HONEYWELL

MULTICS PL/I LANGUAGE SPECIFICATION

SOFTWARE

MULTICS

PL/I LANGUAGE SPECIFICATION

SUBJECT

A Semi-formal Definition of the Multics PL/I Language

SOFTWARE SUPPORTED

Multics Software Release 8.2

SPECIAL INSTRUCTIONS

This manual supersedes the previous edition, Rev. 1, dated January 1974, which superseded: Rev. 0, dated July 1972, *The Multics PL/I Language Specification* (1969), and *A Users Guide to the Multics PL/I Implementation* (October 1969).

Includes update pages issued as Addendum A in October 1977, Addendum B July 1978, Addendum C July 1979, Addendum D, September 1979, and Addendum E in March 1981.

ORDER NUMBER

AG94-02

July 1976

Honeywell

PREFACE

This reference manual contains detailed information for users of the Multics PL/I language. Contained herein are exact answers to detailed questions concerning the syntax and semantics of PL/I. Additional information useful to the Multics PL/I programmer is found in the following documents:

Multics PL/I Reference Manual - Order No. AM83

Multics Programmers' Manual (MPM):

MPM Reference Guide - Order No. AG91

MPM Commands and Active Functions - Order No. AG92

MPM Subroutines - Order No. AG93

MPM Peripheral Input/Output - Order No. AX49

MPM Subsystem Writers' Guide - Order No. AK92

The Multics PL/I Reference Manual provides an introduction to Multics PL/I, furnishes guidance for writing a Multics PL/I program, and explains the relationship between Multics PL/I and the run-time environment supplied by the Multics system.

The MPM Reference Guide describes in general terms the functions and features of the Multics system; for example, representation of PL/I data.

The MPM Commands and Active Functions contains descriptions of the commands in the command repertoire and the active functions available to the Multics system.

The MPM Subroutines contains descriptions of the subroutines available on the system.

The MPM Peripheral Input/Output contains descriptions of commands and subroutines used to perform peripheral I/O. This manual includes the commands and subroutines that manipulate tapes and disks as I/O devices as well as such special-purpose communications I/O as binary synchronous operations.

The MPM Subsystem Writers' Guide contains such detailed descriptions as the the exact layout of a PL/I activation record (stack frame), the internal format of a PL/I area, and the calling sequence generated for a PL/I call. Most users will not require the extent of detail contained in this volume.

The MPM Communications Input/Output contains descriptions of commands and subroutines used to perform communications I/O. This manual includes information on terminal types.

Significant Changes in AG94 , Revision 2 , Addendum E

This list of changes includes only those changes made to AG94 Addendum E that were accompanied by changes to the Multics PL/I implementation.

1. Clarification (manual only) of operand conversion for the exponentiation operator.

CONTENTS

Page

SECTION 1

INTRODUCTION

This document is a semi-formal definition of the language supported by the Multics PL/I compiler. The document is intended to be used as a reference manual by programmers who need exact answers to detailed questions concerning the syntax and semantics of Multics PL/I. In keeping with that purpose, the document defines the language in an analytic rather than a synthetic manner; i.e., it explains the meaning of programs, but does not describe how to construct programs.

1.1 Language

The Multics PL/I language is a dialect of the American National Standard Programming Language PL/I, ANSI X3.53-1976; it also conforms to International Standards Organization standard 6160-1979. Refer to Appendix A for a description of the two differences between standard PL/I and Multics PL/I. The languages are so similar that nearly all Multics PL/I programs are valid programs in standard PL/I.

1.2 Method of Definition

The language is defined using a formal meta-language to define the syntax and prose to describe the semantics. Although this is a semi-formal definition, both the syntactic and semantic descriptions are reasonably precise and complete.

Example:

```
<based attribute>::= based[(<locator reference>)]
```

When the prose refers to a <based attribute> or a <locator reference>, these terms appear exactly as they do in the syntax rule. When a keyword appears in prose, it is enclosed in quotes to distinguish it from the text; for example, "based" and "float."

Terms defined in prose are underlined when defined and not underlined thereafter. Examples are provided to aid understanding but are not intended to be comprehensive or definitive. All examples are clearly set off from the rest of the text as shown by the example on this page. Within examples where empty space might be misleading, ß denotes a blank.

1.2.1 Meta-Language

The syntax of the PL/I language is defined by a set of syntax rules expressed in a formal notation derived from Backus-Naur Form. Each syntax rule describes a character-string or pattern of characters that constitutes a syntactic construct of the PL/I language. The complete set of syntax rules describes all syntactically correct PL/I programs.

Example:

```
<skip option>::= skip[(<expression>)]
```

In this example, <skip option> is a notation variable that represents the character-string described by the syntax expression on the right of the definition symbol "::= ". "skip" is a notation constant that represents an actual occurrence of the character-string "skip." <expression> is a notation variable defined by another syntax rule. [and] are brackets that indicate that the parenthesized <expression> is optional. The brackets are symbols of the meta-language, they are not part of the <skip option>.

Readers familiar with formal grammars should note that these syntax rules are designed to aid presentation of both syntax and semantics. Therefore, constructs like multiple closure of <group>s and <block>s and the balancing of "then" and "else" keywords of <if statement>s are not described by the syntax rules, but are described in prose.

Readers not familiar with formal descriptions of syntax should not be concerned if they do not fully understand the formalism. They are urged to compare examples against the syntax rules and from time-to-time consult the description of the formalism given in the following section.

1.2.2 Syntax Expressions

A syntax expression consists of operators, notation variables, notation constants, braces { } and brackets [ ]. The operators have a property known as precedence that determines the order in which the syntax expression is interpreted. Operators with higher precedence are interpreted before operators with lower precedence. Braces and brackets have the effect of parentheses and force the interpretation of their contents as subexpressions.

The operators of the meta-language in order of decreasing precedence are:

Repetition	X...	Denotes one or more occurrences of X.
Juxtaposition	X Y	Denotes an occurrence of X followed by an occurrence of Y.
Alternation	X|Y	Denotes an occurrence of X or Y but not both.

Brackets and braces define the order of expression interpretation. Brackets also indicate that the syntax described by their enclosed subexpression is optional.

[X]	Denotes zero or one occurrence of X.
{A|B}C	Denotes an A or a B, followed by a C.

Example:

A|B|C

describes any of the following three strings:

A B C

Example:

{A|B}C

describes either of the following two strings:

AC BC

Example:

[A|B]C

describes any of the following three strings:

AC BC C

Example:

AB[C]...

describes any string beginning with AB followed by zero or more occurrences of the letter C.

AB ABC ABCCCC

Example:

A B...

describes any string beginning with A and followed by one or more occurrences of the letter B.

AB ABB ABBBBB

Example:

AB...

describes any string consisting of one or more occurrences of AB.

AB ABAB ABABABAB

1.2.3 A Formal Definition of the Meta-Language

Syntax:

<meta-language>::= <syntax rule>...

<syntax rule>::= <notation variable>::=<blank>
<syntax expression>

<syntax expression>::= <sequence>|
<sequence>|<syntax expression>

<sequence>::= <unit>|<unit><sequence>

<unit>::= <notation variable>|<notation constant>|<unit>...|
{<syntax expression>}|[<syntax expression>]

<notation variable>::= <<meta-letter>
[<meta-letter>|<blank>|-]...>

<meta-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

<blank>::= a blank space

<notation constant>::= Any string of ASCII characters not
containing a <blank>. If the string is one of the
following, it must be underlined to distinguish it
from symbols of the meta-language.

{ } [ ] | ::= < > ...

A <blank> is required between any adjacent <notation constant>s in a <sequence>.

1.3 Warning

PL/I, like most other programming languages, is a language in which it is possible to write programs whose meaning is undefined. Furthermore, it is not practical to always detect such programs either during compilation or during execution.

Because of the large number of constructs in PL/I, it is very easy to inadvertently write a program whose meaning is undefined. Programmers are advised to learn the exact rules for using each construct, and are advised to carefully consider the warning given in this section.

Only those strings described by <external procedure> are syntactically valid. All others violate the syntactic constraints specified by the syntax rules and are in error.

When the description of a language construct specifies a constraint either by means of syntax rules, by specifically enumerating the constraints as is done in Section 12, or by giving the constraint in the description of the semantics, the constraint has the following meaning:

> A program that violates the constraint may or may not be compiled by the Multics PL/I compiler. If compiled, it may or may not execute. If executed, it may or may not produce consistent results in the current or future versions of the implementation.

Constraints are given by the syntax rules or are stated clearly in the prose. In the prose, two descriptive methods are used: either the constraint is specifically described as an error or the words "must", "cannot", or "restricted" are used to imply the constraint.

Examples:

> It is an error to refer to the value ...
>
> The program is in error if ...
>
> N must not be ...

A <read statement> cannot contain ...

The value of q is restricted ...

This document explicitly states the circumstances in which the order of evaluation of expressions or statement parts is a well defined property of the language and when it is not. When the order is said to be unspecified or undefined, any program that depends on the order is in error.

SECTION 2

STRUCTURE OF A PL/I PROGRAM

2.1 External Procedure

A PL/I program consists of one or more <external procedure>s together with their operating environment. An <external procedure> is a <procedure> that is not contained within another <procedure>. An <external procedure> is the largest syntactic construct of the language and serves as the unit of input to the Multics PL/I compiler.

The set of <external procedure>s that constitute a program is determined during execution of the program as described in Section 3.

Syntax:

```
<external procedure>::= <procedure>
```

2.2 Blocks and Block Structure

The most important syntactic construct of the language is the <block>. It delimits the scope of names and is the major unit that determines the flow of control during program execution. Refer to Section 3 for a discussion of the flow of control and to Section 5 for the scope of names.

Syntax:

```
<block>::= <procedure>|<begin block>

<procedure>::= <procedure statement>
     [<procedure component>]...<end statement>

<begin block>::= <begin statement>
     [<block component>]...<end statement>

<procedure component>::= <block component>|<entry statement>

<block component>::= <block>|<group>|<declare statement>|
     <default statement>|<format statement>|
     <independent statement>
```

The full syntax and semantics of each <statement> are given in Section 12.

