broken and that the named step is to be executed next.

The "terminate" instruction indicates that the execution of the current operation is to be terminated and control is to be returned to the calling operation. If control reaches the end of an operation that does not return a result, an implicit "terminate" instruction is assumed. The "return" instruction indicates that the current operation is to be terminated with a reference to a specified tree as the result.

The "if" instruction indicates that if the specified condition is true then the instruction list following the "then" is executed and the instruction list (if any) following the "otherwise" is skipped. If the condition is false the instruction list following the "then" is skipped and, if there is one, the instruction list following the "otherwise" is executed.

The "for each" instruction specifies actions that are to be carried out once for each member of a set of objects, in some sequence which may be in any order or in some specified order.

1.3.4 THE PROCESSOR

The processor is the active agent capable of performing various actions on the <machine-state> tree. These actions are carried out as directed by the written algorithm which comprises the PL/I definition.

A portion of the <machine-state> tree, either the <control-state> or a <statement-control>, holds the information which controls the processor. An <operation> is known by the processor if it is a simple component of the <control-state> or a <statement-control>. There is at most one active operation. An <operation> is active if:

- (1) it is the last member of its immediately containing <operation-list>, and
- (2) either:
 - (2.1) the <operation> is simply contained in an <operation-list> which is simply contained in a <statement-control> which is simply contained in a <block-state> that is the last member of its immediately containing <block-state-list>, or
 - (2.2) there are no <statement-controls> contained in the <machine-state>, and the <operation> is simply contained in an <operation-list> which is simply contained in the <control-state>.

(See Section 1.4.1 and Section 5.1 for the definitions of these category-names.)

The processor carries out actions as follows:

Whenever there exists a known active <operation>, the processor carries out the actions specified by the written description of the corresponding operation.

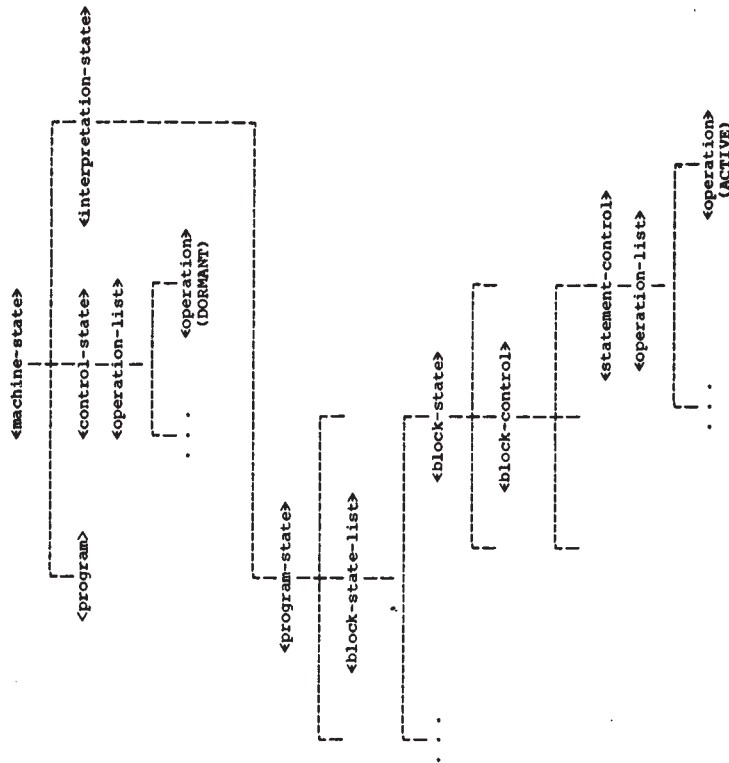
Whenever there exists no known active <operation>, the processor does nothing.

1.3.5 MECHANIZATION OF THE METALANGUAGE

A deeper understanding of the metalanguage may be obtained by considering how its formal mechanization may be carried out as an extension of the <machine-state>. Each <operation> in an <operation-list> may be given a subtree containing the name of the operation, the names and designator values of its operands, the names and designator values of its local variables, the local-trees constructed during the performance of this operation, and control information indicating which Step or Case is currently being executed, which members of an iterative "for each" have still to be performed, and so on. When a "perform" instruction causes a new operation to be invoked, a new <operation> tree will be appended to the <operation-list> and activity in the present operation will be suspended. On return from an operation, any local-tree returned by it as a result is copied back as a subnode of the preceding <operation> tree, and the <operation> tree for the terminating operation is deleted together with all its local information. Activity in the invoking operation is then resumed from the point at which it was suspended.

Example 1.5. Example Showing <operation-list>s of a <machine-state>.

During the translation-phase, the only <operation-list> is the one in the <control-state>. During most of the interpretation-phase, the <operations> in the <control-state> will be dormant while there is an active <operation> in a <statement-control> as shown below.



1.4 Initialization of the Machine-state

1.4.1 THE MACHINE-STATE

- M1. $\langle \text{machine-state} \rangle ::= \langle \text{program} \rangle$
 $\quad \quad \quad \langle \text{control-state} \rangle$
 $\quad \quad \quad \{ \langle \text{translation-state} \rangle \mid \langle \text{interpretation-state} \rangle \}$
- M2. $\langle \text{control-state} \rangle ::= \langle \text{operation-list} \rangle$
- M3. $\langle \text{translation-state} \rangle ::= \{ \langle \text{concrete-external-procedure} \rangle \}$
- M4. $\{ \langle \text{concrete-external-procedure} \rangle \} ::= \{ \{ \langle \text{declaration-commalist} \rangle \} \{ \text{procedure} \} \}$
- M5. $\langle \text{operation} \rangle ::=$

The exact structure of $\langle \text{operation} \rangle$ is left unformalized and unspecified. It must have adequate structure and capacity to represent the carrying out of the actions of an operation. This includes designating the particular operation and the current position within it, holding the operands given to the operation, and holding the values of any variables used by the operation (see Section 1.3.5).

The definitions of $\{ \langle \text{declaration} \rangle \}$ and $\{ \text{procedure} \}$ are given in Chapter 2; the definition of $\langle \text{program} \rangle$ is given in Chapter 3; the definition of $\langle \text{interpretation-state} \rangle$ is given in Chapter 5.

1.4.2 INITIALIZATION

The PL/I definition process begins by creating an initial $\langle \text{machine-state} \rangle$ tree, consisting of:

```

<machine-state>:
  <program>
  <control-state>:
    <operation-list>:
      <operation> for define-program.

```

The processor then performs the $\langle \text{operation} \rangle$ for define-program.

1.4.3 THE TOP-LEVEL OPERATIONS

1.4.3.1 Define-program

Operation: define-program

- Step 1. Perform translation-phase.
- Step 2. Perform interpretation-phase.
- Step 3. No action. (Reaching this point indicates the successful completion of the definition algorithm.)

1.4.3.2 Translation-phase

Operation: translation-phase

- Step 1. Append $\langle \text{translation-state} \rangle$ to the $\langle \text{machine-state} \rangle$.

Step 2.

Step 2.1. Obtain, from a source outside this definition, a sequence of characters composing a putative PL/I external procedure, constructed in the form of a $\{ \text{symbol-list} \}, \text{sl}$.

Step 2.2. Perform $\text{translate}(\text{sl})$ to obtain an $\langle \text{abstract-external-procedure} \rangle, \text{aep}$. Append aep to the $\langle \text{abstract-external-procedure-list} \rangle$ in the $\langle \text{program} \rangle$.

Step 2.3. Optionally go to Step 2.

Step 3. Perform validate-program.

Step 4. Delete the $\langle \text{translation-state} \rangle$.

1.4.3.3 Interpretation-phase

Operation: interpretation-phase

- Step 1. Obtain, from a source outside this definition, the following items:

- (1) A collection of information to be used for input/output, constructed in the form of a suitable $\langle \text{dataset-list} \rangle, \text{dl}$.
- (2) A designation, as the first to be activated, of one of the $\langle \text{entry-points} \rangle$ of a $\langle \text{procedure} \rangle$ simple component of $\langle \text{program} \rangle$, constructed in the form of a suitable $\langle \text{entry-value} \rangle, \text{ev}$. Such an $\langle \text{entry-point} \rangle$ must exist and must not have $\langle \text{parameter-name-list} \rangle$ or $\langle \text{returns-descriptor} \rangle$ components.

Step 2. Perform $\text{interpret}(\text{dl}, \text{ev})$ (See Section 5.3.1).

Chapter 2: Concrete Syntax

2.0 Introduction

The Concrete Syntax of PL/I is specified mainly by means of production-rules using the notation defined in Chapter 1. The first such rule defines a `{procedure}`, and subsequent rules define the permitted forms of a `{procedure}` and its components in increasingly fine detail, until every component is ultimately described in terms of sequences of characters of the language character set.

2.1 The Intent of this Definition

As the first stage of translation (Chapter 4), any given sequence of symbols is parsed to determine whether that sequence indeed represents a `{procedure}` valid according to a set of rules known in this document as the "Concrete Syntax".

2.1.1 CONCRETE AND ABSTRACT SYNTAXES

This formal Concrete Syntax is permissive in the sense that some of the constructs permitted are not actually valid `{procedures}`. Thus, for example, the sequence of symbols "DCL X FLOAT FIXED;" is a syntactically correct construct that may be parsed as a `{declare-statement}`. Errors of this sort will be detected later in the translation, because of a failure to satisfy the Abstract Syntax (Chapter 3).

2.2 Organization of the Concrete Syntax

The rules of the Concrete Syntax, which are context-free, are arranged in three levels, so that two context-dependent features of this grammar, namely the presence of blanks and comments, and the so-called "multiple closure", may be resolved at the interface between the levels.

The three levels of syntax correspond to the three levels of the parse algorithm described in Chapter 4.

2.3 The High-level Syntax of PL/I

2.3.1 PROCEDURE

CH1. `{procedure} ::= {prefix-list} {procedure-statement} {unit-list} {ending}`

2.3.2 UNIT

CH2. `{unit} ::= {statement-name-list} {declare-statement} | {default-statement} | {statement-name-list} {entry-statement} | {prefix-list} {format-statement} | {procedure} | {executable-unit}`

2.3.3 EXECUTABLE UNITS

CH3. `{executable-unit} ::= {prefix-list} {group} | {begin-block} | {on-statement} | {if-statement} | {executable-single-statement}`

CH4. `{if-statement} ::= {if-clause} {executable-unit} | {balanced-unit} ELSE {executable-unit}`

CH5. `{balanced-unit} ::= {prefix-list} {executable-single-statement} | {group} | {begin-block} | {on-statement} | {if-clause} {balanced-unit} ELSE {balanced-unit}`

CH6. `{group} ::= {do-statement} {unit-list} {ending}`

CH7. `{begin-block} ::= {begin-statement} {unit-list} {ending}`

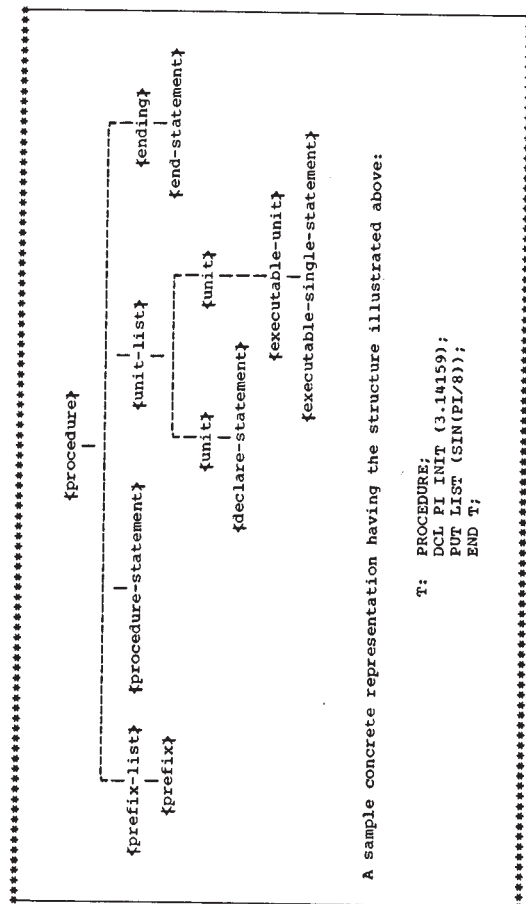
CH8. `{on-statement} ::= ON {condition-name-commalist} [SNAP] {on-unit} | SYSTEM;`

CH9. `{on-unit} ::= [{condition-prefix-commalist}]; | {executable-single-statement} | {begin-block}`

CH10. `{ending} ::= {statement-name-list} {end-statement}`

Example 2.1 illustrates the tree of the high-level structure of a simple `{procedure}`.

Example 2.2 illustrates the high-level structure of a simple `{if-statement}`.



Example 2.1. The High-level Structure of a Simple `{procedure}`.

2.4.2 STATEMENT

CM6. `{single-statement} ::= {statement-name-list} | {declare-statement} | {default-statement} | {end-statement} | {statement-name-list} {entry-statement} | {prefix-list} {executable-single-statement} | {begin-statement} {unmatched} | {do-statement} {unmatched} | {format-statement} | SYSTEM; | {procedure-statement} {unmatched}`

Note: `{unmatched}` is used only by the operation high-level-parse.

CM7. `{executable-single-statement} ::= {allocate-statement} | {assignment-statement} | {open-statement} | {call-statement} | {read-statement} | {close-statement} | {return-statement} | {delete-statement} | {free-statement} | {rewrite-statement} | {get-statement} | {set-statement} | {write-statement} | {goto-statement} | {locate-statement} | {null-statement}`

2.4.3 PREFIXES

2.4.3.1 Condition Prefixes

CM8. `{condition-prefix} ::= {computational-condition} | {disabled-computational-condition}`

CM9. `{computational-condition} ::= CONVERSION | FIXEDOVERFLOW | OVERFLOW | SIZE | STRINGRANGE | STRINGSIZE | SUBSCRIPTRANGE | UNDERFLOW | ZERO Divide`

CM10. `{disabled-computational-condition} ::= NOCONVERSION | NOFIXEDOVERFLOW | NOOVERFLOW | NOSIZE | NOSTRINGRANGE | NOSTRINGSIZE | NOSUBSCRIPTRANGE | NOUNDERFLOW | NOZERODIVIDE`

2.4.3.2 Statement Name Prefixes

CM11. `{statement-name} ::= {identifier} ({signed-integer-commalist})`

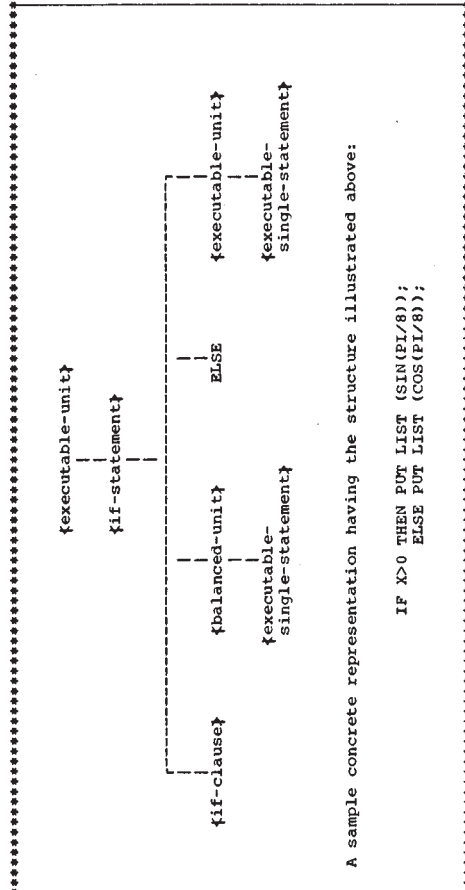
CM12. `{signed-integer} ::= [+ | -] {integer}`

2.4.4 DATA DECLARATION

CM13. `{declare-statement} ::= DECLARE {declaration-commalist}`

CM14. `{declaration} ::= {level} {identifier} ({declaration-commalist}) | {dimension-suffix} {attribute-list}`

CM15. `{level} ::= {integer}`



Example 2.2. The High-level structure of an `{if-statement}`.

2.4 The Middle-level Syntax of PL/I

2.4.1 SENTENCE

A `{sentence}` in the middle-level syntax corresponds to that which comes between semi-colons in a PL/I `{procedure}`.

CM1. `{sentence} ::= {prefixed-clause-list} {single-statement} | {else-part}`

CM2. `{else-part} ::= ELSE {prefixed-clause-list} {single-statement}`

CM3. `{prefixed-clause} ::= {prefix-list} | {if-clause} | ON {condition-name-commalist} [SNAP]`

CM4. `{prefix} ::= {condition-prefix-commalist} | {statement-name}`

CM5. `{if-clause} ::= IF {expression} THEN`

2.4.4.1 Dimension Attribute and Dimension Suffix

CM16. `{dimension-attribute} ::= DIMENSION { {dimension-suffix} }`
 CM17. `{dimension-suffix} ::= ( {bound-pair-commalist} )`
 CM18. `{bound-pair} ::= {lower-bound} {upper-bound} | *`
 CM19. `{lower-bound} ::= {extent-expression}`
 CM20. `{upper-bound} ::= {extent-expression}`
 CM21. `{extent-expression} ::= {expression} {refer-option}`
 CM22. `{refer-option} ::= REFER { {unsubscripted-reference} }`

2.4.4.2 Attributes

CM23. `{attribute} ::= {data-attribute}`
 AUTOMATIC
 BASED ({reference})
 BUILTIN
 CONDITION
 CONSTANT
 CONTROLLED
 DEFINED ({reference} | {reference})
 DIRECT
 ENVIRONMENT
 EXTERNAL
 {generic-attribute}
 INITIAL
 INPUT
 INTERNAL

KEYED
 LINE {unsubscripted-reference}
 LOCAL
 {options}
 OUTPUT
 PARAMETER
 POSITION ({expression})
 PRINT
 RECORD
 SEQUENTIAL
 STATIC
 STREAM
 UPDATE
 VARIABLE

2.4.4.3 Data Attributes

CM24. `{data-attribute} ::= ALIGNED`
 AREA ({area-size})
 AREA {precision}
 BIT ({maximum-length})
 CHARACTER ({maximum-length})
 COMPLEX {precision}
 DECIMAL {precision}
 {dimension-attribute}
 ENTRY (({description-commalist}))
 FILE
 FIXED {precision}
 FLOAT ({number-of-digits})
 FORMAT

LABEL
 MEMBER
 NONVARYING
 OFFSET ({reference})
 PICTURE ({picture})
 POINTER
 PRECISION {precision}
 REAL {precision}

{returns-descriptor}
 STRUCTURE
 UNALIGNED
 VARYING

CM25. `{area-size} ::= {extent-expression} | *`
 CM26. `{precision} ::= {number-of-digits} {scale-factor}`
 CM27. `{number-of-digits} ::= {integer}`
 CM28. `{scale-factor} ::= {signed-integer}`
 CM29. `{maximum-length} ::= {extent-expression} | *`
 CM30. `{description} ::= {level} { {dimension-suffix} } { {data-attribute-list} }`

Constraint: At least one subnode must be present.

CM31. `{picture} ::= {simple-character-string-constant}`

2.4.4.4 Environment and Options

CM32. `{environment} ::= ENVIRONMENT { {environment-specification} }`
 CM33. `{environment-specification} ::=`
 CM34. `{options} ::= OPTIONS { {options-specification} }`
 CM35. `{options-specification} ::=`

2.4.4.5 Generic

CM36. `{generic-attribute} ::= GENERIC ( { {generic-element-commalist} } )`
 CM37. `{generic-element} ::= reference WHEN ( {generic-description-commalist} )`
 CM38. `{generic-description} ::= {level} { {asterisk-bounds} } { {generic-data-attribute-list} } | *`

Constraint: At least one subnode must be present.

CM39. `{generic-data-attribute} ::= ALIGNED`
 AREA
 BINARY {generic-precision}
 BIT
 CHARACTER
 COMPLEX {generic-precision}
 DECIMAL {generic-precision}
 DIMENSION {asterisk-bounds} | REAL {generic-precision}
 ENTRY (({description-commalist}))
 FILE
 FIXED {generic-precision}
 FLOAT {generic-precision}
 FORMAT

LABEL
 MEMBER
 NONVARYING
 OFFSET
 PICTURE {picture}
 POINTER
 PRECISION {generic-precision}
 REAL {generic-precision}
 {returns-descriptor}
 STRUCTURE
 UNALIGNED
 VARYING

CM40. `{asterisk-bounds} ::= ( {*-commalist} )`

CM41. `{generic-precision} ::= {number-of-digits} [ {scale-factor} ]`
 [{scale-factor}]

2.4.4.6 Initial

CM42. `{initial} ::= INITIAL ( { {initial-element-commalist} } )`
 CM43. `{initial-element} ::= * | {parenthesized-expression} | {iteration-factor} { {initial-constant-two} } | * | {initial-constant-one}`
 CM44. `{initial-constant-one} ::= {prefix-operator} {simple-string-constant} | {initial-constant-two}`
 CM45. `{initial-constant-two} ::= {prefix-operator} {reference} | {repeated-string-constant} | {arithmetic-constant} | {+ | -} {real-constant} {+ | -} {imaginary-constant}`
 CM46. `{iteration-factor} ::= {expression}`

2.4.8 THE DO STATEMENT

CM62. {do-statement} ::= DO; | DO {while-option}; | DO {do-spec};
 CM63. {do-spec} ::= {reference} = {spec-commalist}
 CM64. {spec} ::= {expression} {to-by} | {repeat-option} | {while-option};
 CM65. {to-by} ::= {to-option} {by-option} | {by-option} {to-option} {to-option};
 CM66. {to-option} ::= TO {expression}
 CM67. {by-option} ::= BY {expression}
 CM68. {while-option} ::= WHILE {expression}
 CM69. {repeat-option} ::= REPEAT {expression}

2.4.9 THE END STATEMENT

CM70. {end-statement} ::= END {identifier};

2.4.10 FLOW OF CONTROL STATEMENTS

2.4.10.1 The Call and Return Statements

CM71. {call-statement} ::= CALL {reference};
 CM72. {return-statement} ::= RETURN ({expression});

2.4.10.2 The Go To Statement

CM73. {goto-statement} ::= (GOTO | GO TO) {reference};

2.4.10.3 The Null Statement

CM74. {null-statement} ::= ;

2.4.10.4 The Revert and Signal Statements

CM75. {revert-statement} ::= REVERT {condition-name-commalist};
 CM76. {signal-statement} ::= SIGNAL {condition-name};
 CM77. {condition-name} ::= {computational-condition} | {named-io-condition} |
 {programmer-named-condition} | AREA | ERROR |
 FINISH | STORAGE
 CM78. {named-io-condition} ::= {io-condition} {reference};
 CM79. {io-condition} ::= ENDFILE | ENDPAGE | KEY | NAME | RECORD | TRANSMIT |
 UNDEFINEDFILE
 CM80. {programmer-named-condition} ::= CONDITION {identifier}

2.4.10.5 The Stop Statement

CM81. {stop-statement} ::= STOP;

.40

2.4.4.7 The Default Statement

CM47. {default-statement} ::= DEFAULT {default-specification} | NONE | SYSTEM;
 CM48. {default-specification} ::= {predicate-expression}
 {ERROR | {default-attributes-commalist}}
 CM49. {predicate-expression} ::= {predicate-expression-three} |
 {predicate-expression} {predicate-expression-three}
 CM50. {predicate-expression-three} ::= {predicate-expression-two} |
 {predicate-expression-three} &
 {predicate-expression-two}
 CM51. {predicate-expression-two} ::= {predicate-expression-one} |
 {predicate-expression-two}
 CM52. {predicate-expression-one} ::= {predicate-expression} | {attribute-keyword} |
 {range-specification}

CM53. {range-specification} ::= RANGE ({identifier} | {letter} : {letter} | *)

CM54. {attribute-keyword} ::= ALIGNED | AREA | DEFINED | DIMENSION | INTERNAL | KEYED | PRECISION |
 AUTOMATIC | BASED | DIRECT | LABEL | PRINT | REAL |
 BINARY | ENTRY | ENVIRONMENT | LOCAL | RECORD | RETURN |
 BIT | EXTERNAL | MEMBER | NONVARYING | SEQUENTIAL |
 BUILTIN | FILE | FIXED | OFFSET | STATIC |
 CHARACTER | FLOAT | OUTPUT | STREAM |
 COMPLEX | FORMAT | PARAMETER | STRUCTURE |
 CONSTANT | GENERIC | PICTURE | UNALIGNED |
 CONTROLLED | INITIAL | POINTER | UPDATE |
 DECIMAL | INPUT | POSITION | VARYING

CM55. {default-attributes} ::= {attribute-list}

2.4.5 THE PROCEDURE STATEMENT

CM56. {procedure-statement} ::= PROCEDURE {entry-information};
 CM57. {entry-information} ::= ({parameter-name-commalist})
 ({returns-descriptor}) • ({options}) • {RECURSIVE}

CM58. {parameter-name} ::= {identifier}

CM59. {returns-descriptor} ::= RETURNS ({description-commalist})

2.4.6 THE ENTRY STATEMENT

CM60. {entry-statement} ::= ENTRY {entry-information};

2.4.7 THE BEGIN STATEMENT

CM61. {begin-statement} ::= BEGIN {options};