All of the text of a <begin block>, except the <label prefix>s of the <begin statement> and the <closure label> of its <end statement>, is contained in the <begin block>.

Example:

```
A: begin
   .
   .
   .
   end A;
```

The text shown with lines is contained in <begin block> A.

All of the text of a <procedure>, except the <label prefix>s of its <procedure statement> and each of its <entry statement>s and the <closure label> of its <end statement>, is contained in the <procedure>.

Example:

```
A: procedure
_____________
B: entry
_____________
   end A;
```

The text shown with lines is contained in <procedure> A.

The text contained in <block> A, but not contained in any other <block> contained in A, is immediately contained in <block> A.

Example:

```
P: procedure;
   ___________
   ___________
Inner: procedure;
       ___________
       ___________
    E: entry;
       ___________
        B: begin;
           _________
           _________
           end;
       end;
   end;
```

In this example, P is an <external procedure> that contains the <procedure> Inner. Inner has a secondary entry E, and Inner contains a <begin block> B. B is contained in Inner and P, and is immediately contained in Inner.

2.3 Groups

A <group> is a programming device used to determine the flow of control during program execution.

Syntax:

<group>::= <iterative group>|<noniterative group>

<iterative group>::= <iterative do>
[<block component>]...<end statement>

<noniterative group>::= <noniterative do>
[<procedure component>]...<end statement>

The effect of <group>s on the flow of control is discussed in Sections 3 and 12.

Examples:

```
A: do;
   _____
   _____
   end;
```

```
B: do i = 1 to 10;
   ____
   ____
   end;
```

In this example, the text from A to "end;" is a <noniterative group> and the text from B to "end;" is an <iterative group>.

2.4 Multiple Closure of Groups and Blocks

The syntax of a <group> or <block> requires that the <group> or <block> terminate with an <end statement>. Since <group>s and <block>s may be nested, it is possible for several <end statement>s to immediately follow each other.

Example:

```
a: begin;
   b: begin;
      c: begin;

         end;
      end;
d: end;
```

The syntax of an <end statement> allows an optional <identifier> to follow the keyword "end."

Syntax:

<end statement>::= [<prefix>]end[<closure label>];

<closure label>::= <identifier>

The <closure label> provides a means of terminating more than one <group> or <block> with a single <end statement>. The following example is equivalent to the previous example.

Example:

```
a: begin;
   b: begin;
      c: begin;
d: end a;
```

An <end statement> with a <closure label> terminates all preceding <group>s and <block>s including, but not exceeding, the nearest <group> or <block> whose first <statement> has a <label prefix> that is the same <identifier> as the <closure label>.

The program is syntactically incorrect if the <closure label> is not a <label prefix> on a preceding <begin statement>, <procedure statement>, or <do statement>.

2.5 Statements

Syntax:

```
<statement>::= <declarative statement>|
     <dependent statement>|<independent statement>

<dependent statement>::= <format statement>|
     <entry statement>|<procedure statement>|
     <begin statement>|<do statement>|<end statement>

<independent statement>::= <allocate statement>|
     <assignment statement>|<call statement>|
     <close statement>|<delete statement>|
     <free statement>|<get statement>|
     <goto statement>|<if statement>|
     <locate statement>|<null statement>|
     <on statement>|<open statement>|
     <put statement>|<read statement>|
     <return statement>|<revert statement>|
     <rewrite statement>|<signal statement>|<stop statement>|
     <write statement>

<declarative statement>::=<declare statement>|<default statement>
```

The syntax and semantics of each <statement> are given in Section 12. The effect of <declarative statement>s on the establishment of declarations is described in Section 5.

All <statement>s are executable, although the execution of a <declarative statement> or <format statement> has no effect.

The <dependent statement>s control input/output, define entries to <procedure>s, and form <procedure>s, <begin block>s, and <group>s in accordance with the syntax rules in paragraphs 2.2 through 2.4.

2.5.1 Statement Prefixes

Syntax:

```
<prefix>::= [<condition prefix>]...[<label prefix>]...

<condition prefix>::= (<prefix name>[,<prefix name>]...):

<label prefix>::= <declared name>[<prefix subscript>]:

<prefix subscript>::= ([+|-]<decimal integer>)

<declared name>::= <identifier>
```

A <label prefix> names a <statement>. Any <statement> may be labeled by a <label prefix>.

A <condition prefix> is a means of controlling the type of error checking that is to occur during execution of the <statement>. The <prefix name> must be one of the names given in paragraph 10.2, where conditions are fully described. A <condition prefix> cannot appear on a <declare statement>, <default statement>, or <entry statement>.

Example:

```
L: a = b+c;
A(3): x = y+z;
(zerodivide): p = q/r;
(overflow,size): T1: t = s+1;
```

In this example, L:, A(3): and T1: are <label prefix>s, whereas (zerodivide): and (overflow,size): are <condition prefix>s.

2.6 Lexical Syntax of PL/I

The smallest syntactic construct of the language is called a <lexeme>. Sequences of <lexeme>s form <statement>s, that in turn form the <group>s and <block>s of an <external procedure>.

Syntax:

```
<lexeme>::= <identifier>|<literal constant>|<isub>|<delimiter>
```

2.6.1 Identifiers

An <identifier> is used as a keyword or as a <declared name>. A keyword is an <identifier> used within the language to identify <statement>s or components of <statement>s. In Multics PL/I, keywords consist entirely of lower case letters.

Syntax:

```
<identifier>::= <letter>[<letter>|<digit>|_|$]...

<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|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

<digit>::= 0|1|2|3|4|5|6|7|8|9
```

In Multics PL/I, an <identifier> cannot be more than 256 characters long. Refer to the Multics PL/I Reference Manual for a discussion of the significance of a "$" in an <identifier> used as an external name.

Examples:

```
Capital
i
X
declare
tag7
ioa_
may$day
```

2.6.2 Literal Constants

A <literal constant> is a <lexeme> denoting an arithmetic or string value.

Syntax:

```
<literal constant>::= <bit-string constant>|
     <character-string constant>|<arithmetic constant>
```

The character-string used to represent a <literal constant> in Multics PL/I cannot be more than 256 characters long, including quotation marks and final "b" character, if any.

2.6.2.1 Bit-string constants

A <bit-string constant> denotes the bit-string value formed by converting the characters contained within quotes to bits according to the following table and then concatenating that value to itself N-1 times, where N is the value of the <decimal integer>. N must be greater than 0.

<character>	Bit Value for <character> by <radix factor>			
	b,b1	b2	b3	b4
0	0	00	000	0000
1	1	01	001	0001
2	--	10	010	0010
3	--	11	011	0011
4	--	--	100	0100
5	--	--	101	0101
6	--	--	110	0110
7	--	--	111	0111
8	--	--	--	1000
9	--	--	--	1001
a	--	--	--	1010
b	--	--	--	1011
c	--	--	--	1100
d	--	--	--	1101
e	--	--	--	1110
f	--	--	--	1111
Other	--	--	--	--

Note: -- indicates that the corresponding <character> is invalid for this <radix factor>.

Syntax:

```
<bit-string constant>:::=
     [(<decimal integer>)]"[<character>]..."<radix factor>

<radix factor>:::= {b|b1|b2|b3|b4}
```

In Multics PL/I, the value of an expanded <bit-string constant> cannot be more than 253 bits long.

A null bit-string value is denoted by ""b.

Examples:

```
"01011"b
"1"b
""b
"0247"b3
"0e5"b4
(3)"1"b
```

The last example is equivalent to "111"b.

This page intentionally left blank.

```
<binary constant>::= <binary number>[<scale type><exponent>]b[p]

<binary number>::= <binary integer>[.[<binary integer>]]¦
                   .<binary integer>

<binary integer>::= <binary digit>...

<binary digit>::= 0¦1
```

The <exponent> of a <binary constant> is written as a <decimal integer> and denotes a power of two. The <exponent> of a <decimal constant> denotes a power of ten.

The arithmetic value denoted by an <arithmetic constant> must lie within the range allowed by the maximum precision supported for the data type and base of the constant. See paragraph 4.1.5 for a precise description of the range of values supported for each data type and base.

For definitions of "p", "e", "i", and "f" in an <arithmetic constant>, see Section 5.2.6.

Examples:

```
47              .3
101b            1p
25.7            1bp
10.30           1bpi
07.20           8.64f10
10.24e3         1f18b
12.1e+5
101.101e+5b
25i
```

2.6.2.2 Character-String Constants

A <character-string constant> denotes the character-string value formed by replacing all double quotes by a single quote, removing the containing quotes, and concatenating the value to itself N-1 times, where N is the value of the <decimal integer>. N must be greater than zero.

Syntax:

```
<character-string constant>::= [(<decimal integer>)]
     "[<character>]..."

<character>::= ""| Any ASCII character except a quote
```

In Multics PL/I, the value of an expanded <character-string constant> cannot be more than 254 characters long.

A null character-string value is denoted by "".

Examples:

```
"abc"
"This is a character-string constant"
(25)" "
""
"he said, ""I don't know"""
```

2.6.2.3 Arithmetic Constants

An <arithmetic constant> denotes an arithmetic value of a given type, base, mode, and precision. The type, base, mode, and precision are known as <attribute>s of the constant and are normally determined by the syntax of the constant. Refer to Section 5 for a discussion of the declaration of constants.

Syntax:

```
<arithmetic constant>::= <real constant>|<imaginary constant>

<imaginary constant>::= <real constant>i

<real constant>::= <decimal constant>|<binary constant>

<decimal constant>::= <decimal number>[<scale type><exponent>]
                      [p]

<decimal number>::= <decimal integer>[.[<decimal integer>]]|
                    .<decimal integer>

<decimal integer>::= <digit>...

<digit>::= 0|1|2|3|4|5|6|7|8|9

<scale type>::= e|f

<exponent>::= [+|-]<decimal integer>
```

2.6.3 Isubs

An <isub> is a <lexeme> used only in a <subscript> of a <base reference> of a <defined attribute>. Its semantics are described in paragraph 4.3.3.4.

Syntax:

```
<isub>::= <decimal integer>sub
```

Example:

```
5sub
```

2.6.4 Delimiters, Blanks and Comments

The <identifier>s, <arithmetic constant>s, and <isub>s of an <external procedure> are separated from one another by one or more <delimiter>s.