2.4.13 EXPRESSIONS

- CM154. $\{expression\} ::= \{expression-seven\} \mid \{expression\} \{ \mid \} \{expression-seven\}$
 CM155. $\{expression-seven\} ::= \{expression-six\} \mid \{expression-seven\} \& \{expression-six\}$
 CM156. $\{expression-six\} ::= \{expression-five\} \mid \{expression-six\} \{comparison-operator\} \{expression-five\}$
 CM157. $\{comparison-operator\} ::= > \mid >= \mid = \mid < \mid <= \mid \rightarrow \mid \rightarrow= \mid \sim \mid \sim <$
 CM158. $\{expression-five\} ::= \{expression-four\} \mid \{expression-five\} \{ \mid \} \{expression-four\}$
 CM159. $\{expression-four\} ::= \{expression-three\} \mid \{expression-four\} \{ + \mid - \} \{expression-three\}$
 CM160. $\{expression-three\} ::= \{expression-two\} \mid \{expression-three\} \{ * \mid / \} \{expression-two\}$
 CM161. $\{expression-two\} ::= \{primitive-expression\} \mid \{prefix-expression\} \mid \{parenthesized-expression\} \mid \{expression-one\}$
 CM162. $\{expression-one\} ::= \{primitive-expression\} \mid \{parenthesized-expression\} \mid \{parenthesized-expression\}$
 CM163. $\{prefix-expression\} ::= \{prefix-operator\} \{expression-two\}$
 CM164. $\{prefix-operator\} ::= + \mid - \mid \sim$
 CM165. $\{parenthesized-expression\} ::= \{expression\}$
 CM166. $\{primitive-expression\} ::= \{reference\} \mid \{constant\} \mid \{isub\}$
 CM167. $\{reference\} ::= \{locator-qualifier\} \{basic-reference\} \{arguments-list\}$
 CM168. $\{locator-qualifier\} ::= \{reference\} \rightarrow$
 CM169. $\{arguments\} ::= \{ \{subscript-commalist\} \}$
 CM170. $\{basic-reference\} ::= \{structure-qualification\} \{identifier\}$
 CM171. $\{structure-qualification\} ::= \{basic-reference\} \{arguments\}$
 CM172. $\{subscript\} ::= \{expression\} \mid *$
 CM173. $\{unsubscripted-reference\} ::= \{unsubscripted-reference\} \mid \{identifier\}$
 CM174. $\{constant\} ::= \{arithmetic-constant\} \mid \{string-constant\}$
 CM175. $\{string-constant\} ::= \{simple-string-constant\} \mid \{replicated-string-constant\}$
 CM176. $\{simple-string-constant\} ::= \{simple-character-string-constant\} \mid \{simple-bit-string-constant\}$
 CM177. $\{replicated-string-constant\} ::= \{integer\} \{simple-string-constant\}$

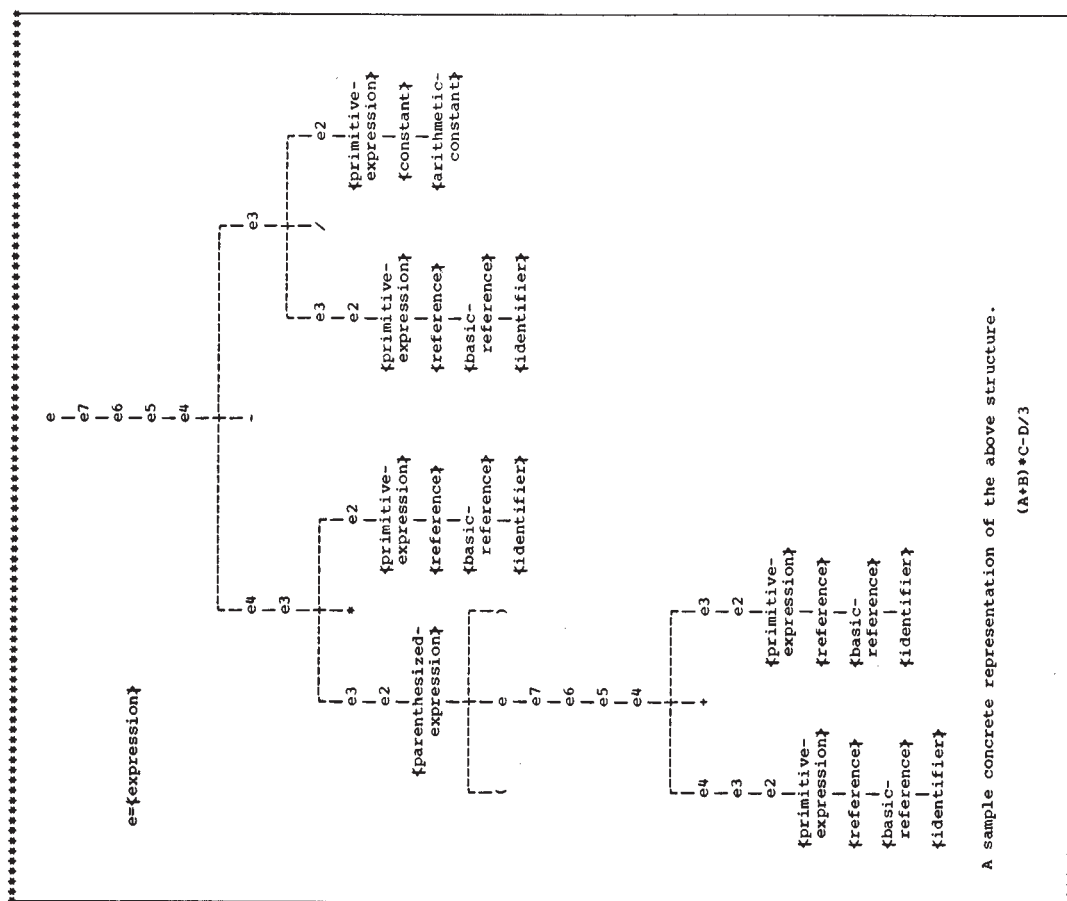
44

2.4.12.3.3 Stream Output Specification

- CM126. $\{output-specification\} ::= \{data-directed-output\} \mid \{list-directed-output\} \mid \{edit-directed-output\}$
 CM127. $\{data-directed-output\} ::= DATA \{ \{data-source-commalist\} \}$
 CM128. $\{data-source\} ::= \{reference\} \mid \{data-source-commalist\} DO \{do-spec\}$
 CM129. $\{list-directed-output\} ::= LIST \{ \{output-source-commalist\} \}$
 CM130. $\{output-source\} ::= \{expression\} \mid \{output-source-commalist\} DO \{do-spec\}$
 CM131. $\{edit-directed-output\} ::= EDIT \{ \{edit-output-pair-list\} \}$
 CM132. $\{edit-output-pair\} ::= \{ \{output-source-commalist\} \} \{ \{format-specification-commalist\} \}$
 2.4.12.3.3 Format Specification Lists and the Format Statement
- CM133. $\{format-specification\} ::= \{format-item\} \mid \{format-iteration\}$
 CM134. $\{format-iteration\} ::= \{ \{format-iteration-factor\} \} \{ \{format-item\} \} \{ \{format-specification-commalist\} \}$
 CM135. $\{format-iteration-factor\} ::= \{integer\} \mid \{expression\}$
 CM136. $\{format-item\} ::= \{data-format\} \mid \{control-format\} \mid \{remote-format\}$
 CM137. $\{data-format\} ::= \{real-format\} \mid \{complex-format\} \mid \{picture-format\} \mid \{string-format\}$
 CM138. $\{real-format\} ::= \{fixed-point-format\} \mid \{floating-point-format\}$
 CM139. $\{fixed-point-format\} ::= F \{ \{expression\} \} \{ \{expression\} \}$
 CM140. $\{floating-point-format\} ::= E \{ \{expression\} \} \{ \{expression\} \}$
 CM141. $\{complex-format\} ::= C \{ \{real-format\} \} \{ \{picture-format\} \} \{ \{real-format\} \} \{ \{picture-format\} \}$
 CM142. $\{picture-format\} ::= P \{ \{picture\} \}$
 CM143. $\{string-format\} ::= \{character-format\} \mid \{bit-format\}$
 CM144. $\{character-format\} ::= A \{ \{expression\} \}$
 CM145. $\{bit-format\} ::= \{radix-factor\} \{ \{expression\} \}$
 CM146. $\{control-format\} ::= \{tab-format\} \mid \{line-format\} \mid \{space-format\} \mid \{skip-format\} \mid \{column-format\} \mid PAGE$
 CM147. $\{tab-format\} ::= TAB \{ \{expression\} \}$
 CM148. $\{line-format\} ::= LINE \{ \{expression\} \}$
 CM149. $\{space-format\} ::= X \{ \{expression\} \}$
 CM150. $\{skip-format\} ::= SKIP \{ \{expression\} \}$
 CM151. $\{column-format\} ::= COLUMN \{ \{expression\} \}$
 CM152. $\{remote-format\} ::= R \{ \{reference\} \}$
 CM153. $\{format-statement\} ::= FORMAT \{ \{format-specification-commalist\} \}$

Chapter 2: Concrete Syntax 43

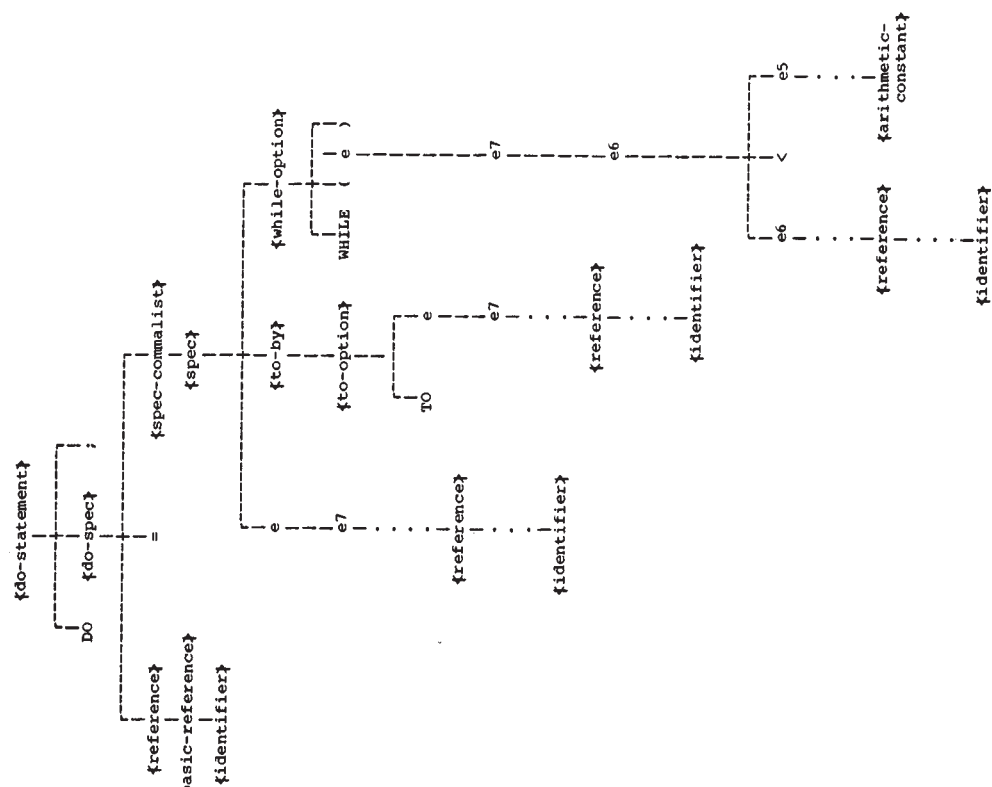
The following are examples of two middle-level parses. As in the previous examples, each is accompanied by an example of a construct that matches the given syntax.



Example 2.3. An Example of the Middle-level Structure of an `{expression}`.

Chapter 2: Concrete Syntax 45

94



A sample concrete representation of the above structure:

```
DO I=K TO J WHILE(A<0);
```

Example 2.4. An Example of the Middle-level Structure of a {do-statement}.

2.5 The Low-level Syntax of PL/I

2.5.1 PL/I TEXT

```

CL1.  {pli-text} ::= {delimiter-list} { {delimiter-pair-list} }
CL2.  {delimiter-pair} ::= {non-delimiter} {delimiter-list}
CL3.  {delimiter} ::= + | - | * | / | ** | > | < | = | <= | >= | ~< | ~> | ~| ~| {comment} |
      & | { } | { | } | ( | ) | . | , | ; | : | ~ | ~| {comment} |
      {include}
CL4.  {non-delimiter} ::= {identifier} | {arithmetic-constant} |
      {simple-bit-string-constant} |
      {simple-character-string-constant} | {sub}

```

2.5.2 COMMENT

```

CL5.  {comment} ::= /* {comment-body-list} {*-list} /
CL6.  {comment-body} ::= {*-list} {comment-character} | /
CL7.  {comment-character} ::= {letter} | {digit} | . | ' | ' | = | + | - | ( | ) | , |
      | $ | ; | : | & | { } | ~ | > | < | % |
      /extralingual-character/

```

Note: Rules CL5-7 effectively state that a comment begins with /* and ends with */ and that any characters may appear between these except the consecutive pair */.

2.5.3 IDENTIFIER

```

CL8.  {identifier} ::= {letter} | {identifier} {letter} | {digit} | _
CL9.  {letter} ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
CL10. {digit} ::= 0|1|2|3|4|5|6|7|8|9

```

2.5.4 ARITHMETIC CONSTANT

```

CL11. {arithmetic-constant} ::= {real-constant} | {imaginary-constant}
CL12. {real-constant} ::= {decimal-constant} | {binary-constant}
CL13. {decimal-constant} ::= {decimal-number} {scale-type} {exponent} | {P}
CL14. {decimal-number} ::= {integer} ( . {digit-list} ) | . {digit-list}
CL15. {integer} ::= {digit-list}
CL16. {scale-type} ::= E | F
CL17. {exponent} ::= [ + | - ] {integer}
CL18. {binary-constant} ::= {binary-number} {scale-type} {exponent} | B {P}
CL19. {binary-number} ::= {binary-digit-list} ( . {binary-digit-list} ) |
      . {binary-digit-list}
CL20. {binary-digit} ::= 0 | 1
CL21. {imaginary-constant} ::= {real-constant} I

```

2.5.5 STRING CONSTANTS AND PICTURES

```

CL22. {simple-bit-string-constant} ::= '{string-or-picture-symbol-list}' {radix-factor}
CL23. {radix-factor} ::= B|B1|B2|B3|B4
CL24. {simple-character-string-constant} ::= '{string-or-picture-symbol-list}'
CL25. {string-or-picture-symbol} ::= {letter} | {digit} | . | ' | ' | = | + | - | * |
      / | ( | ) | , | ; | : | & | { } | ~ | > |
      < | % | {extralingual-character}

```

Note: A {string-or-picture-symbol} may be two consecutive characters ' ' or any character other than '.

```

CL26. {extralingual-character} ::=

```

The category {extralingual-character} in rule CL26 is implementation-defined, as specified in Section 2.6.2.

2.5.6 ISUB

```

CL27. {isub} ::= {integer} SUB

```

2.5.7 INCLUDE

```

CL28. {include} ::= %INCLUDE {text-name};

```

```

CL29. {text-name} ::=

```

The category {text-name} in rule CL29 is implementation-defined and such that if it contains {;}, then that {;} must be contained in a {simple-character-string-constant} which is the only immediate subnode of {text-name}.

2.6 Character Sets

The character set used in the formation of PL/I text is a finite set of symbols, comprising 57 language characters, and zero or more {extralingual-characters} which are distinct from these and from each other and are implementation-defined.

```

{symbol} ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
S | T | U | V | W | X | Y | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
. | ' | ' | = | + | - | * | / | ( | ) | , | ; | : | & | { } |
~ | < | > | % | {extralingual-character}

```

This document does not specify internal or external hardware representations of the characters, nor does it define a collating sequence for them. These may, however, be the subject of other standards.

2.6.1 LANGUAGE CHARACTER SET

The names of the symbols in the language character set, together with the graphic representations of them to be used in this document, are given in Sections 2.6.1.1 and 2.6.1.2.

2.7 Abbreviations

Abbreviations are provided for certain keywords (see Section 4.2.2) and builtin-function-names. The abbreviations will be recognized as synonymous in every respect with the full denotations, except that in the case of builtin-function-names the abbreviations have separate declarations (explicit or contextual) and name scopes. The abbreviations are shown to the right of the full denotations in the following list.

ALLOCATE	ALLOC	NOOVERFLOW	NOOFL
ALLOCATION	ALLOCN	NOSTRINGRANGE	NOSTRG
AUTOMATIC	AUTO	NOSTRINGSIZE	NOSTRZ
BINARY	BIN	NOSUBSCRIPTRANGE	NOSUBRG
CHARACTER	CHAR	NOUNDERFLOW	NOUNFL
COLUMN	COL	NOZERODIVIDE	NOZDIV
COMPLEX	CPLX	OVERFLOW	OFL
CONDITION	COND	PARAMETER	PARM
CONTROLLED	CTL	PICTURE	PIC
CONVERSION	CONV	POSITION	POS
DECIMAL	DECL	PRECISION	PREC
DECLARE	DCL	PROCEDURE	PROC
DEFAULT	DEF	SEQUENTIAL	SEQ
DEFINED	DEF	STRINGRANGE	STRG
DIMENSION	DIM	STRINGSIZE	STRZ
ENVIRONMENT	ENV	SUBSCRIPTRANGE	SUBRG
EXTERNAL	EXT	UNALIGNED	UNAL
FIXEDOVERFLOW	FOFL	UNDERFLOW	UFL
INITIAL	INIT	UNDEFINEDFILE	UNDF
INTERNAL	INT	VARYING	VAR
NOCONVERSION	NOCONV	ZERODIVIDE	ZDIV
NOFIXEDOVERFLOW	NOFOFL		
NONVARYING	NONVAR		

2.6.1.1 Letters and Digits

Name	Graphic
Letter A	A
Letter B	B
Letter C	C
Letter D	D
Letter E	E
Letter F	F
Letter G	G
Letter H	H
Letter I	I
Letter J	J
Letter K	K
Letter L	L
Letter M	M
Letter N	N
Letter O	O
Letter P	P
Letter Q	Q
Letter R	R
Letter S	S
Letter T	T
Letter U	U
Letter V	V
Letter W	W
Letter X	X
Letter Y	Y
Letter Z	Z

Name	Graphic
Digit 0	0
Digit 1	1
Digit 2	2
Digit 3	3
Digit 4	4
Digit 5	5
Digit 6	6
Digit 7	7
Digit 8	8
Digit 9	9

2.6.1.2 Special Characters

Name	Graphic
Plus	+
Minus	-
Asterisk	*
Slash	/
Greater than	>
Less than	<
Equal	=
Not	~
And	&
Or	
Percent	%

Name	Graphic
Period	.
Comma	,
Semicolon	;
Colon	:
Blank	
Single quote	'
Left parenthesis	(
Right parenthesis	)
Break	^
Dollar	\$

2.6.2 DATA CHARACTER SET

Data in stream-datasets or in character-string-values may be represented by characters from the language character set plus any other characters permitted by the particular implementation's fextralingual-character_s.

Chapter 3: Abstract Syntax

3.0 Introduction

This chapter specifies the Abstract Syntax of PL/I, which is the syntax of programs in a tree representation which is convenient for the definition of semantics. The notation for this syntax is defined in Chapter 1, together with some examples of its use. Further examples are provided at the end of this chapter.

Many parts of the Abstract Syntax bear a strong resemblance to the corresponding parts of the Concrete Syntax, and the relationship between them is intended to be, or become, intuitively obvious. In other parts, notably in the treatment of <declarations>, the Abstract Syntax exhibits a structuring and completeness of information which involves a more complex transformation. The detailed description of the transformation between the concrete and abstract forms of a program will be given in the next chapter.

There are also many instances of context-dependent constraints which have been inserted in parentheses in the Abstract Syntax. These are attached to categories such as <expression>, <target-reference>, <value-reference> and <declaration-designator>, where these are required to fall within the scope of appropriate <declarations> so that they have the properties indicated. The Translator (Chapter 4) checks that these constraints are satisfied.

3.1 Abstract Syntax Rules

3.1.1 PROGRAM

- A1. <program> ::= [**abstract-external-procedure-list**]
 A2. <abstract-external-procedure> ::= [**declaration-list**] <procedure>

3.1.2 PROCEDURE

- A3. <procedure> ::= [**declaration-list**] [**procedure-list**] [**format-statement-list**]
 [**condition-prefix-list**] [**recursive**]
 <entry-or-executable-unit-list>

3.1.3 DECLARATION

- A4. <declaration> ::= <identifier> <scope> <declaration-type> {**declaration-designator**}
 A5. <scope> ::= <external> | <internal>
 A6. <declaration-type> ::= <variable> | <named-constant> | <builtin> | <condition>

3.1.4 VARIABLE

- A7. <variable> ::= <storage-type> <data-description>
 A8. <storage-type> ::= <storage-class> | <defined> | <parameter>
 A9. <storage-class> ::= <automatic> | <based> | <controlled> | <static>
 A10. <based> ::= [**value-reference**] (scalar & locator) | {**reference-designator**}
 A11. <defined> ::= <base-item> {<position>}
 A12. <base-item> ::= <variable-reference> (-defined & -based) |
 {**reference-designator**}
 A13. <position> ::= <expression> (scalar & computational-type) | {**expression-designator**}

3.1.5 DATA-DESCRIPTION

- A14. <data-description> ::= <dimensioned-data-description> |
 <structure-data-description> | <item-data-description>
 A15. <dimensioned-data-description> ::= <element-data-description> <bound-pair-list>
 A16. <element-data-description> ::= <structure-data-description> |
 <item-data-description>
 A17. <bound-pair> ::= <lower-bound> <upper-bound> | <asterisk>
 A18. <lower-bound> ::= <extent-expression>
 A19. <upper-bound> ::= <extent-expression>
 A20. <extent-expression> ::= {<expression> (scalar & computational-type) |
 {**expression-designator**} | <integer-value>}} {<refer-option>}
 A21. <refer-option> ::= <identifier-list> | <integer-value>
 A22. <structure-data-description> ::= {<identifier-list>} <member-description-list>
 A23. <member-description> ::= <data-description>
 A24. <item-data-description> ::= [**alignment**] <data-type> {<initial>}

Constraint: An <item-data-description> must not have an <initial> component with an <iteration-factor>, or an <initial-element-list> with more than one <initial-element> immediate component, unless the <item-data-description> is a (not necessarily immediate) component of a <dimensioned-data-description>.

- A25. <initial> ::= <initial-element-list> | {&b>initial-designator}
 A26. <initial-element> ::= <asterisk> | <parenthesized-expression> (scalar) |
 <iteration-factor> <initial-element-list>
 A27. <iteration-factor> ::= <expression> (scalar & computational-type)
 A28. <alignment> ::= <aligned> | <unaligned>

3.1.6 DATA-TYPE

- A29. <data-type>::= <computational-type> | <non-computational-type>
- A30. <computational-type>::= <arithmetic> | <string> | <picture>
- A31. <non-computational-type>::= <area> | <entry> | <file> | <format> | <local> | <label> | <locator>
- A32. <arithmetic>::= <mode> <base> <scale> <precision>
- A33. <mode>::= <real> | <complex>
- A34. <base>::= <binary> | <decimal>
- A35. <scale>::= <fixed> | <float>
- A36. <precision>::= <number-of-digits> [<scale-factor>]
- A37. <number-of-digits>::= <integer>
- Constraint: The <integer> must not be zero.
- A38. <scale-factor>::= <signed-integer>
- A39. <string>::= <string-type> <maximum-length> [<varying> | <nonvarying>]
- A40. <string-type>::= <character> | <bit>
- A41. <maximum-length>::= <extent-expression> | <asterisk>
- A42. <picture>::= <picture-character> | <picture-numeric>
- A43. <locator>::= <pointer> | <offset>
- A44. <offset>::= [<variable-reference> (scalar & area) | {reference-designator}]
- A45. <entry>::= [<parameter-descriptor-list>] [<returns-descriptor>] [<options>]
- A46. <parameter-descriptor>::= <data-description>
- A47. <returns-descriptor>::= <data-description>
- A48. <options>::=
- This category is implementation-defined.
- A49. <area>::= <area-size>
- A50. <area-size>::= <extent-expression> | <asterisk>

3.1.7 NAMED-CONSTANT

- A51. <named-constant>::= { <entry> | <file> <file-description> | <format> | <label> } [<bound-pair-list>]
- A52. <file-description>::= [<stream>] [<record>] [<input>] [<output>] [<update>] [<sequential>] [<direct>] [<print>] [<keyed>] [<environment>]
- A53. <environment>::=

This category is implementation-defined.