Syntax:

```
<delimiter>::= <graphic delimiter>|<space>|<comment>
     <macro>|<bit-string constant>|
     <character-string constant>|"<picture>"

<graphic delimiter>::= +|-|*|/|**|^|&|||||:|;|(|)|,|.|->|
     =|^=|<|^<|>|^>|<=|>=

<space>::= <blank>|<newline>|<tab>|<newpage>

<blank>::= ASCII blank character

<newline>::= ASCII newline character

<tab>::=  ASCII horizontal tab or ASCII vertical tab

<newpage>::= ASCII newpage character

<comment>::= /* ASCII characters except an asterisk
     followed by a slash       */
```

There is no restriction on the length of a <comment> or on the number of <space>s used as a <delimiter>.

The higher level syntax rules do not indicate where <space>s, <comment>s, and <include macro>s can be used. They can be freely used between any two <lexeme>s. Where the high-level syntax rules show two adjacent <identifier>s, <arithmetic constant>s, or <isub>s, at least one <space> or <comment> is required to separate them.

Examples:

```
a+b+7
do i = 1 to 10;
do i = 1 to/* upper limit */10;
declare a bit(19),b pointer;
/* This is a comment */
```

2.7 Compile-Time Macros

<macro>::= <include macro>|<page macro>|<skip macro>

2.7.1 Include Macro

Syntax:

```
<include macro>::= %include<space>...
     {<identifier>|<character-string constant>};
```

The compiler replaces each <include macro> with the contents of the segment whose name is formed by appending ".incl.pl1" to the <identifier> or <character-string constant>. The segment is searched for by using the "translator" search list, which has a synonym of "trans". (See the add_search_paths command in the MPM Commands and Active Functions manual.)

The replacement of <include macro>s is performed during the application of the lexical-level syntax rules and, consequently, has no effect on the high-level syntax rules. The replacement is performed from left-to-right.

After the replacement is performed, the scan resumes at the beginning of the included text; therefore, the included text may contain <include macro>s. The text that results from the expansion of all <include macro>s must be a valid <external procedure> as described by the syntax rules. Refer to the Multics PL/I Reference Manual, Order No. AM83 for a discussion of segments and segment names.

Example:

```
declare p pointer;
%include T;
declare f fixed;

becomes

declare p pointer;
declare 1 record, 2 field1, 2 field2;
declare f fixed;
```

Where "declare 1 record, 2 field1, 2 field2;" is the contents of a segment whose name is "T.incl.pl1".

2.7.2 Page Macro

Syntax:

```
<page macro>::= %page[(<decimal integer>)];
```

The effect of the <page macro> is to continue the listing of the source program on a new page.

The compiler deletes each <page macro> from the text of the program, so it has no effect on the meaning of the program.

Let N be the value of <decimal integer>. If <decimal integer> is not specified, let N be 1. The <page macro> inserts N newpage characters into the listing.

2.7.3 Skip Macro

Syntax:

```
<skip macro>::= %skip[(<decimal integer>)];
```

The effect of the <skip macro> is to continue the listing of the source program after inserting one or more blank lines.

The compiler deletes each <skip macro> from the text of the program, so it has no effect on the meaning of the program.

Let N be the value of <decimal integer>. If <decimal integer> is not specified, let N be 1. The <skip macro> inserts N newline characters into the listing.

SECTION 3

DYNAMIC BEHAVIOR OF A PL/I PROGRAM

3.1 Flow of Control

A PL/I program is executed by a processor, or control, that follows a path through the program known as the flow of control.

The program determines the flow of control by the use of <goto statement>s, <if statement>s, <call statement>s, <function reference>s, <begin block>s, and <group>s. A program cannot control the real time rate at which it is executed and it cannot create multiple paths for control to follow simultaneously.

3.2 A Multics PL/I Program

A PL/I program is a set of <external procedure>s and their operating environment. The set of <external procedure>s that constitute a program in Multics PL/I is dynamically determined by the execution of the program. When an <external procedure>, A, is first referenced within a process or run unit, it becomes part of the program. Subsequent calls to A, or to any entry of A, invoke the <external procedure> incorporated in the program by the first reference to A. Refer to the Multics PL/I Reference Manual for a brief discussion of Multics dynamic linking and name resolution.

Throughout this document, a Multics process, exclusive of contained run units, is considered to be a single PL/I program and a control. The control begins executing the program when the process is created. A process is either in a state of execution or is waiting to be executed. A waiting process is blocked. The process may be blocked at the discretion of the operating system or as a result of explicit calls to Multics procedures. The blocking of a process has no effect on the subsequent execution of the process except to delay its execution in real time.

A Multics run unit, which is a separate environment similar to, but contained in, a process, is also considered to be a single PL/I program and a control. A process may cause a run unit to be activated; its execution resumes upon termination of the run unit.

When a process or run unit terminates, all files opened during its execution, and remaining open, are closed, unless termination is due to partial destruction of the process or run unit, or unless termination is due to exhaustion of process resources.

3.3 Dynamic Block Structure

3.3.1 Block Activation

A <block> is activated when control enters the <block>. It remains active until control returns from the <block>. At least one <block> is always active, the first one that control entered. Since <block>s may be nested and <procedure>s may call each other, several <block>s may be active. The order in which the <block>s were activated determines the dynamic relationship between the <block>s.

If control passes from an active <block> A to <block> B, A is said to be the dynamic predecessor of B, and B is the dynamic descendent of A.

A block activation is a given activation of a given <block>. An activation record is a unit of storage allocated for a block activation. This unit of storage contains information needed by control in order to execute the <statement>s in the <block> and is the place where all automatic variables declared in the <block> are allocated. Refer to paragraph 4.3.2.2 for a discussion of storage allocation for automatic variables. Label, format, and entry values contain as part of their value a pointer to an activation record. Refer to paragraph 4.1 for a discussion of data types.

3.3.2 Environment of a Block Activation

Every block activation has a parent pointer. A parent pointer is a pointer to an activation record of a <block>'s immediately containing <block>. Since <external procedure>s have no containing <block>, their parent pointer is null.

When control references automatic variables, defined variables, parameters, label constants, format constants, or entry constants declared in a containing <block>, control must know which of several possible activation records of the containing <block> it is to reference. The parent pointer of a block activation points to the correct activation record of its immediately containing <block>. When a reference is made from within a <block> through several containing <block>s, the parent pointer of each <block> points to the correct activation record of its immediately containing <block>.

Example:

```
P: procedure;
   declare A fixed automatic;
   declare I entry external static;
   if first_invocation then I = Inner; else call E;
   call F;

   Inner: procedure;

        A = A*2;
```

Assume that P calls F, and F calls P, then P calls E, and E calls I. The order of block activations is P,F,P,E,I. When I references A, it must select the correct activation record of P so that it can reference the correct instance of A. In this case, the correct activation record of P is the first activation record of P, because it was that activation that created the entry value I by assigning the internal entry constant Inner to I.

The parent pointer of an activation of a <begin block>, other than <begin block>s used as <on unit>s, is a pointer to the most recent activation record of the <block> that immediately contains the <begin block>. Because a <begin block> cannot be invoked except by the <block> which immediately contains it, the activation record pointed to by the parent pointer is also the immediate dynamic predecessor of the <begin block> activation.

The parent pointer of a <procedure> block activation is the activation record pointer part of the entry value used to invoke the procedure. .Refer to paragraph 4.1.11 for a discussion of entry data.

The parent pointer of an activation of an <on unit> is a pointer to the activation record of the block activation that established the <on unit>. See paragraph 3.6.3 for a discussion of <on unit>s and the flow of control.

This page intentionally left blank.

3.4 Flow of Control Within a Block Activation

The flow of control within a block activation proceeds from <statement> to <statement> in the order in which the <statement>s appear in the text of the <block>, except as influenced by actions of a <statement>.

The order in which the components of a <statement> are evaluated is defined in Section 12 where the syntax and semantics of each type of <statement> are defined. The order of evaluation of <expression>s is given in Section 7.

The flow of control within a <group> is specified by the <do statement> which begins the <group> and by <statement>s within the <group>.

A <goto statement> transfers control to any labeled <statement> within the <block> by referencing a name declared by a <label prefix> appearing on any <statement>, other than a <format statement>, <entry statement>, or <procedure statement>, within the <block>.

A <goto statement> also transfers control to a <statement> within the current block activation if it references a label variable or label-valued function that identifies a <statement> within the <block>, but only if the activation record pointer part of the label value points to the current block activation record. Refer to paragraph 4.1 for a discussion of data types.

3.5 Local and Nonlocal Goto Statements

A <goto statement> that transfers control to another <statement> within the same block activation is known as a local goto. A <goto statement> that transfers control to a <statement> in a dynamically preceding block activation is a nonlocal goto. It is an error for a <goto statement> to attempt to transfer control to a <statement> within an inactive <block>.

3.6 Inter-Block Flow of Control

3.6.1 Begin Blocks

Control enters a <begin block> by passing through the <begin statement> which heads the <block>. Control returns from a <begin block> by passing through the <end statement> that terminates the <block>, or by the execution of a <return statement> or a nonlocal goto. A <begin block> cannot be invoked by <function reference>s or <call statement>s. A <label prefix> on a <begin statement> defines a label constant, not an entry constant.

When control returns from a <begin block> by execution of the <end statement>, it returns to the dynamically preceding <block>; which is always the <block> immediately containing the <begin block>. Execution continues with the <statement> following the <end statement>.

If a <return statement> within a <begin block> is executed, it returns control to the dynamic predecessor of the most recent <procedure> block activation.

Example:

```
X: procedure;
     begin;
          begin;
          return;
          end;
     end;
   end;
```

Execution of the <return statement> in this example returns control to the block activation that invoked X.

3.6.2 Procedures

Control enters a <procedure> when one of its entries is invoked by a <function reference> or <call statement>. Control returns from the <procedure> by the execution of a <return statement>, the execution of the <end statement> which terminates the <block>, or by the execution of a nonlocal goto.