3.1.8 ENTRY-OR-EXECUTABLE-UNIT

- A54. <entry-or-executable-unit>::= <entry-point> | <executable-unit>
- A55. <entry-point>::= [<statement-name>] <entry-information>
- A56. <statement-name>::= <identifier> [<signed-integer-list>]
- A57. <entry-information>::= [<parameter-name-list>] [<returns-descriptor>] [<options>]
- A58. <parameter-name>::= <identifier>
- A59. <executable-unit>::= [<condition-prefix-list>] [<statement-name-list>] [<begin-block>]
- | | | |
|------------------------|--|---------------------|
| <null-statement> | | <group> |
| <assignment-statement> | | <on-statement> |
| <call-statement> | | <open-statement> |
| <close-statement> | | <put-statement> |
| <delete-statement> | | <read-statement> |
| <end-statement> | | <return-statement> |
| <free-statement> | | <rewind-statement> |
| <goto-statement> | | <rewrite-statement> |
| <if-statement> | | <signal-statement> |
| <locate-statement> | | <stop-statement> |
| | | <write-statement> |

3.1.9 BEGIN-BLOCK

- A60. <begin-block>::= [<declaration-list>] [<procedure-list>] [<format-statement-list>] [<options>] <executable-unit-list>

3.1.10 GROUPS

- A61. <group>::= <iterative-group> | <non-iterative-group>
- A62. <iterative-group>::= <controlled-group> | <while-only-group>
- A63. <controlled-group>::= <do-spec> <executable-unit-list>
- A64. <do-spec>::= <target-reference> (scalar) <spec-list>
- A65. <spec>::= <expression> (scalar) [<to-by> | <repeat-option>] [<while-option>]
- A66. <to-by>::= <to-option> [<by-option>] | <by-option>
- A67. <repeat-option>::= <expression> (scalar)
- A68. <by-option>::= <expression> (scalar & computational-type)
- A69. <to-option>::= <expression> (scalar & computational-type)
- A70. <while-option>::= <expression> (scalar & computational-type)
- A71. <while-only-group>::= <while-option> <executable-unit-list>
- A72. <non-iterative-group>::= <entry-or-executable-unit-list>

3.1.11 ON STATEMENT

- A73. <on-statement>::= <condition-name-list> [<snap>] { <on-unit> | <system-action> }
- A74. <on-unit>::= <procedure>

3.1.12 IF STATEMENT

A75. <if-statement> ::= <test> <then-unit> [<else-unit>]
 A76. <test> ::= <expression> (scalar & computational-type)
 A77. <then-unit> ::= <executable-unit>
 A78. <else-unit> ::= <executable-unit>

3.1.13 FLOW OF CONTROL STATEMENTS

A79. <call-statement> ::= <subroutine-reference>
 A80. <goto-statement> ::= <value-reference> (scalar & label)
 A81. <return-statement> ::= [<expression>]
 A82. <revert-statement> ::= <condition-name-list>
 A83. <signal-statement> ::= <condition-name>
 A84. <condition-name> ::= <computational-condition> | <named-io-condition> |
 <programmer-named-condition> | <area-condition> |
 <error-condition> | <finish-condition> | <storage-condition>
 A85. <condition-prefix> ::= <computational-condition> (<enabled> | <disabled>)
 A86. <computational-condition> ::= <conversion-condition> | <fixedoverflow-condition> |
 <overflow-condition> | <size-condition> |
 <stringrange-condition> | <stringsize-condition> |
 <subscriptrange-condition> | <underflow-condition> |
 <zerodivide-condition>
 A87. <named-io-condition> ::= <io-condition> <value-reference> (scalar & file)
 A88. <io-condition> ::= <endfile-condition> | <endpage-condition> | <key-condition> |
 <name-condition> | <record-condition> | <transmit-condition> |
 <undefinedfile-condition>
 A89. <programmer-named-condition> ::= <declaration-designator> (condition)

3.1.14 STORAGE STATEMENTS

A90. <assignment-statement> ::= <target-reference-list> <expression>
 A91. <allocate-statement> ::= <allocation-list>
 A92. <allocation> ::= <declaration-designator> (based | controlled)
 [<set-option>] [<in-option>]
 A93. <set-option> ::= <variable-reference> (scalar & locator)

Constraint: The <data-description> immediate component of the <variable-reference> must not have <offset> without a <variable-reference> subnode.

A94. <in-option> ::= <variable-reference> (scalar & area)
 A95. <free-statement> ::= <freeing-list>
 A96. <freeing> ::= [<locator-qualifier>] <declaration-designator> (based | controlled)
 [<in-option>]

3.1.15 I/O STATEMENTS

A97. <open-statement> ::= <single-opening-list>
 A98. <single-opening> ::= <file-option> [<tab-option>] [<title-option>]
 [<linesize-option>] [<pagesize-option>] [<stream>] [<record>]
 [<input>] [<output>] [<update>] [<sequential>] [<direct>]
 [<print>] [<keyed>] [<environment>]
 A99. <file-option> ::= <value-reference> (scalar & file)
 A100. <tab-option> ::= <expression-list> (scalar & computational-type)
 A101. <title-option> ::= <expression> (scalar & computational-type)
 A102. <linesize-option> ::= <expression> (scalar & computational-type)
 A103. <pagesize-option> ::= <expression> (scalar & computational-type)
 A104. <close-statement> ::= <single-closing-list>
 A105. <single-closing> ::= <file-option> [<environment>]

3.1.16 RECORD I/O STATEMENTS

A106. <delete-statement> ::= <file-option> [<key-option>] [<key-option>]
 A107. <locate-statement> ::= <declaration-designator> (based) <file-option>
 [<pointer-set-option>] [<keyfrom-option>]
 A108. <pointer-set-option> ::= <variable-reference> (scalar & pointer)
 A109. <read-statement> ::= <file-option> | <pointer-set-option> | <ignore-option>
 [<into-option>] | <key-option> | <keyto-option>
 A110. <into-option> ::= <variable-reference>
 A111. <ignore-option> ::= <expression> (scalar & computational-type)
 A112. <rewrite-statement> ::= <file-option> | <from-option>
 [<key-option>] [<keyfrom-option>]
 A113. <write-statement> ::= <file-option> <from-option>
 [<keyfrom-option>]
 A114. <from-option> ::= <variable-reference>
 A115. <key-option> ::= <expression> (scalar & computational-type)
 A116. <keyfrom-option> ::= <expression> (scalar & computational-type)
 A117. <keyto-option> ::= <target-reference> (scalar & character)

3.1.17 STREAM I/O STATEMENTS

A118. <get-statement> ::= <get-file> | <get-string>
 A119. <get-file> ::= <file-option> [<copy-option>] [<skip-option>] [<input-specification>]
 Constraint: At least one of the last two options must be present.
 A120. <skip-option> ::= <expression> (scalar & computational-type)
 A121. <copy-option> ::= <value-reference> (scalar & file)

A122. <get-string> ::= <expression> {scalar & computational-type}
<input-specification> {<copy-option>}

A123. <put-statement> ::= <put-file> | <put-string>

A124. <put-file> ::= <file-option> {<skip-option>} {<line-option>} {<page>}
{<output-specification>}

Constraint: At least one of the last four options must be present and the <skip-option> must not be used together with a <line-option> or <page>.

A125. <line-option> ::= <expression> {scalar & computational-type}

A126. <put-string> ::= <target-reference> {scalar & character} <output-specification>

A127. <input-specification> ::= <data-directed-input> | <list-directed-input> |
<edit-directed-input>

A128. <data-directed-input> ::= {<data-target-list>}

A129. <data-target> ::= <variable-reference> {computational-type}

Constraint: The <variable-reference> must not contain a <locator-qualifier> or a <subscript-list>, and must not contain a <by-name-parts-list>.

A130. <list-directed-input> ::= <input-target-list>

A131. <input-target> ::= <target-reference> {computational-type} |
<input-target-list> <do-spec>

A132. <edit-directed-input> ::= <edit-input-pair-list>

A133. <edit-input-pair> ::= <input-target-list> <format-specification-list>

A134. <output-specification> ::= <data-directed-output> | <list-directed-output> |
<edit-directed-output>

A135. <data-directed-output> ::= {<data-source-list>}

A136. <data-source> ::= <variable-reference> {computational-type} |
<data-source-list> <do-spec>

Constraint: The <variable-reference> must not have a <locator-qualifier>.

A137. <list-directed-output> ::= <output-source-list>

A138. <output-source> ::= <expression> {computational-type} |
<output-source-list> <do-spec>

A139. <edit-directed-output> ::= <edit-output-pair-list>

A140. <edit-output-pair> ::= <output-source-list> <format-specification-list>

A141. <format-specification> ::= <format-item> | <format-iteration>

A142. <format-iteration> ::= <format-iteration-factor> <format-specification-list>

A143. <format-iteration-factor> ::= <expression> {scalar}

A144. <format-item> ::= <data-format> | <control-format> | <remote-format>

A145. <data-format> ::= <real-format> | <complex-format> | <picture-format> |
<string-format>

A146. <floating-point-format> ::= <expression> {scalar & computational-type} |
<integer-value>
{<expression> {scalar & computational-type} |
<integer-value>}

A147. <complex-format> ::= {<real-format> | <picture-format>}
{<real-format> | <picture-format> | <picture-format>}

Constraint: A <complex-format> must not contain <picture-character>.

A148. <picture-format> ::= <real-format> | <picture-format> |
<real-format> | <picture-format> |
<integer-value> |
{<expression> {scalar & computational-type} |
<integer-value>}

A149. <character-format> ::= {<expression> {scalar & computational-type} |
<integer-value>}

A150. <bit-format> ::= <radix-factor> {<expression> {scalar & computational-type} |
<integer-value>}

A151. <string-format> ::= <character-format> | <bit-format>

A152. <control-format> ::= {<expression> {scalar & computational-type} |
<integer-value>}

A153. <tab-format> ::= <tab-factor> | <line-format> | <space-format> | <skip-format> |
<column-format> | <page>

A154. <radix-factor> ::= 1 | 2 | 3 | 4

A155. <tab-format> ::= <expression> {scalar & computational-type} | <integer-value>

A156. <line-format> ::= <expression> {scalar & computational-type} | <integer-value>

A157. <space-format> ::= <expression> {scalar & computational-type} | <integer-value>

A158. <skip-format> ::= <expression> {scalar & computational-type} | <integer-value>

A159. <column-format> ::= <expression> {scalar & computational-type} | <integer-value>

A160. <remote-format> ::= <expression> {scalar & computational-type} | <integer-value>

A161. <variable-reference> ::= <variable-reference> {scalar & format} |
<named-constant-reference> {scalar & format}

A162. <format-statement> ::= {<condition-prefix-list> |
<statement-name-list> <format-specification-list>}

3.1.18 EXPRESSION

A163. <expression> ::= {<value-reference> | <constant> | <isub> |
<infix-expression> | <prefix-expression> |
<parenthesized-expression>} <data-description>

A164. <infix-expression> ::= <expression> <infix-operator> <expression> <data-description>

A165. <infix-operator> ::= <or> | <and> | <gt> | <ge> | <eq> | <le> | <lt> | <ne> |
<cat> | <add> | <subtract> | <multiply> | <divide> | <power>

A166. <prefix-expression> ::= <prefix-operator> <expression> <data-description>

A167. <prefix-operator> ::= <plus> | <minus> | <not>

A168. <parenthesized-expression> ::= <expression> <data-description>

4.0 Introduction

This Chapter defines an operation which translates a {symbol-list} into an {abstract-external-procedure}. All the {abstract-external-procedure}s that are part of a PL/I {program} are then combined. The {program} is used by the PL/I machine to determine the course of execution.

4.1 Translate

A concrete-block is a {begin-block}, {procedure}, or {concrete-external-procedure}.
x is a concrete-block-component of y if y is a concrete-block, and y contains x but does not contain any other concrete-block which also contains x. In this case y concrete-block-contains x.
An abstract-block is a {begin-block}, {procedure}, or {abstract-external-procedure}.
x is an abstract-block-component of y if y is an abstract-block, and y contains x but does not contain any other abstract-block which also contains x. In this case y abstract-block-contains x.

The informal term block-component is used for either concrete-block-component or abstract-block-component, and block-contains is used for either concrete-block-contains or abstract-block-contains where the context makes it obvious which formal term is required.

A {declaration}, {description}, {default-attributes} or {generic-description}, d declaration-contains a node n, if d contains n and d does not contain a {description} or {generic-description} that also contains n.

A node, n, is a declaration-component of a node, d, if d declaration-contains n.

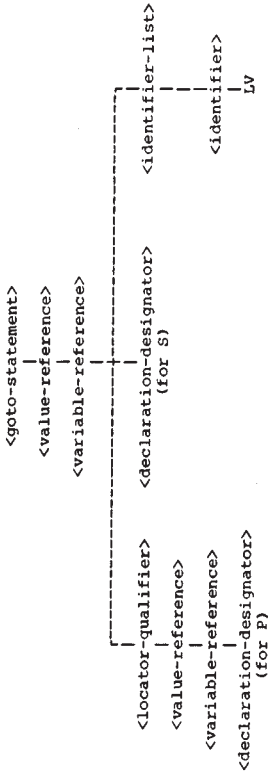
Operation: translate(t)
where t is a {symbol-list}.
result: an {abstract-external-procedure}.