Execution of the <statement> which invoked the <procedure> is incomplete if the <procedure> returns via a nonlocal goto. Such incomplete executions are in error only if the invocation resulted from the evaluation of an <initial attribute> or <extent expression> of an automatic or defined variable, and only if control returned to the <block> in which the variable was declared. Refer to paragraph 4.3.2.

A <procedure> invoked as a subroutine by a <call statement> cannot return control by the execution of a <return statement> that contains a <return value>.

A <procedure> invoked as a function by a <function reference> must return control by the execution of a <return statement> containing a <return value>, or it must return by the execution of a nonlocal goto.

If control reaches a <procedure statement>, except as a result of an invocation of the <procedure>, it passes around the <procedure> and continues with the execution of the <statement> following the <procedure>.

3.6.3 On Units

Syntax:

<on statement>::= [<prefix>]on<condition list>[snap]<on unit>

<on unit>::= <begin block>|<independent statement>|system;

<condition list>::= <condition name>[,<condition name>]...

The execution of an <on statement> causes the <on unit> to be established, but does not cause execution of the <on unit>. An established <on unit> is associated with the block activation that contains the <on statement>.

Control enters an established <on unit> when one of the conditions identified by the condition list is signalled. A condition is signalled by the execution of a <signal statement> or by detection of the condition during program execution.

Control returns from an <on unit> by the execution of a nonlocal goto or when control reaches the end of the <on unit>. An <on unit> consisting of an <independent statement> behaves as if it were a <block>, and the execution of an <on unit> is effectively a block activation. The parent pointer of an activation of an <on unit> is a pointer to the activation record of the block activation that established the <on unit>. A complete discussion of conditions is given in Section 10.

SECTION 4

DATA OF PL/I

4.1 Data Types

4.1.1 Representation of Data

Each value is a member of only one set of values called its data type. The data type of a value determines how that value is stored within the computer, and determines which operations can be performed on the value.

The internal representation of data is not defined by the language and no feature of the language, except the "unspec" and nonstandard Multics built-in functions, depend on it. Refer to the Multics PL/I Reference Manual, Order No. AM83, and to the MPM Reference Guide, Order No. AG91, for a description of the internal representation of PL/I data.

Only arithmetic and string values have an external character-string representation defined by PL/I. The external representation of arithmetic and string data is a string of characters formed according to the syntax rules given in paragraph 2.6.2 for <literal constant>s, or produced by arithmetic to character-string conversion as described in Section 8.

4.1.2 Constants

A constant is a value that cannot change during program execution. A constant is either a <literal constant> or a named constant. A <literal constant> is a constant whose lexigraphical representation in the text of an <external procedure> denotes its value. <bit-string constant>s, <character-string constant>s, and <arithmetic constant>s are <literal constant>s and are defined in paragraph 2.6.2. A named constant is a constant whose value is represented in the text of an <external procedure> by a <reference> to a <declared name> declared with the <constant attribute>. Label, format, entry, and file constants are named constants. Refer to Section 5 for a discussion of declarations.

4.1.3 Variables

A variable is a named object capable of representing different values all having the same data type. Because variables are restricted to representing values of a given data type, they are characterized by their data type and are referred to as: bit-string variables, fixed-point variables, etc.

Since values are stored in variables, a variable must own sufficient storage to contain any value that it may represent. A variable's storage is its generation of storage. Section 4.3.2 describes how storage is allocated for variables.

4.1.4 Data Types of Expressions and Functions

Although <literal constant>s and <reference>s are simple forms of <expression>s, throughout this section we will use expression to denote either an infix expression or a prefix expression as described in Section 7.

Expressions and functions are restricted to computing values of a given data type.

The data type of the values of an expression is determined by the rules of expression evaluation given in Section 7 and by the rules of data type conversion given in Section 8. The data type of the values returned by a function is determined by the <returns attribute> specified in the declaration of the function. In the following sections, functions are described as having a data type. This is a convenient way of referring to the data type of the values returned by the function. Refer to Section 5 for a discussion of declarations.

4.1.5 Arithmetic Data

An arithmetic value is either a fixed-point value or a floating-point value. The data type of an arithmetic value is completely specified by four properties: The mode, which may be complex or real, the base, which may be binary or decimal, the type, which may be fixed-point or floating-point, and the precision.

The precision of a fixed-point value is (p,q), where p is the total number of binary or decimal digits in the number, and q is a scale factor giving the location of the implied decimal or binary point. A positive scale factor means that the point is located q places to the left of the rightmost digit. A negative scale factor means that the point is located q places to the right of the rightmost digit. Thus, a fixed-point value can be considered to be the implied product of an integer of p digits times b**-q, where b is the base of the value, (10 or 2).

The precision of a floating-point value is (p), where p is the minimum number of binary or decimal digits that are to be maintained in the mantissa. A floating-point value consists of a mantissa and an exponent. In Multics PL/I, a binary floating-point value has a mantissa that is a binary fraction, f, whose absolute value is $(1/2) \leq f < 1$ or is zero, and whose exponent, e, is an integer whose value is $-128 \leq e \leq 127$. The binary floating-point value is f*2**e. A decimal floating-point value has a mantissa that is an integer, m, whose value is in the range $\pm((10**p)-1)$, and an exponent, e, that is an integer whose value is $-128 \leq e \leq 127$. The decimal floating-point value is m*10**e.

In Multics PL/I, the precision of floating-point binary data is restricted to no more than 63 binary digits. The precision of fixed-point binary data is restricted to no more than 71 binary digits, and the precision of decimal data, either fixed-point or floating-point, is restricted to no more than 59 decimal digits. The scale factor is restricted to $-128 \leq q \leq 127$.

When necessary to avoid loss of significant digits, a decimal floating-point value is normalized such that the most significant digit of the mantissa is nonzero. A binary floating-point value is always normalized as a binary fraction whose most significant digit is nonzero, unless the entire value is zero. The overflow condition occurs when a computation or conversion develops a floating-point value whose exponent exceeds 127, and the underflow condition occurs when a computation or conversion develops a floating-point value whose exponent is less than -128. Refer to Section 10 for a discussion of conditions.

The rules of PL/I arithmetic are such that computations on fixed-point values produce true arithmetic results, except for fixed-point division which truncates low order digits. Computations on floating-point values produce floating-point results that preserve at least the most significant p digits of the true arithmetic result. Refer to Section 7 for a discussion of PL/I arithmetic.

Arithmetic variables and arithmetic-valued functions are declared with the <fixed attribute> or the <float attribute>, a <binary attribute> or a <decimal attribute>, a <real attribute> or a <complex attribute>, and a <precision attribute> as described in Section 5.

4.1.6 String Data

A string value is either a bit-string value or a character-string value. A bit-string value is a sequence of bits, and a character-string value is a sequence of ASCII characters.

The number of characters or bits in the value is the current length of the string value. A bit-string value with no bits is a null bit-string, and a character-string value with no characters is a null character-string.

A string expression or string-valued function can yield string values whose lengths differ each time the expression or function is evaluated. The value is either always a bit-string or always a character-string.

String variables, however, have a maximum length that is determined when storage is allocated for the variable. A nonpictured string variable is declared with either the <varying attribute> or the <nonvarying attribute>. These <attribute>s determine the way that string values are assigned to string variables. String variables and string-valued functions are declared with either the <bit attribute>, <character attribute> or <picture attribute> as described in Section 5.

A variable declared with a <picture attribute> is a pictured character-string variable. It differs from nonpictured character-string variables only in the way values are assigned to it and in the way its values are converted. A function declared to return a pictured value returns a character-string value. That value differs from other character-string values only in the way it is converted. The length of a pictured character-string value can not exceed 64 characters.

Refer to Section 8 for a discussion of conversion. Refer to paragraph 2.6.2.1 for the syntax of a <bit-string constant> and to paragraph 2.6.2.2 for the syntax of a <character-string constant>.

4.1.7 Locator Data

A locator value identifies a generation of storage of a variable. A locator is analogous to, but not necessarily the same as, a machine address. A locator can identify the storage of any variable, regardless of its data type or its relationship to its containing aggregate. The null locator value is a unique value that does not identify a generation of storage. It can be assigned to locator variables and can be used in locator comparison.

A locator loses its validity when the generation of storage it identifies is freed. Such locators do not automatically receive the null locator value.

It is an error to use an invalid or null locator as a <locator qualifier> in a <reference> to a based variable. Refer to paragraph 6.6 for a discussion of <locator qualified reference>s.

A locator datum has no <literal constant> representation in the text of an <external procedure>, but the null locator value is returned by the null built-in function.

There are two types of locator data in PL/I: pointer data and offset data.

A pointer value identifies a generation of storage within any storage class. In addition to the situations described previously in this section, a pointer value is invalid when it is used as a <locator qualifier> or in a comparison operation within a process other than the process that created it. If a pointer value is created within a run unit, it may only be used within that run unit.

An offset value identifies the storage generation of a based variable allocated within an area variable. An offset value is a relative locator value identifying the generation of storage with respect to the area variable in which the generation is allocated.

An offset value is valid when used in any Multics process or run unit that also has valid access to the area variable.

Locator variables and locator-valued functions are declared with the <pointer attribute> or <offset attribute> as described in Section 5.

4.1.8 Area Data

An area value is a generation of storage in which based variables can be dynamically allocated by the execution of an <allocate statement>. In Multics PL/I, an area has a size that is the number of 36-bit words occupied by the generation. The amount of space available within an area in Multics PL/I and the amount occupied by each generation allocated within an area are given in the MPM Subsystem Writer's Guide.

Areas maintain their validity when accessed in a Multics process other than the process in which they were created. In order for the process to access the generations allocated within an area, offset locator values must be used because pointer values are invalid when used in a process other than the process that created them.

Area variables and area-valued functions are declared with an <area attribute> as described in Section 5.

4.1.9 Label Data

A label constant identifies a <statement> within the text of an <external procedure>.

An <identifier> is declared as the name of a label constant by appearing as a <declared name> in a <label prefix> on any <statement> other than an <entry statement>, <procedure statement> or <format statement>. Refer to Section 5 for a discussion of declarations.