- Step 1. Perform parse(t,{procedure}) to obtain a {procedure}.cep. Append a {concrete-external-procedure}: cep; to the {translation-state}.
- Step 2. Perform complete-concrete-procedure.
- Step 3. Let aep be an {abstract-external-procedure}.
- Step 4. For each {declaration},d which is a block-component of the {concrete-external-procedure} perform create-declaration(d) to obtain a {declaration}.ad, and append ad to the {declaration-list} in aep.
- Step 5. For each {declaration},d which is a block-component of aep and which contains at least one {expression-designator} or {reference-designator}, perform replace-concrete-designators(d).
- Step 6. Let p be the {procedure} immediate component of the {concrete-external-procedure}. Perform create-procedure(p) to obtain a {procedure}.ap, and attach ap to aep.
- Step 7. Delete the {concrete-external-procedure}.
- Step 8. Perform validate-procedure(aep).
- Step 9. Return aep.

The abstract text corresponding to the concrete representation

GO TO P -> S.LV;

is as follows:

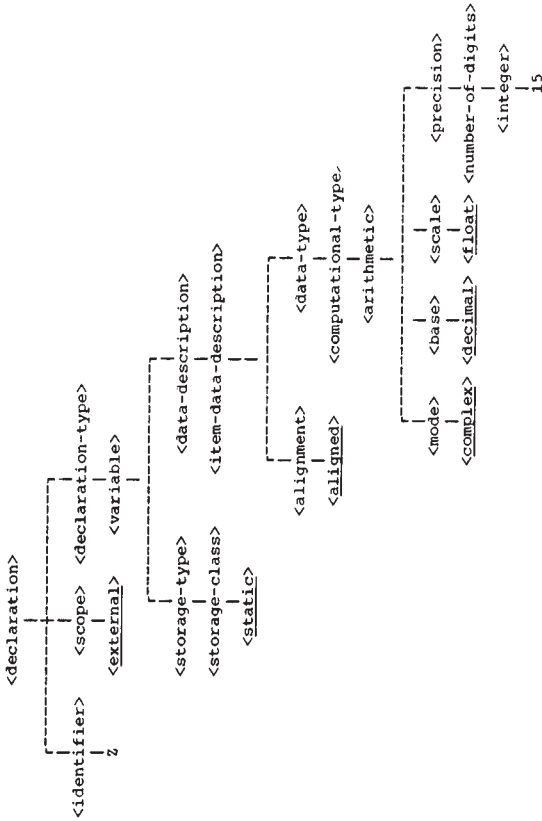


Note: {value-reference} and {variable-reference} nodes immediately contain a {data-description} (not shown).

The abstract text corresponding to the concrete-representation

DECLARE Z EXTERNAL STATIC ALIGNED COMPLEX DECIMAL FLOAT(15);

is as follows:



Example 3.1. Examples of Abstract Text.

4.2 Forming the Concrete Procedure

The parse operation is applied to a $\{\text{symbol-list}\}$ to construct a complete tree with respect to the Concrete Syntax for a specified category-name. If this category-name is defined in the high-level syntax or the middle-level syntax then some additional mapping at the interfaces between these syntaxes is required.

If parse is called for a $\{\text{procedure}\}$, it calls itself recursively to build trees consistent with the low-level, middle-level, and high-level syntaxes, in that order.

Operation: $\text{parse}(\text{sl}, n)$

where sl is a $\{\text{symbol-list}\}$,
 n is a tree with a single node, whose type is a non-terminal category in the Concrete Syntax.

result: a complete tree with respect to the Concrete Syntax for n .

Case 1. The type of n is a non-terminal of the high-level syntax.

Perform $\text{parse}(\text{sl}, \{\text{sentence-list}\})$ to obtain a $\{\text{sentence-list}\}$, snl . Perform high-level-parse(snl, n) to obtain nt .

Return nt .

Case 2. The type of n is a non-terminal of the middle-level syntax.

Perform $\text{parse}(\text{sl}, \{\text{pli-text}\})$ to obtain a $\{\text{pli-text}\}$, pt . Perform middle-level-parse(pt, n) to obtain nt .

Return nt .

Case 3. The type of n is a non-terminal of the low-level syntax.

Perform low-level-parse(sl, n) to obtain nt .

Return nt .

4.2.1 LOW-LEVEL-PARSE

Operation: $\text{low-level-parse}(\text{sl}, n)$

where sl is a $\{\text{symbol-list}\}$,
 n is a tree with a single node, whose type is a non-terminal category-name at the low-level syntax.

result: a complete tree with respect to the low-level syntax for n .

Step 1. There must exist one and only one tree, nt , which is a complete tree with respect to the low-level syntax for n , such that the following conditions are true:

- (1) the concrete-representation of nt is exactly the same as the concrete-representation of sl , and
- (2) every occurrence of $\{/**\}$ or $\{/**\}$ in the concrete-representation of nt must be such that the $\{/**\}$ and $\{/**\}$ are nodes of a $\{\text{comment}\}$ category or are contained in a $\{\text{non-delimiter}\}$, and
- (3) of all possible trees satisfying conditions (1) and (2), nt is that one containing the least number of $\{\text{delimiter-pairs}\}$ and $\{\text{delimiters}\}$.

Step 2. Return nt .

4.2.2 MIDDLE-LEVEL-PARSE

A keyword is a category-name specified in the middle-level syntax as a sequence of uppercase letters.

$\{\text{delimiter-or-non-delimiter}\} ::= \{\text{delimiter}\} \mid \{\text{non-delimiter}\}$

Operation: $\text{middle-level-parse}(\text{pt}, \text{cn})$

where pt is a $\{\text{pli-text}\}$,
 cn is a tree with a single node, whose type is a non-terminal of the middle-level syntax.

result: a complete tree with respect to the syntax composed of all the production-rules occurring in the middle-level syntax and in the low-level syntax with the root-node cn .

Step 1. Let t be a $\{\text{delimiter-or-non-delimiter-list}\}$ which contains a copy of the $\{\text{delimiter}\}$ and $\{\text{non-delimiter}\}$ components of pt in the same order.

Step 2. Repeat Steps 2.1 through 2.5 as long as there is a $\{\text{delimiter-or-non-delimiter}\}$, d : $\{\text{delimiter}\} ::= \{\text{include}\};;$ in t .

Step 2.1. Let s be $\{\text{symbol-list}\}$: $\{\text{symbol}\}$: $\emptyset$.

Step 2.2. Append to s any $\{\text{symbols}\}$ obtained in an implementation-defined way from the $\{\text{text-name}\}$ in d . Append $\{\text{symbol}\}$: $\emptyset$ to s .

Step 2.3. Perform low-level-parse($s, \{\text{pli-text}\}$) to obtain a $\{\text{pli-text}\}$, tx .

Step 2.4. Let t_1 be a $\{\text{delimiter-or-non-delimiter-list}\}$ which contains a copy of the $\{\text{delimiter}\}$ and $\{\text{non-delimiter}\}$ components of tx in the same order.

Step 2.5. Replace d by the immediate components of t_1 in the same order.

Step 3. Delete from t any $\{\text{delimiter}\}$ containing a $\emptyset$ or a $\{\text{comment}\}$. This must not cause t to be deleted.

Step 4. Let $\text{nt}(i)$, $i=1, \dots, n$, be the ordered list of nodes which are the immediate components of the $\{\text{delimiters}\}$ and $\{\text{non-delimiters}\}$ in t .

Step 5. There must exist one and only one tree, mt , which is a complete tree with respect to the middle-level syntax for the root-node cn and satisfies any additional constraints specified with that syntax, and which is such that mt contains m terminal nodes $\text{mnt}(j)$, $j=1, \dots, m$ and there is a one-to-one correspondence between the $\text{nt}(i)$, $i=1, \dots, n$ and $\text{mnt}(j)$, $j=1, \dots, m$ taken in left-to-right order as specified by Case 5.1 through Case 5.5 except for the following instances:

If $\text{mnt}(j)$ is an $\{\text{environment-specification}\}$ or an $\{\text{options-specification}\}$ then it corresponds to k nodes $\text{nt}(i)$, $i=1, \dots, i+k-1$ such that

(1) no $\text{nt}(i)$, $i=1, \dots, i+k-1$ is a $\{/**\}$;

(2) all nodes $\text{nt}(i)$, $i=1, \dots, i+k-1$ which are either a $\{/**\}$ or a $\{/**\}$ must be matched in the normal way for balancing parentheses.

Case 5.1. $\text{mnt}(j)$ is a keyword.

$\text{nt}(i)$ must be an $\{\text{identifier}\}$ containing the same terminals as the characters appearing either in the denotation of $\text{mnt}(j)$ or in the abbreviation for $\text{mnt}(j)$. (See Section 2.7.)

Case 5.2. $\text{mnt}(j)$ is a non-bracketed category-name other than a keyword.

$\text{mnt}(j)$ and $\text{nt}(i)$ must be equal.

- Step 2. Let $nt[i], i=1, \dots, n$ be the ordered sequence of components of sl which are such that for each component, c , the following conditions are satisfied:
- (1) the category-name of c is terminal with respect to the middle-level syntax, and
 - (2) c is not contained in any component of sl whose category-name is terminal with respect to the middle-level syntax.
- Step 3. There must exist one and only one tree, ht , with root-node of the same type as cn such that the following conditions are true:
- (1) ht is a complete tree with respect to the syntax composed of all the production-rules occurring in the high-level syntax, and in the middle-level syntax, and
 - (2) ht contains n terminal nodes $htn[j], j=1, \dots, n$, such that for every $i, i=1, \dots, n$, the type of the node $nt[i]$ is the same as the type of $htn[i]$.
- Step 4. For each $i, i=1, \dots, n$, replace $htn[i]$ by $nt[i]$. Return ht .

Case 5.3. $nmt[j]$ is a $\{radix-factor\}$.

$nt[i]$ must be an $\{identifier\}$ such that the ordered sequence, seq , of its terminals can be a denotation of an immediate component of a $\{radix-factor\}$.

Replace $nmt[j]$ by a $\{radix-factor\}$: seq .

Case 5.4. $nmt[j]$ is an $\{imaginary-constant\}$, a $\{real-constant\}$, or an $\{integer\}$.

$nt[i]$ must be an $\{arithmetic-constant\}$ containing just a node, lc , with the same type as $nmt[j]$. (There may be intermediate nodes between $nt[i]$ and lc , but no side branches.)

Replace $nmt[j]$ by the tree with root-node lc .

Case 5.5. $nmt[j]$ is one of the following:

```

{identifier}
{arithmetic-constant}
{simple-bit-string-constant}
{simple-character-string-constant}
{isub}
{fletter}

```

$nt[i]$ and $nmt[j]$ must be of the same type.

Replace $nmt[j]$ by $nt[i]$.

Step 6. Each $\{description\}$ and $\{generic-description\}$ must contain a subtree.

Step 7. Return mt .

4.2.3 HIGH-LEVEL-PARSE

Operation: high-level-parse(sl, cn)

where sl is a $\{sentence-list\}$,
 cn is a tree with a single node, whose type is a non-terminal of the high-level syntax.

result: a complete tree with respect to the Concrete Syntax for cn .

Step 1. For each $\{single-statement\}, s$, component of sl , in left-to-right order perform Steps 1.1 through 1.3.

Step 1.1. If s contains a $\{begin-statement\}$, $\{do-statement\}$, or $\{procedure-statement\}$, then attach $\{unmatched\}$ to s .

Step 1.2. If s contains an $\{end-statement\}$ not containing an $\{identifier\}$ then remove the rightmost preceding $\{unmatched\}$.

Step 1.3. If s contains an $\{end-statement\}$ containing an $\{identifier\}, id$ then perform Steps 1.3.1 and 1.3.2.

Step 1.3.1. Let rpu be the rightmost preceding $\{single-statement\}$ containing $\{unmatched\}$ such that rpu contains also a tree of the form $\{statement-name\}, sn: \{identifier\}, idsn$; and $idsn$ is equal to id . sn must not contain a $\{signed-integer-commalist\}$. rpu must exist.

Step 1.3.2. Let k be the number of $\{unmatched\}$ components of sl following rpu and preceding s . Let es be a

```

{sentence}:
{single-statement};
{end-statement};
END;};

```

Attach k copies of es to sl immediately preceding s . Delete the $k+1$ instances of $\{unmatched\}$ which immediately precede s .

4.3 Completion of the Concrete Procedure

The {concrete-external-procedure} is "completed" in the sense that all declarations are constructed or completed.

Operation: complete-concrete-procedure

- Step 1. Perform reorganize.
- Step 2. Perform construct-explicit-declarations.
- Step 3. Perform complete-structure-declarations.
- Step 4. Perform construct-contextual-declarations.
- Step 5. Perform construct-implicit-declarations.
- Step 6. Perform complete-declarations.
- Step 7. Perform validate-concrete-declarations.

4.3.1 REORGANIZE

The {concrete-external-procedure} is reorganized in various ways to simplify and complete it.

Operation: reorganize

- Step 1. Perform complete-options.
- Step 2. Perform modify-statement-names.
- Step 3. Perform complete-attribute-implications.
- Step 4. For each {declaration-commalist}, ds immediate component of a {declare-statement} component of the {concrete-external-procedure} perform defactor-declarations (ds) to obtain a {declaration-commalist}, dds and replace ds by dds.