A label constant is transformed into a label value each time it is referenced during program execution. A label value is, therefore, always derived from a label constant. A label value identifies the same <statement> as the label constant from which it was derived, but it also points to an activation record. The activation record pointed to by a label value is determined when the label constant is transformed into a label value.

When a label constant is transformed into a label value by the evaluation of a <reference> in the same <block> in which the label constant is declared, the activation record pointer assigned to the label value points to the activation record of the current block activation.

When a label constant is transformed into a label value by the evaluation of a <reference> in a <block> contained within the <block> in which the label constant is declared, the activation record pointer assigned to the label value points to the first activation record of the declaring <block> found by following the parent pointer of the block activation making the transformation. Refer to paragraph 3.3 for a discussion of block activation and the parent pointer.

A label value retains its validity only as long as the block activation record that it points to remains active. It is an error to reference a label value that has lost its validity.

Both the <statement> identification and the activation record pointer values of a label value are used in label value comparison. Two label values compare equal only if they identify the same <statement> and the same activation record.

Label variables and label-valued functions are declared with the <label attribute> as described in Section 5.

4.1.10 Format Data

A format constant identifies a <format statement> within the text of an <external procedure>.

An <identifier> is declared as the name of a format constant by appearing as a <declared name> in a <label prefix> on a <format statement>. Refer to Section 5 for a discussion of declarations.

A format constant is transformed into a format value each time it is referenced during program execution. A format value is, therefore, always derived from a format constant. A format value identifies the same <format statement> identified by the format constant from which it was derived, but it also points to an activation record. The activation record pointed to by a format value is derived in the same manner as that of a label value.

A format value retains its validity only as long as the activation record that it points to remains active. It is an error to reference a format value that has lost its validity.

Both the <statement> identification and the activation record pointer values of a format value are used in format value comparison. Two format values compare equal only if they identify the same <statement> and the same activation record.

Format variables and format-valued functions are declared with the <format attribute> as described in Section 5.

4.1.11 Entry Data

An entry constant identifies an entry point to a <procedure>. An external entry constant identifies an entry point of an <external procedure> and an internal entry constant identifies an entry point of a nested <procedure>.

An <identifier> is declared as the name of an entry constant by appearing as a <declared name> in a <label prefix> on an <entry statement> or <procedure statement>. External entry constants that identify entry points into other <external procedure>s must be declared by a <declare statement>.

An entry constant is transformed into an entry value each time it is referenced during program execution. An entry value is, therefore, always derived from an entry constant. An entry value identifies the same entry point as the entry constant from which it was derived, but it also points to an activation record. If the entry value identifies an external entry point the activation record pointer is null. If the entry value identifies an internal entry point the activation record pointer is determined when the entry constant is transformed into the entry value.

When an internal entry constant is transformed into an entry value by the evaluation of a reference in the same <block> in which the entry constant is declared, the activation record pointer assigned to the entry value points to the activation record of the current block activation.

When an internal entry constant is transformed into an entry value by the evaluation of a <reference> in a <block> contained within the <block> in which the constant is declared, the activation record pointer assigned to the entry value points to the first activation record of the declaring <block> found by following the parent pointer of the block activation making the transformation. Refer to paragraph 3.3 for a discussion of block activation and parent pointer.

It is an error to invoke an internal <procedure> with an entry value that points to an activation record that has been freed. The most common circumstance in which this occurs is when an internal entry constant is assigned to an entry variable by an activation of the <block> in which the entry constant was declared, and control returns from the block activation that made the assignment. Subsequent use of the entry variable to invoke the internal entry is an error because the activation record pointed to by the entry value was freed by the return.

Both the entry point and the activation record pointer values of an entry value participate in entry value comparison. Two entry values compare equal only if they identify the same entry point and the same activation record.

External entry constants, entry variables and entry-valued functions are declared with the <entry attribute> as described in Section 5.

4.1.12 File Data

A file value identifies a file-state block. A file constant always identifies the same file-state block, but a file variable can identify any file-state block. A file-state block is a composite value that defines the relationship between the program and a data set.

A program has as many file-state blocks as it has file constants. A file value can be assigned to a file variable and functions can return file values. File description attributes can only be declared for file constants because they are properties of the file-state block and not properties of the file value.

A file-state block includes:

1. File description attributes.
2. The open/closed status.
3. Line size.
4. Page size.
5. Page number and line number.
6. The column position.
7. Record designators and stream position.
8. A data set designator (title).

The components of a file-state block may change during program execution as the relationship between the program and the data set changes.

A file-state block identifies a data set by the value of the title or data set designator. The value is set when a file is opened and remains unchanged until the file is closed. It is possible for a given file-state block to identify several data sets during program execution; but in any given opening, the file-state block identifies a single data set.

File constants, file variables, and file-valued functions are declared with a <file attribute> as described in Section 5. Refer to Section 11 for a discussion of input/output and a more detailed description of file-state blocks.

4.2 Aggregates of Data

Each of the data types discussed in paragraph 4.1 is the data type of a scalar value. An aggregate value is a set of scalar values stored as an ordered sequence. An aggregate value is either an array of scalar values, a structure containing scalar and/or aggregate values, or an array of structure values.

Named constants, variables, functions, and expressions can have aggregate values.

The data type of an aggregate value is the ordered set of data types of its scalar components. The aggregate type of an aggregate variable, named constant, or function value is the dimensionality and array-extents specified by the <dimension attribute> and the structuring specified by the <level>s used in its declaration. The <level>s are adjusted so that they are minimal and each array-extent is given by H-L+1, where L is the lower <bound> and H is the upper <bound>. Refer to Section 5 for a discussion of <level>s and <bound>s.

The aggregate type of an expression is the dimensionality, array-extents, and structuring determined by the rules of expression evaluation given in Section 7 and by the rules of aggregate promotion given in Section 9.

The aggregate type of a <reference> to a structure variable contained in an <assignment statement> containing a <by-name option>, but not contained in a <locator qualifier>, <subscript>, or <argument list> is determined according to the rules given in paragraph 12.2.

4.2.1 Arrays of Scalars

An array of scalars is an n-dimensional set of scalar values that all have the same data type. The scalar components of an array are elements of the array, and are identified by their position within the array by subscripts. For example, The (i,j)th element of a two-dimensional array is in the ith position of the 1st dimension and the jth position of the 2nd dimension. Refer to paragraph 6.2 for a discussion of <subscripted reference>s.

The elements of an array are stored as an ordered sequence in row-major order. This means that when the elements are accessed in the order in which they are stored, the rightmost subscript varies most rapidly and the leftmost subscript varies least rapidly.

Named constants, variables, functions, and expressions can have arrays of scalars as values. Only named constants and variables can be subscripted.

This page intentionally left blank.

The array-extent of each dimension of an array variable is determined when storage for the array variable is allocated. The array values assigned to the variable must have the same aggregate type as the variable, that is they must have the same number of dimensions and the same array-extents as the variable had when it was allocated.

All array values yielded by a given expression or function have the same number of dimensions, but the array-extent of each dimension may, in some cases, change from one evaluation to the next.

4.2.2 Structures

A structure is a hierarchically ordered set of scalar and aggregate values that do not necessarily have the same data type. The immediate components of a structure are members of the structure and are ordered from left to right. The outermost structure is the major structure, and nested structures are substructures.

Variables, functions, and expressions can have structure values, but only the members of variables can be referenced by name. Members of structure variables are referenced by <structure qualified reference>s as described in paragraph 6.4.

The hierarchical order of a structure variable or function value is specified by means of <level>s similar to the section numbers used in this manual. The outermost structure is known as the level-one structure, its members are known as level-two members. If one of the level-two members is a substructure, its members are level-three members, etc.

All structure values assigned to a structure variable must have the same aggregate type as the variable. All structure values yielded by a given expression or function have the same aggregate type, except that the array-extent of each dimension may, in some cases, change from one evaluation to the next.

4.2.3 Arrays of Structures

An array of structures is an n-dimensional set of structure values each of which has identical structuring and identical data types. The elements of an array of structures are known by their position within the array and are referenced with subscripts as are elements of arrays of scalars. Like the elements of an array of scalars, the elements of an array of structures are stored in row-major order.

Variables, functions, and expressions can have arrays of structures as values. Only the elements of variables can be subscripted.

All array of structure values yielded by a given expression or function have the same aggregate type, except that the array-extent of each dimension may, in some cases, change from one evaluation to the next.

4.3 Storage of Data

4.3.1 Packing and Alignment of Variables

The <aligned attribute> and the <unaligned attribute> are declared for scalar and aggregate variables as described in Section 5. The precise effect of the <aligned attribute> and the <unaligned attribute> on the packing and alignment of a variable's values in storage is not defined by the language, but the rules governing the use of these <attribute>s are designed to ensure that programs using them will run correctly in different implementations of PL/I on different computers. In the discussion that follows the effects of these <attribute>s on the packing of data in Multics PL/I are described.

Packed and unpacked are terms that describe the representation of values and the efficiency of access of values. Packed and unpacked are not <attribute>s and cannot be used in declarations.

An arithmetic, nonvarying string, or pointer variable declared with the <unaligned attribute> is a packed scalar variable. An aggregate variable consisting entirely of packed scalar and packed aggregate variables and declared with the <unaligned attribute> is a packed aggregate variable.

An arithmetic, nonvarying string, or pointer variable declared with the <aligned attribute> is an unpacked scalar variable. Varying string, offset, entry, label, file and area variables are unpacked scalar variables regardless of which alignment attribute they are declared with.

An aggregate variable containing an unpacked scalar or unpacked aggregate variable, or an aggregate variable declared with the <aligned attribute> is an unpacked aggregate variable. Therefore, an unpacked structure can contain both packed and unpacked members, but a packed structure consists entirely of packed members.

4.3.1.1 Packing and Alignment of Scalar Variables

A packed scalar variable occupies the minimum number of bits necessary to represent its values. If a packed scaler variable is declared with the <unsigned attribute>, no storage is used to store its sign. An unpacked scalar variable is stored in such a way as to facilitate access to its values. In Multics PL/I, unpacked variables are aligned on word or multiple-word boundaries, and occupy an integral number of words.

Example:

```
declare A fixed binary precision(8) unaligned;
declare B pointer unaligned;
declare C fixed binary(38) aligned;
declare D pointer aligned;
```

In Multics PL/I, both A and B are packed scalar variables. A occupies 9 bits of storage and B occupies 36 bits. Both C and D are unpacked scalar variables, each occupies 2 words of storage and each is aligned on an even storage address.

4.3.1.2 Packing and Alignment of Structures

A packed structure, which is not a packed array of structures, contains no unused bits between its members except those bits necessary to make character-string, decimal arithmetic, or pictured values begin on 9-bit byte storage boundaries. A packed structure is aligned on a 9-bit byte storage boundary if any of its members are so aligned.

An unpacked member of a structure is aligned on a storage boundary that facilitates access to the member and occupies an integral number of words of storage. An unpacked structure is aligned on a storage boundary that is the maximum boundary required by any of its members and occupies an integral number of words. A packed member of a structure begins on the next bit or 9-bit byte following the previous member.

Example:

```
declare 1 S,
        2 A bit(6) unaligned,
        2 B character(3) unaligned,
        2 C bit(8) aligned,
        2 D bit(18) unaligned;
```

In Multics PL/I, S is an unpacked structure because it contains an unpacked member C. S occupies 90 bits of three words; the first 6 are occupied by A, the next 3 are unused, the next 27 are occupied by B, the next 36 are occupied by C, and the last 18 are occupied by D. The remaining bits of the third word are unused.

4.3.1.3 Packing and Alignment of Arrays

A member of a dimensioned structure is an array whose dimensionality is that given by its own <dimension attribute>, if it has one, and that supplied by the <dimension attribute>s of all containing structures. If a dimensioned structure contains more than one member at the same structuring level, those members are interleaved arrays because the storage for their elements is interleaved.

Example:

```
declare 1 S(3),2 A,2 B;
```

The elements of A and B are interleaved as follows:

```
A(1) B(1) A(2) B(2) A(3) B(3)
```

An unconnected array is an array whose elements are separated from one another in storage by other values. An interleaved array is an unconnected array. A defined array, a parameter array and an array cross-section may also be unconnected arrays. Isub defining is discussed in paragraph 4.3.3.4 and cross-paragraph references are described in paragraph 6.3.

Example:

```
declare A(3,3);
declare B(3) defined(A(1sub,1sub));
```

The array cross-section A(*,2) and the defined array B are both unconnected arrays because their elements are separated from each other by other elements of the array A.

A connected array is an array whose elements are not separated from one another in storage by other values. A connected array of packed scalar elements contains no unused storage bits between its elements.

Example:

```
declare A(5) character(2) unaligned;
```

In Multics PL/I the array A is a packed connected array and occupies 90 bits of storage.

A connected array of packed structures contains no unused storage between its elements except those bits necessary to make character-string, decimal arithmetic, and pictured values begin on 9-bit byte storage boundaries.

A packed unconnected array of scalars contains no unused storage between its elements, except the storage occupied by the other variables.

A packed unconnected array of structures contains no unused storage between elements, except for the storage occupied by other variables and the storage necessary to ensure that contained character-string, decimal arithmetic, and pictured variables begin on 9-bit byte boundaries.

An element of an unpacked array of scalars or structures begins on a storage boundary that facilitates access to the element and occupies an integral number of words of storage.

4.3.1.4 Sign Types

Real arithmetic data may be stored in variables with or without a sign. The sign type of a variable determines whether or not it includes a sign. If the variable is declared with the <unsigned attribute>, its sign type is unsigned; otherwise, if the variable is arithmetic, its sign type is signed. If the variable is not arithmetic, it has no sign type. The sign type of a variable affects the amount of storage it occupies only if it is packed.

This page intentionally left blank.

4.3.2 Storage Classes

4.3.2.1 Allocation of Storage

A generation of storage is an ordered sequence of bits of sufficient length to represent all of the values within the range permitted by a variable's data type. Each variable is declared with one of the following storage class attributes: the <automatic attribute>, <static attribute>, <based attribute>, <controlled attribute>, <parameter attribute>, or <defined attribute> as described in Section 5.

A storage class is a mechanism for allocating and freeing generations of storage for a variable. Because each variable has a single storage class, variables are characterized by their storage class and are referred to as: automatic variables, based variables, etc.

Components of a structure are allocated when their containing major structure is allocated. A generation of storage for a structure variable contains a generation of storage for each of its members. A generation of storage for a component of a structure is freed only when the storage of its containing major structure is freed.

The allocation of a generation of storage for a variable consists of performing the following steps in the indicated order.

1. Evaluate each <extent expression> specified in the variable's declaration and convert its value to a real, fixed-point, binary, integer.

2. Determine the amount of storage required by examining the data type, sign type, alignment <attribute>s, and the evaluated extents from step 1.

3. Allocate a generation of storage of sufficient size. If the variable being allocated is not based, associate the newly allocated generation with the name of the variable. If the variable being allocated is based, assign a locator value that identifies the newly allocated generation to the locator variable given by the <set option> of the <statement> that caused this allocation to occur.

4. If the variable is an area, set it to the empty state.

5. If the variable being allocated is based, assign each evaluated extent to the variable identified by its <refer option>, if it has one.

6. Evaluate each <initial attribute> specified in the variable's declaration and assign initial values to the newly allocated generation.

4.3.2.2 Automatic Storage

A generation of storage is allocated for each automatic variable declared in a given <block> each time the <block> is activated. The generation is allocated in the block activation record as described in paragraph 3.3.1. The block activation record is freed when the block activation for which it was allocated is deactivated. Recursive activation of a <block> has the effect of stacking generations of the <block>'s automatic variables.

The <extent expression>s and <initial attribute>s of an automatic variable can contain <expression>s whose values are computable upon block activation. A value is computable upon block activation if it can be computed without referencing any automatic or defined variable declared in the <block>.

The <extent expression>s are evaluated and stored in the activation record. Subsequent references to the generation of the automatic variable use these evaluated extents.

4.3.2.3 Static Storage

A single generation of storage is allocated for each static variable at or before the time that the variable is first referenced within the process or run unit. The generation remains allocated until termination of the process or run unit.

Static variables must have constant <extent expression>s and <initial attribute>s because they may be allocated prior to block activation.

Internal static variables are variables whose scope has been declared internal by the use of an <internal attribute>. Multics PL/I allocates these variables at compile time.

External static variables are variables whose scope has been declared external by the use of an <external attribute>. External static variables are allocated when they are first referenced within a process or run unit. Refer to Section 5 for a discussion of scope, and to the Multics PL/I Reference Manual for a discussion of external storage allocation.

4.3.2.4 Controlled Storage

A generation of storage is allocated for a controlled variable when an <allocate statement> containing an <allocation> containing an <allocation reference> that identifies the controlled variable is executed. The generation is allocated in "system storage". Refer to paragraph 12.1.

The most recently allocated generation of a <free reference> that identifies the controlled variable is executed. All generations of controlled storage not explicitly freed by the execution of a <free statement> are freed upon process or run unit termination. Refer to paragraph 12.13.

The <extent expression>s and <initial attribute>s of a controlled variable can contain <expression>s whose values are computable without depending on any value of the same controlled variable. No <extent expression> or <initial attribute> of any member of a controlled structure can depend on the value of any scalar component of the same major structure.

The evaluated extents used to allocate a given generation of a controlled variable are saved with the generation. These extents are used whenever a reference is made to the generation. The <extent expression> specified in the variable's declaration are not reevaluated for each reference to the controlled variable. If multiple generations are allocated for a given variable, they are stacked such that a reference to the controlled variable always references the most recently allocated generation. When that generation is freed, the next most recently allocated generation becomes the current generation.

4.3.2.5 Based Storage

A generation of storage is allocated for a based variable when an <allocate statement> containing an <allocation> containing an <allocation reference> that identifies the based variable is executed. If the <allocation> contains an <in option> or derived <in option>, the generation is allocated in the storage of the area variable specified by the <in option>; otherwise, the generation is allocated in "system storage". Refer to paragraph 12.1.

A generation of an explicitly allocated based variable is freed when a <free statement> containing a <freeing> containing a <free reference> containing a <locator qualifier> that identifies the generation is executed. Freeing the storage of an area frees the storage of all generations allocated within the area. Generations of based storage allocated within "system storage" and not explicitly freed by the execution of a <free statement> are freed upon process or run unit termination. Refer to section 12.13.

The program is in error if the generation identified by the <locator qualifier> contained in the <freeing> contained in the <free statement> used to free the generation was not allocated by the execution of an <allocate statement> containing an <allocation> containing an <allocation reference> that identified a based variable of identical aggregate type, data type, sign type, alignment, and extents.

The program is also in error if the generation being freed was allocated in an area and the <freeing> contains an <in option> that does not specify the same area. Similarly, the program is in error if the generation was allocated in "system storage" and the <freeing> contains an <in option>.

A based variable is a description of a generation of storage, but no generation is ever directly associated with the name of the based variable. The specific generation of storage accessed by a <reference> to a based variable is specified by a locator-valued <expression> used as a <locator qualifier> in a <locator qualified reference>. Refer to paragraph 6.6.

Syntax:

<locator qualified reference>::= <locator qualifier>->
<based reference>

Example:

```
P->R
```

P identifies a generation of storage whose data type, sign type, alignment, extents, and aggregate type are described by R.

If the value of P was derived from the evaluation of an "addr" or nonstandard Multics built-in function, the generation is an equivalenced based generation. Refer to paragraph 4.3.3.2.

If the value of P was derived from the execution of an <allocate statement>, <locate statement>, or <read statement> containing a <set option>, the generation is an explicitly allocated based generation. Refer to paragraph 4.3.2.1.

Equivalenced based generations are freed when the generation to which they are equivalenced is freed. Explicitly allocated based generations are freed as described in paragraph 4.3.2.1.

The <extent expression>s in the declaration of a based variable are evaluated for each <reference> to the based variable. It is the programmer's responsibility to ensure that these <extent expression>s accurately describe the extents of the generation referenced by the based variable.