4.3.1.1 Complete-options

Various modifications are made to {put-statements}, {get-statements}, and {format-statements} to complete their options. These are performed before the application of any {default-statements}.

Operation: complete-options

- Step 1. For each {get-statement}, gs component of the {concrete-external-procedure} perform Steps 1.1 and 1.2.
 - Step 1.1. If gs does not contain a {file-option} or a {get-string}, then perform parse("FILE(SYSIN)", {file-option}) to obtain a {file-option}, fo, and attach fo to gs.
 - Step 1.2. If gs contains a {copy-option}, co which does not contain a {reference}, then perform parse("COPY(SYSPRINT)", {copy-option}) to obtain a {copy-option}, nco, and replace co by nco.
- Step 2. For each {put-statement}, ps component of the {concrete-external-procedure}, if ps does not contain a {file-option} or a {put-string}, then perform parse("FILE(SYSPRINT)", {file-option}) to obtain a {file-option}, fo, and attach fo to ps.
- Step 3. For each {tab-format}, tf component of the {concrete-external-procedure} where tf does not contain an {expression}, perform parse("TAB(1)", {tab-format}) to obtain a {tab-format}, ntf, and replace tf by ntf.

Step 4. For each {skip-option}, sk component of the {concrete-external-procedure}, if sk does not contain an {expression}, then perform parse("SKIP(1)", {skip-option}) to obtain nsk, and replace sk by nsk.

Step 5. For each {skip-format}, sf component of the {concrete-external-procedure}, if sf does not contain an {expression}, then perform parse("SKIP(1)", {skip-format}) to obtain nsf, and replace sf by nsf.

Step 6. For each {radix-factor}, rf component of the {concrete-external-procedure}, if rf contains only B then replace rf by a {radix-factor}, Bi.

4.3.1.2 Modify-statement-names

The {statement-name} components of a {declare-statement} or {default-statement} are removed and attached to {null-statement}s. (It is not possible for control to branch to a declaration or a default during execution.) Multiple occurrences of {statement-name}s in the {prefix-list} component of a {procedure}, or in {units} which contain {entry-statements}, are also simplified.

Operation: modify-statement-names

Step 1. For each {unit}, u immediate component of a {unit-list}, ul component of the {concrete-external-procedure}, where u immediately contains a {statement-name-list}, snl and a {declare-statement} or {default-statement}, perform Steps 1.1 and 1.2.

Step 1.1. Let pl be a {prefix-list}. For each {statement-name}, sn of snl perform Step 1.1.1.

Step 1.1.1. Append {prefix}: sn; to pl, where sn; is a copy of sn.

Step 1.2. Let un be a

```
{unit}:
{executable-unit}:
    pl
    {executable-single-statement}:
        {null-statement}:
            {};
```

Attach un to ul immediately preceding u. Delete snl.

Step 2. For each {procedure}, pc, where pc is a component of the {concrete-external-procedure} and pc immediately contains a {prefix-list}, pf, perform Steps 2.1 and 2.2.

Step 2.1. pf must contain at least one {statement-name}. For each {statement-name}, sn component of pf, after the leftmost one, perform Steps 2.1.1 through 2.1.3.

Step 2.1.1. Let sn; be a copy of sn. Let un1 be a

```
{unit}:
    {statement-name-list}:
        sn;
    {entry-statement}, es:
        ENTRY
        {};
```

Step 2.1.2. If the {procedure-statement} immediate component of pc contains an {entry-information}, ei then let eic be a copy of ei, delete any RECURSIVE subnode of eic, and attach eic to es.

Step 2.1.3. Attach un1 to pc as the first component of the {unit-list} immediate component of pc.

Step 2.2. Delete every {statement-name} except the first from pf.

Step 3. For each {unit}, u component of the {concrete-external-procedure}, where u immediately contains a {statement-name-list}, snl and an {entry-statement}, es and snl contains more than one {statement-name}, perform Steps 3.1 and 3.2.

Step 3.1. For each {statement-name}, sn component of snl after the first, perform Steps 3.1.1 and 3.1.2.

Step 3.1.1. Let snc be a copy of sn. Let esc be a copy of es. Let unl be a

```
{unit}:
  {statement-name-list}:
    snc;
  esc;
```

Step 3.1.2. Attach unl to the {unit-list} which immediately contains u so that unl immediately precedes u.

Step 3.2. Delete every {statement-name} from snl except the first.

4.3.1.3 Complete-attribute-implications

The {dimension-attribute}, {precision}, and the {data-attribute}: FIXED, can be implied without the use of the keywords DIMENSION, PRECISION, and FIXED. These implications are replaced by explicit declarations of these attributes.

Operation: complete-attribute-implications

Step 1. For each {dimension-suffix}.ds component of the {concrete-external-procedure} such that ds is not a component of a {dimension-attribute} append a

```
{data-attribute}:
  {dimension-attribute}:
    DIMENSION
  dsc;;
```

where dsc is a copy of ds, to the node immediately containing ds, and delete ds.

Step 2. For each {generic-description}.gda component of the {concrete-external-procedure}, if gda immediately contains an {asterisk-bounds}.ab then append a {generic-data-attribute}: DIMENSION ab; to the {generic-data-attribute-list} component of gda and delete ab.

Step 3. For each {attribute}: {data-attribute}.atr; or {data-attribute}.atr which is an immediate component of a list, l, component of the {concrete-external-procedure}, if atr simply contains a {precision}.p but not PRECISION then append to l a

```
{data-attribute}:
  PRECISION
  pl;
```

where pl is a copy of p, and delete p.

Step 4. For each {data-attribute-list}.al or {attribute-list}.al of the {concrete-external-procedure}, if al simply contains a {data-attribute} with {precision}.p and p contains a {scale-factor} then append to al a

```
{data-attribute}:
  FIXED.
```

4.3.1.4 Defactor-declarations

The syntax of {declare-statement} allows {identifier}s to be factored together to give them the same structuring or attributes. This factoring is unravelled to provide a single {declaration} for each {identifier}.

Operation: defactor-declarations(dc)

where dc is a {declaration-commalist}.

result: a {declaration-commalist}.

Step 1. Let ds be a copy of dc.

Step 2. For each {declaration}.d immediate component of ds, if d immediately contains a {declaration-commalist}.dcl then perform defactor-declarations(dcl) to obtain a {declaration-commalist}.dcl1 and replace dcl by dcl1.

Step 3. For each {declaration}.d immediate component of ds, if d immediately contains a {declaration-commalist}.d1 then perform Steps 3.1 through 3.3.

Step 3.1. If d immediately contains a {level}.lv then there must not be a {level} declaration-contained in any of the {declarations} immediately contained in d1, and attach a copy of lv to each {declaration} immediate component of d1.

Step 3.2. If d immediately contains an {attribute-list}.al then append a copy of each {attribute} of al to each {declaration} immediate component of d1.

Step 3.3. Replace d by the immediate components of d1 in the same order.

Step 4. Return ds.

4.3.2 CONSTRUCT-EXPLICIT-DECLARATIONS

The occurrence of an {identifier} in the {concrete-external-procedure} as an immediate component of a {declaration} explicitly specifies that {identifier} as the name of some data item. Other contexts may also constitute explicit declarations of an {identifier}, in which case a {declaration} is created for it.

{declared-statement-names} ::= [{name-list}] [{procedure} | {begin-block}]

{name} ::= {statement-name} [{entry-information}]

Operation: construct-explicit-declarations

Step 1. For each {procedure}.p contained in the {concrete-external-procedure} perform declare-parameters(p).

Step 2. Let p be the {procedure} immediate component of the {concrete-external-procedure}. Perform declare-statement-names(p) to obtain a {declared-statement-names}.dsn. Replace p by the {procedure} of dsn. The {name-list}.nl of dsn must not contain a {signed-integer-commalist}.

Step 3. Perform construct-statement-name-declarations(nl,attr) where attr is an

```
{attribute}:
  {data-attribute}:
    ENTRY();;
```

to obtain a {unit} containing a {declaration-commalist}.dc. Replace all occurrences of {attribute}: INTERNAL; in dc by {attribute}: EXTERNAL.

Step 4. Attach dc to the {concrete-external-procedure}.

4.3.2.1 Declare-parameters

Each {identifier} occurring in a {parameter-name-commalist} represents an explicit declaration of a parameter. Unless it is already declared, a declaration is introduced for it. If it has been declared erroneously then a conflict will occur when this declaration is being transformed to its abstract equivalent.

Operation: declare-parameters(p)

where p is a {procedure}.

Step 1. For each {parameter-name-commalist}, pl block-component of p perform Step 1.1.

Step 1.1. pl must be such that no two {identifier} components of pl are equal. For each {identifier}, id component of pl perform Steps 1.1.1 through 1.1.3.

Step 1.1.1. Perform find-applicable-declaration(id) to obtain d.

Step 1.1.2. If d is <absent>, or d is not a concrete-block-component of p, or d is a concrete-block-component of p but declaration-contains a {level} whose value is not 1, then let d be a

```
{unit};
{declare-statement};
DECLARE
{declaration-commalist};
{declaration};d;
id;;
{;};;
```

and append d to the {unit-list} of p.

Step 1.1.3. If d does not contain {attribute}: PARAMETER; then attach {attribute}: PARAMETER; to d.

4.3.2.2 Declare-statement-names

A {statement-name} may occur as a component of a {unit} which contains an {entry-statement}, {format-statement}, or some other {executable-unit}, or as a component of a {procedure}. These contexts are used to determine the {attributes} to be attached to the explicit declarations for the {identifier} of the {statement-name}.

Operation: declare-statement-names(p)

where p is a {procedure} or {begin-block}.

result: a {declared-statement-names}.

Step 1. Let pc be a copy of p and let lnl, pnl, fnl, and enl each be a {name-list} with no components.

Step 2. For each {statement-name}, sn concrete-block-component of pc perform Step 2.1.

Step 2.1. One of the following Cases must apply:

Case 2.1.1. sn is simply contained in an {executable-unit}, eu.

Append sn to lnl.

Case 2.1.2. sn is simply contained in a {unit} that has a {format-statement} immediate component.

Append sn to fnl.

Case 2.1.3. sn is contained either in a {prefix-list} immediate component of pc where pc has {procedure-statement}, ep, or in a {unit} with an {entry-statement}, ep as an immediate component.

Case 2.1.3.1. ep has an {entry-information}, ei.

If ep is an {entry-statement}, then ei must not contain RECURSIVE. Append {name}: sn ei; to enl.

Case 2.1.3.2. ep has no {entry-information}.

Append {name}: sn; to enl.

Step 3. For each {procedure} or {begin-block}, pn concrete-block-component of pc perform Step 3.1.

Step 3.1. Perform declare-statement-names(pn) to obtain a {declared-statement-names}, dsn. Append the elements of the {name-list} of dsn to pnl. Replace pn by the {procedure} or {begin-block} of dsn. If pn is a {begin-block}, then the {name-list} component of dsn must be empty.

Step 4. If lnl is not empty then perform construct-statement-name-declarations(lnl, lattr), where lattr is an

```
{attribute};
{data-attribute};
LABEL;;
```

to obtain a {unit}, d, and append d to the {unit-list} of pc.

Step 5. If fnl is not empty then perform construct-statement-name-declarations(fn, fattr), where fattr is an

```
{attribute};
{data-attribute};
FORMAT;;
```

to obtain a {unit}, d and append d to the {unit-list} of pc.

Step 6. If pnl is not empty then perform construct-statement-name-declarations(pnl, pattr), where pattr is an

```
{attribute};
{data-attribute};
ENTRY;;
```

to obtain a {unit}, d and append d to the {unit-list} of pc.

Step 7. Return the {declared-statement-names} consisting of enl and pc.

4.3.2.3 Construct-statement-name-declarations

This operation takes a {name-list} containing {statement-names} and possibly {entry-informations} and constructs a {declare-statement} for them. The type of the {statement-names} is given by a supplied {attribute}. Any type of {statement-name} may contain one or more {integers}, signifying that it is one element of an array of {statement-names}. The explicit declaration for this array is constructed with a {dimension-attribute} component.

Operation: construct-statement-name-declarations(nl, att)

where nl is a {name-list},
att is an {attribute}.

result: a {unit}.

Step 1. Let nlc be a copy of the {name-list}, nl. Let un be a

```
{unit};
{declare-statement};
DECLARE
{declaration-commalist}, dcl
{;};
```

Step 2. While nlc contains any element, perform Steps 2.1 through 2.7.

Step 2.1. Let id be the {identifier} component of the first element of nlc.

Step 2.2. Let tnl be a {name-list} containing a copy of all elements of nlc whose {statement-name} immediately contains an {identifier} equal to id.