The <extent expression>s and <initial attribute>s of a based variable can contain <expression>s whose values are computable without depending on any values of the same generation of the based variable. No <extent expression> or <initial attribute> of any member of a based structure can depend on the value of any scalar component of the same major structure generation, except for the dependencies expressed by a <refer option>.

The <refer option> allows the extent value used by references to the based variable to be defined by the generation being referenced. The <refer option> can only be used in the <extent expression>s of members of based structures. Such structures are self-defined structures.

Syntax:

<extent expression>::= <expression>[<refer option>]

<refer option>::= refer(<reference>)

Constraints:

Evaluation of the <expression> must yield a scalar value suitable for conversion to a fixed-point, binary, real integer. It must also be suitable for conversion to the data type of the variable identified by the <reference> in the <refer option>.

The <reference> in the <refer option> must identify a preceding scalar component of the same major structure that contains the <refer option>.

The variable identified by the <reference> in the <refer option> must not be dimensioned, and thus cannot have any inherited dimensions.

The <reference> in the <refer option> cannot be a <locator qualified reference>.

Example:

```
declare 1  S  based,
        2  K  fixed,
        2  A  char(N refer(S.K));
```

The explicit allocation of a generation of S by the execution of an <allocate statement> causes N to be evaluated to determine the required amount of storage. The storage is allocated and the value of N is assigned to P->S.K where P identifies the generation.

A <locator qualified reference> to S.A uses the value of S.K as the length of S.A.

Example:

P->S.A Q->S.A

This example shows two generations of S.A, each identified by a unique locator value. The length of the first generation of S.A is P->S.K, while the length of the second generation of S.A is Q->S.K.

4.3.3 Storage Sharing

The language provides three mechanisms for sharing a generation of storage among two or more variables.

1. parameters
2. based variables
3. defined variables

All of these mechanisms require that the variables that share a generation of storage have identical data types, sign types, and alignment <attribute>s. This requirement ensures that the variables have identical storage representations.

All three mechanisms provide techniques for sharing a generation of storage that is contained in an aggregate generation without having to share the entire aggregate. For example, scalars can be mapped onto array elements or members of structures, etc.

A variable declared with a <picture attribute> is considered to match only variables declared with equivalent pictures. Two pictures are equivalent if they are translated into identical <normal pictures> as described in paragraph 8.2.12.

```
P = addr(A(3));
```

P->S is a valid <reference> to A(3), and P->S.X is a valid <reference> to A(3).X.

4.3.3.1 Storage Sharing by Parameters

The discussion of argument passing in paragraph 6.10 describes argument passing by-value and by-reference. When a variable is passed by-reference to a parameter, the variable and the parameter refer to the same generation of storage and thus share that generation.

Example:

```
call f(X);                f: proc(Y);
```

During the block activation of f caused by the execution of "call f(X);", X and Y both refer to the same generation of storage.

4.3.3.2 Storage Sharing by Based Variables

Since the locator value identifying a generation of a variable in any storage class can be derived by the use of the addr built-in function, it is possible for a based variable to be effectively equivalenced to a generation in any storage class.

Example:

```
declare A automatic;
declare B based;
P = addr(A);
P->B = 7;
```

The value of A after execution of the last <assignment statement> is seven.

It is also possible for several based variables to be referenced using the same locator value, thus effectively equivalencing all of those based variables to the same generation of storage.

Example:

```
Q->X     Q->Y     Q->Z
```

In this example, the based variables X, Y and Z are equivalenced to the generation of storage identified by Q.

If the referenced generation and the based variable used to reference it satisfy the criteria for simple defining given in paragraph 4.3.3.5, the based variable can access the generation. In this case, the data type, sign type, alignment, aggregate type, and extents must match.

Example:

```
declare 1 A(5),
        2 X,
        2 Y;

declare 1 S based,
        2 X,
        2 Y;
```

If the referenced generation and the based variable used to reference it satisfy the criteria for string overlay defining given in paragraph 4.3.3.6, the based variable can access the generation. In this case, the data types must match, except that pictured data and nonpictured character-string data types are considered to match. The aggregate types do not have to match.

Example:

```
declare A(5) character(1);
declare B character(5) based;
P = addr(A);
P->B = "abcde"
```

The last <assignment statement> in this example sets the array A to the value "abcde". The first element of the array has the value "a" and the last element has the value "e".

A based structure declaration may be used as a description of a portion of the referenced generation without describing the entire generation. If the generation does not meet the criteria for string overlay defining as described in paragraph 4.3.3.6, the based structure must match the referenced generation from left-to-right up to and including all members contained within level-two of the item being referenced.

Example:

```
declare 1 S,      declare 1 T based,     declare 1 X based,
        2 A,              2 A,                   2 A,
        2 B,              2 B,                   2 B,
        3 C,              3 C,                   3 C;
        3 D,              3 D;
        2 E,
        etc

P = addr(S);
```

A <reference> to P->T.B.C is a valid <reference> to S.B.C, but a <reference> to P->X.B.C is not valid because the declaration of X does not describe all of the level-two substructure S.B.

A based variable cannot access the storage of an unconnected array.

A based variable cannot access the storage of a parameter, except during the block activation to which the storage was passed as an argument. For example, it is an error to take the "addr" of a parameter and assign the resulting locator value to static storage and subsequently, in another block activation, use the locator value.

4.3.3.3 Storage Sharing by Defined Variables

The purpose of the <defined attribute> and the <position attribute> is to map a defined variable onto a generation of storage of another variable. Three types of mapping are possible:

simple defining
isub defining
string overlay defining

Syntax:

<defined attribute>::= {defined|def}<base reference>

<base reference>::= (<reference>)|<reference>

<position attribute>::= {position|pos}[(<position>)]

<position>::= <expression>

The <extent expression>s of a defined variable are evaluated upon block activation and saved in the block activation record. Consequently, they must satisfy the criteria given in paragraph 4.3.2.2 for the extents of automatic variables.

Since a defined variable is associated with the generation identified by its <base reference>, it is never allocated and has no <initial attribute>.

The variable identified by the <base reference> cannot be a defined variable or named constant.

The <defined attribute> cannot be specified for members of structures. When specified for a structure, it maps the entire structure onto the generation of storage identified by the <base reference>.

Both the extents of the defined variable and those of the base variable, are used to determine if the subscriptrange, stringrange or stringsize condition has occurred. Refer to Section 10.

The <expression>s contained in the <base reference> are evaluated for each reference to the defined variable. Any <reference>s in the <base reference> are resolved in the <block> in which the defined variable is declared. Refer to Section 6 for a complete discussion of <reference> resolution and evaluation.

4.3.3.4 Isub Defining

Isub defining allows an array to be defined onto another array by means of a programmer-defined mapping between the elements of the defined array and its base array.

The <defined attribute> specifies isub defining if the <base reference> contains any <isub>s in its <subscript>s.

A <subscripted reference> to an element of an isub-defined array is mapped into a <subscripted reference> to the base array by replacing each <isub> in the <base reference> with the ith <subscript> used in the <subscripted reference> to the defined variable. There must be an <isub> for each dimension of the defined variable or the program is in error. Each <subscript> is converted to a binary integer before replacing an <isub>.

Example:

```
declare A(3,3);
declare B(3) defined(A(1sub,1sub));
```

The array B is a three element array whose elements constitute the diagonal of the array A. A <reference> to B(K) is equivalent to a <reference> to A(K,K).

An unsubscripted <reference> to an isub-defined array is equivalent to a cross-section <reference> in which all <subscript>s are asterisks.

Example:

```
declare A(3,3);
declare B(3) defined(A(1sub,1sub));
```

A <reference> to B is equivalent to a <reference> to B(*) which is equivalent to a <reference> to the array formed by the elements A(1,1), A(2,2) and A(3,3).

The <position attribute> cannot be used with isub defining.

The data type, sign type, alignment <attributes>s, string <length>, and <area size> of the defined array must be identical to the data type, sign type, alignment <attributes>s, string <length>, and <area size> of the base array. If the defined variable is a structure, the structuring of the defined variable and the base variable must be identical, and the data types, sign types, alignment <attribute>s, and extents of all members of the defined variable must be identical to the data types, sign types, alignment <attribute>s, and extents of their corresponding members in the base structure.

4.3.3.5 Simple Defining

Simple defining allows a defined variable to share the storage generation referenced by the <base reference>. The data type, sign type, alignment <attribute>s, string <length>, and <area size> of the defined variable must be identical to the data type, sign type, alignment <attribute>s, <string length>, and <area size> of the generation identified by the <base reference>. If the defined variable is a structure the structuring of the defined variable and the base variable must be identical, and the data types, sign types, alignment <attribute>s, and extents of all members of the defined variable must be identical to the data types, sign types, alignment <attribute>s, and extents of their corresponding members in the base structure.

The <defined attribute> specifies simple defining when there are no <isub>s given in the <base reference>, no <position attribute> specified, and the attributes and extents of the defined variable match those of the base variable as described in the previous paragraph.

A <subscripted reference> to a simple-defined array is mapped into a <subscripted reference> to the base array by replacing the jth asterisk in the <base reference> with the jth <subscript> used in the <subscripted reference> to the defined array. There must be as many asterisks in the <base reference> as there are <subscript>s in the <subscripted reference> to the defined array or the program is in error. Each <subscript> is converted to a binary integer before replacing an asterisk.

Example:

```
declare A(3,3);
declare B(3) defined(A(*,2));
```

A <subscripted reference> to B(K) is mapped into a <subscripted reference> to A(K,2).

An unsubscripted <reference> to a simple-defined array variable is equivalent to a cross-section <reference> to the defined array variable in which all of the <subscript>s are asterisks.

Example:

```
declare A(3,3);
declare B(3) defined(A(*,2));
```

The <reference> B is equivalent to B(*), which is equivalent to a <reference> to the array formed from the elements A(1,2), A(2,2), and A(3,2).

4.3.3.6 String Overlay Defining

String overlay defining allows a string variable (aggregate or scalar) to be defined onto the storage of another string variable (aggregate or scalar) such that the defined variable occupies all or part of the storage occupied by the base variable. The <position attribute> specifies the relative position within the storage of the base variable that the defined variable occupies.