Step 2.3. Let d be a

```
{declaration}:
id
{attribute-list}, al:
{attribute}:
INTERNAL;
{attribute}:
CONSTANT;
att.
```

Step 2.4. If tnl contains more than one element or if tnl contains only one element and this element has a {signed-integer-commalist}, then perform Steps 2.4.1 through 2.4.6.

Step 2.4.1. Let m be the number of elements of tnl. Each element of tnl must contain a {signed-integer-commalist} with the same number of elements, n. tnl must be such that no two {signed-integer-commalist} components of tnl represent the same ordered sequence of numerical values.

Step 2.4.2. Let sifi,j be the j'th {signed-integer} of the {signed-integer-commalist} of the i'th element of tnl.

Step 2.4.3. Let ub[k] and lb[k], k=1,...,n be, respectively, the largest and least value of sifi,kj,...,sifm,kj.

Step 2.4.4. Let bpl be the {bound-pair-list} containing n {bound-pair} elements such that the j'th element is a

```
{bound-pair}:
{lower-bound}:
el;
{upper-bound}:
e2;;
```

where e1 and e2 are {extent-expressions} representing lb[j] and ub[j] respectively.

Step 2.4.5. Let bpci be the {bound-pair-commalist} produced by inserting commas as appropriate in bpl.

Step 2.4.6. Append an

```
{attribute}:
{data-attribute}:
{dimension-attribute}:
DIMENSION
{dimension-suffix}:
(
bpci
);;;;
to al, the {attribute-list} of d.
```

to al, the {attribute-list} of d.

Step 2.5. If al contains ENTRY then perform Steps 2.5.1 through 2.5.2.

Step 2.5.1. If any {entry-information} of tnl contains a {returns-descriptor}, then perform Steps 2.5.1.1 and 2.5.1.2.

Step 2.5.1.1. Each element of tnl must contain a {returns-descriptor}.

Note: a check for consistency is made in copy-descriptors.

Step 2.5.1.2. Append to al an

```
{attribute}:
{data-attribute}:
rd.
```

Step 2.5.2. If any {entry-information} of tnl contains an {options}, op then perform Steps 2.5.2.1 and 2.5.2.2.

Step 2.5.2.1. Each element of tnl must contain an {options}.

Note: a check for consistency is made in copy-descriptors.

Step 2.5.2.2. Append to al an

```
{attribute}:
op.
```

Step 2.6. If dcl has a subcomponent, append {,} to dcl. Append d to dcl.

Step 2.7. Delete from nlc all those elements of which there is a copy in tnl.

Step 3. Return un.

4.3.3 COMPLETE-STRUCTURE-DECLARATIONS

A structure is specified by a hierarchical set of names that refers to a group of individual items each of which may have a different data type. Conversely, an array is specified by a single name referring to a group of items all of the same data type. The component items of a structure may themselves be structures or arrays.

Operation: complete-structure-declarations

Step 1. For each {declaration-commalist}, {description-commalist}, or {generic-description-commalist}, dl component of the {concrete-external-procedure} perform determine-structure(dl) to obtain a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}, dlm. Replace dl by dlm.

Step 2. Let dl[j] be the j'th {declaration} component of the {concrete-external-procedure} that declaration contains LIKE. For each dl[j] perform Step 2.1.

Step 2.1. Perform expand-like-attribute(dl[j]) to obtain a {declaration-commalist}, dcmll[j].

Step 3. For each dl[j] perform Step 3.1.

Step 3.1. Let dl be the {declaration-commalist} that immediately contains dl[j]. Attach a {,} followed by the elements of dcmll[j] in sequence so that they immediately follow dl[j].

Step 4. Delete all {attribute}: LIKE {unsubscripted-reference}; components of {declarations} in the {concrete-external-procedure}.

Step 5. For each {declaration-commalist}, {description-commalist}, or {generic-description-commalist}, dl component of the {concrete-external-procedure} that declaration contains STRUCTURE, perform convert-to-logical-levels(dll) to obtain dll1, a node of the same type as dl. Perform propagate-alignment(dll) to obtain dll2, a node of the same type as dll1. Replace dl by dll2.

4.3.3.1 Determine-structure

Operation: determine-structure(cml)

where cml is a {declaration-commalist}, {description-commalist}, or {generic-description-commalist},
result: a {declaration-commalist}, a {description-commalist}, or {generic-description-commalist}.

Step 1. Let cmlc be a copy of cml. Let e[j] be the j'th immediate component of cmlc that is not {},. Let n be the number of such components.

Step 2. For each e[j] that immediately contains a {level},lv, perform Steps 2.1 through 2.4.

Step 2.1. lv must not be 0.

Step 2.2. If j is less than n and e[j+1] immediately contains a {level} whose numeric value is greater than that of lv, then attach STRUCTURE to e[j].

Step 2.3. If e[j] declaration-contains LIKE then attach STRUCTURE to e[j]. In this case, e[j] must not contain more than one instance of LIKE and cml must be a {declaration-commalist}.

Step 2.4. If the numeric value of lv is greater than one, attach MEMBER to e[j].

Step 3. For each e[j], all of the following must be false:

- (1) e[j] immediately contains {level} and does not declaration-contains STRUCTURE or MEMBER.
- (2) e[j] immediately contains {level} whose value is 1 and e[j] declaration-contains MEMBER.
- (3) e[j] declaration-contains STRUCTURE or MEMBER and does not immediately contain {level}.
- (4) e[j] declaration-contains STRUCTURE, does not declaration-contains LIKE and either j is equal to n or e[j+1] does not declaration-contains a {level} whose value is greater than the {level} declaration-component of e[j].
- (5) e[j] declaration-contains MEMBER and either j is equal to one or e[j-1] does not declaration-contains MEMBER or STRUCTURE.
- (6) e[j] declaration-contains STRUCTURE and LIKE, j is less than n, and e[j+1] declaration-contains MEMBER and a {level} whose numeric value is greater than the numeric value of the {level} declaration-contains in e[j].
- (7) e[j] declaration-contains LIKE but not a {level}.

Step 4. Return cmlc.

4.3.3.2 Expand-like-attribute

Operation: expand-like-attribute(d)

where d is a {declaration}.

result: a {declaration-commalist}.

Step 1. d declaration-contains an

```
{attribute:
  LIKE
  {unscripted-reference},r.
```

Step 2. Perform find-applicable-declaration(r) to obtain ld. ld must be a {declaration} that declaration-contains STRUCTURE and must not declaration-contains LIKE.

Step 3. Let ld be immediately contained in the {declaration-commalist},dcm1. Let e[j] be the j'th element of dcm1 that is not {},. Let k be such that e[k] is identical to ld and let n be the number of elements of dcm1 that are not {},.

Step 4. Let m be the numeric value of the immediately contained {level} of d. Let lv be the numeric value of the immediately contained {level} of e[k] and let cf be the numeric value (m-lv).

Step 5. Let cl be a {declaration-commalist} with no elements. Let i be k+1. Let t be the smallest integer greater than k such that one of the following is true: t is equal to n, e[t+1] does not declaration-contains MEMBER, or e[t+1] immediately contains a {level} whose numeric value is less than or equal to lv. Perform Steps 5.1 through 5.3 while i is less than or equal to t.

Step 5.1. Let ec be a copy of e[i]. Let the numeric value of the immediately contained {level} of ec be lvc. Replace this {level} by one whose numeric value is lvc+cf.

Step 5.2. Append ec to cl, appending commas where necessary.

Step 5.3. Let i be i+1.

Step 6. cl must not contain any instance of LIKE.

Step 7. Return cl.

4.3.3.3 Convert-to-logical-levels

Operation: convert-to-logical-levels(cml)

where cml is a {declaration-commalist}, {description-commalist},
or {generic-description-commalist}.

result: a {declaration-commalist}, {description-commalist},
or {generic-description-commalist}.

Step 1. Let cmlc be a copy of cml. Let e[j] be the j'th component of cmlc that is not {}, and let m be the number of such components. Let l[j] be the {level} declaration-contains in e[j], if it exists, and let n[j] be the numeric value of l[j].

Step 2.

Case 2.1. There exists an integer t less than m such that e[t] and e[t+1] each declaration-contains a {level}, and n[t+1] is greater than n[t]+1.

Let j be the least such t.

Case 2.2. (Otherwise).

Return cmlc.

Step 3. Let j1 be the least integer greater than j such that either j1 is equal to m, or e[j1+1] does not declaration-contains MEMBER, or n[j1+1] is less than or equal to n[j]+1. For i=j+1,...,j1, perform Step 3.1.

Step 3.1. Replace l[i] by a {level} whose numeric value is n[i]-1.

Step 4. Go to Step 2.

Step 2. Let b be the concrete-block that block-contains r . If r is contained in a {statement-name}, sn then perform Step 2.1.

Step 2.1. If sn is contained in a {unit} which simply contains {entry-statement} or sn is contained in a {prefix-list} immediate component of a {procedure} then let b be the concrete-block which block-contains b .

Step 3. Let id be the rightmost {identifier} component of idl .

Step 4. Let dl be a {declaration-designator-list} each component of which designates a {declaration} that is a concrete-block-component of b and that immediately contains an {identifier} that is equal to id .

Step 5. Delete from dl any component that designates a {declaration}, d such that find-fully-qualified-name(d) returns an {identifier-list}, idl such that idl does not contain an ordered sublist of the {identifier}s contained in idl .

Step 6.

Case 6.1. dl is empty.

Let b be the concrete-block that block-contains b . If there is no such block then return $\langle \text{absent} \rangle$; otherwise, go to Step 4.

Case 6.2. dl contains a single component.

Let d be the {declaration} designated by the single component of dl . Return d .

Case 6.3. dl contains more than one component.

dl must contain exactly one component that designates a {declaration}, d such that find-fully-qualified-name(d) returns an {identifier-list} equal to idl . Return d .

4.3.3.6 Find-fully-qualified-name

Each {declaration} has a fully qualified name associated with it. If it is an array or item declaration then this is a single {identifier}. If the {declaration} is a member of a structure then it may have as many {identifier}s as its logical level-number indicates its depth of embedding to be.

Operation: find-fully-qualified-name(d)

where d is a {declaration}.

result: an {identifier-list}.

Case 1. d declaration-contains MEMBER.

Step 1.1. Let $dcm1$ be the {declaration-commalist} that immediately contains d . Let dl be the rightmost preceding {declaration} of $dcm1$ that declaration-contains a {level} whose numeric value is less than the numeric value of the {level} of d .

Step 1.2. Perform find-fully-qualified-name(dl) to obtain an {identifier-list}, idl .

Step 1.3. Append the {identifier} immediate component of d to idl .

Step 1.4. Return idl .

Case 2. d does not declaration-contains MEMBER.

Step 2.1. Let id be the {identifier} immediate component of d .

Step 2.2. Return {identifier-list}: id .

4.3.3.4 Propagate-alignment

Operation: propagate-alignment(cml)

where cml is a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

result: a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

Step 1. Let $cmlc$ be a copy of cml . Let $e[j]$ be the j 'th component of $cmlc$ that is not {,} and let m be the number of such components. Let $l[j]$ be the {level} declaration-contained in $e[j]$, if it exists, and let $n[j]$ be the numeric value of $l[j]$.

Step 2. For $i=1, \dots, m$, perform Steps 2.1 and 2.2.

Step 2.1. $e[i]$ must not declaration-contains both ALIGNED and UNALIGNED.

Step 2.2.

Case 2.2.1. $e[i]$ declaration-contains MEMBER, and $e[i]$ declaration-contains neither ALIGNED nor UNALIGNED.

Let k be the greatest integer such that k is less than i and $n[k]$ is equal to $n[i]-1$. If $e[k]$ declaration-contains ALIGNED or UNALIGNED, then attach ALIGNED or UNALIGNED, respectively, to $e[i]$.

Case 2.2.2. (Otherwise).

No action.

Step 3. For $i=1, \dots, m$, perform Step 3.1.

Step 3.1. If $e[i]$ contains STRUCTURE and ALIGNED, or STRUCTURE and UNALIGNED, delete ALIGNED or UNALIGNED, respectively, from $e[i]$.

Step 4. Return $cmlc$.

4.3.3.5 Find-applicable-declaration

Operation: find-applicable-declaration(r)

where r is an {identifier}, {unsubscripted-reference}, {basic-reference}, or {reference}.

result: a {declaration} or $\langle \text{absent} \rangle$.

Step 1.

Case 1.1. r is an {identifier}.

Let idl be an {identifier-list}: r .

Case 1.2. r is an {unsubscripted-reference}. Append, in order, a copy of each {identifier} component of r to an {identifier-list}, idl .

Case 1.3. r is a {basic-reference}.

Append, in order, a copy of each {identifier} component of r which is not a component of an {arguments}, in order, to an {identifier-list}, idl .

Case 1.4. r is a {reference}.

Let br be the {basic-reference} immediate component of r . Perform find-applicable-declaration(br) to obtain d . Return d .