There are two types of string overlay defining: bit-string and character-string. Bit string overlay defining allows a bit-string variable to share storage of another bit-string variable while character-string overlay defining allows a character-string variable to share storage with another character-string variable.

The <position attribute> specifies bits when used for bit-string defining and characters when used for character-string overlay defining.

If the criteria for isub defining or simple defining are not met, the criteria for string overlay defining must be satisfied or the program is in error.

The defined variable and the base variable meet the criteria for string overlay defining when the defined variable and the base variable identified by the <base reference> are both unaligned nonvarying string scalars or aggregates of unaligned nonvarying string scalars all having the same type (bit or character). The aggregate types of the defined variable and the base variable need not match. For the purpose of string overlay defining pictured character-string variables are considered to have the same data type as nonpictured character-string variables. The generation of storage identified by the <base reference> must not be smaller than the defined variable.

It is an error to use the <position attribute> in isub or simple defining. If it is omitted from string overlay defining, a position value of one is assumed. It is also an error to use an asterisk in the <base reference> of the <defined attribute> in string overlay defining.

The <expression> in the <position attribute> is evaluated for each <reference> to the defined variable.

Let i be the value of the <expression> in the <position attribute>. Let b be the <base reference>. Let n be length(string(b)). Let d be the <reference> to the defined variable. Let j be length(string(d)). The following inequality must be satisfied: $0 \leq i-1 \leq j+i-1 \leq n$.

Example:

```
declare A(5) character(2) unaligned;
declare 1 B defined(A),
        2 X character(5) unaligned,
        2 Y picture"99999" unaligned;
```

A <reference> to B.X is a <reference> to the first two and one half elements of A and a <reference> to B.Y is a <reference> to the last two and one half elements of A.

SECTION 5

DECLARATIONS

An <identifier> may be used as a keyword or as the name of a variable, named constant, built-in function, generic function, or condition. A given <identifier> can be used both as keyword and as a name. The meaning of a name is determined by a declaration of the name. Each declaration is established in a <block> and is accessible throughout a region of the program known as the scope of the declaration or the scope of the name.

5.1 Scope of a Declaration

The scope of a name is the <block> in which it is declared and all contained <block>s in which the name is not redeclared. Refer to Section 2 for a discussion of program structure.

Note that the above definition of scope does not strictly apply to declarations of members of structures. Refer to paragraph 6.4 for a discussion of the scope of member's names.

A name cannot be declared more than once in a given <block>, except as the name of a structure member. No two members of a structure can have the same name. A declaration that violates either of these constraints is a multiple declaration and is an error.

5.1.1 Internal Scope

A declaration containing the <internal attribute> is the declaration of a name whose scope is internal. The name is known only in the <block> in which it is declared and all contained <block>s, except those <block>s in which it is redeclared.

5.1.2 External Scope

A declaration containing the <external attribute> is the declaration of a name whose scope is external. The name is known in all <block>s in which the same name is declared with the <external attribute> and in all contained <block>s, except those <block>s in which it is redeclared with the <internal attribute>. All declarations of an external name must have equivalent <attribute set>s, and all such declarations refer to the same generation of storage, the same constant, or the same condition.

5.2 Establishment of Declarations

This paragraph describes the transformations made to the text of an <external procedure> during compilation in order to establish complete declarations for all names and <literal constant>s used in the <external procedure>. The transformations are made in this strict order:

1. Each <procedure statement> that has more than one <label prefix> is transformed into a <procedure statement> followed by a sequence of <entry statement>s. Each <procedure statement> or <entry statement> thus generated has one <label prefix> and have identical <parameter list>s and <procedure option>s.

   Each <entry statement> that has more than one <label prefix> is transformed into a sequence of <entry statement>s each of which has one <label prefix>, and all <entry statement>s have identical <parameter list>s and <entry option>s.

2. Each <get statement> that has no <file option> and no <string option> is given a <file option> of the form:

   ```
   file(sysin)
   ```

   Each <put statement> that has no <file option> and no <string option> is given a <file option> of the form:

   ```
   file(sysprint)
   ```

   Each <copy option> without a <reference> is given a <reference> of the form:

   ```
   sysprint
   ```

3. Each <declare statement> is defactored as described in paragraph 5.2.1.1.

4. Each <like attribute> is expanded as described in paragraph 5.2.2.

5. Declarations are established for all names, <literal constant>s and <descriptor>s. These declarations are derived from <declare statement>s, <label prefix>s, <parameter descriptor>s, <returns descriptor>s, <literal constant>s, and <simple reference>s to undeclared names.

6. The <attribute set> of each declaration is completed by evaluating <default statement>s, by applying the language default rules, and by creating <parameter descriptor>s and possible <returns attribute> for each entry declaration produced by a <label prefix>.

7. Each declaration is validated as described in paragraph 5.5.

5.2.1 Declare Statements

Each <declare statement> is processed by the compiler and behaves like a <null statement> when executed.

Syntax:

```
<declare statement>::= [<label prefix>]...{declare|dcl}
          <declaration list>;

<declaration list>::= <declaration component>
          [,<declaration component>]...

<declaration component>::= [<level>]{<declared name>|
          (<declaration list>)}[<attribute set>]
```

<declared name>::= <identifier>

<attribute set>::= <attribute>...

<level>::= <decimal integer>

5.2.1.1 Defactoring of Declare Statements

Each <declare statement> is transformed into a <defactored declare> by performing the following steps in the indicated order:

1. Copy the <level> that appears immediately to the left of the innermost parenthesized <declaration list> to a position immediately to the left of each <declared name> in that <declaration list>.

2. For each <declared name> in that <declaration list>, if the <declared name> is immediately followed by an <attribute set>, copy the <attribute set> that appears immediately to the right of the innermost parenthesized <declaration list> to a position immediately to the right of the <attribute set> immediately following the <declared name>; otherwise, copy the aforementioned <attribute set> to a position immediately to the right of the <declared name>.

3. Remove the <level> and the <attribute set> from the innermost parenthesized <declaration list> and remove its parentheses. If any parenthesized <declaration list>s remain, repeat these steps.

The program is in error if the defactoring of a <declare statement> produces a <statement> whose syntax is not described by that given below for a <defactored declare>.

Syntax:

```
<defactored declare>::= [<label prefix>]...{declare|dcl}
     <defactored declaration>[,<defactored declaration>]...;

<defactored declaration>::= [<level>]<declared name>
     [<attribute set>]

<attribute set>::= <attribute>...

<level>::= <decimal integer>

<declared name>::= <identifier>
```

The syntax of each <attribute> is given in paragraph 5.4.

Example:

```
declare ((a,b pointer)automatic)internal;
```

is equivalent to:

```
declare a automatic internal,
        b pointer automatic internal;
```

Example:

```
declare 1 s, 2(a,b) pointer;
```

is equivalent to:

```
declare 1 s, 2 a pointer, 2 b pointer;
```

5.2.1.2 Multiple Attributes

A given <attribute> may occur more than once in an <attribute set> only if all but one such occurrence consists solely of a keyword. If the syntax of the <attribute> requires anything other than a simple keyword, the <attribute> cannot occur more than once in a given <attribute set>.

Example:

```
declare A entry entry(float),B bit(1) bit(1);
```

The declaration of B is in error while the declaration of A is valid. Both are examples of poor programming style.

The example given in paragraph 5.3.2 on the use of the <default statement> shows how multiple occurrences of a given <attribute> can be useful.

5.2.1.3 Normalization of Levels

If a <defactored declaration> has a <level> greater than one, the preceding <defactored declaration> must have a <level>. If a <defactored declaration> has a <level> equal to one, either it must have a <like attribute> or it must be followed by a <defactored declaration> with a <level> greater than its own. These constraints ensure that a <defactored declaration> with a <level> is either a structure or a member of a structure, and that all major structures have a <level> of one.

A declaration is a level-one declaration if it is not a declaration of a member of a structure. A variable is a level-one variable if it is not a member of a structure.

After defactoring is complete and structuring is established, <level>s are normalized so that the <level> of each structure member is one greater than the <level> of its immediately containing structure. Refer to paragraph 5.2.3.1.3 for a description of structure declarations.

Example:

```
declare 1 S, 4 A, 3 B, 3 C;
```

is normalized to:

```
declare 1 S, 2 A, 2 B, 2 C;
```

5.2.2 Expansion of the Like Attribute

Syntax:

```
<like attribute>::= like<like reference>

<like reference>::= <identifier>[.<identifier>]...
```

The <like attribute> is a macro-attribute expanded by the compiler. It is replaced by a copy of the declarations of all of the members of the structure identified by the <like reference>.

The <like reference> is resolved as if it were a <simple reference> or a <structure qualified reference>. It must identify a structure declared in a <block> that contains the <like reference>. Refer to Section 6 for a discussion of <reference>s.

Within a given <block>, all <like reference>s are resolved before any <like attribute>s are expanded. This ensures that the order in which the declarations are processed by the compiler does not affect the resolution of <like reference>s. The program is in error if the structure identified by the <like reference> was produced by the expansion of a <like attribute> or if it was declared with a <like attribute>. Refer to Section 6 for a discussion of <reference>s.

Example:

```
declare 1 A,2 C,3 E,3 F;
declare 1 D,2 C,3 G,3 H;
begin;
declare 1 A like D;
declare 1 B like A.C;
```

Because the <like reference>s of the <begin block> are resolved before any <like attribute>s in the <begin block> are expanded, the <like reference> A.C is resolved to refer to the declaration of A in the outer <block> and the result of the <like attribute> expansion is:

```
declare 1 A,2 C,3 G,3 H;
declare 1 B,2 E,2 F;
```

The only <attribute>s copied by the expansion of the <like attribute> are those <attribute>s that were explicitly specified in the declaration of the members of the structure identified by the <like reference>. No inherited <dimension attribute>, <aligned attribute> or <unaligned attribute> is copied. Since expansion occurs before <attribute>s are supplied by default, <attribute>s supplied by default are not copied.

Example:

```
declare 1 S(5) based,
        2 A bit(1),
        2 B(7) pointer;

declare 1 T like S auto;
```

The expanded declaration of T is:

```
declare 1 T auto,
        2 A bit(1),
        2 B(7) pointer;
```

A <defactored declaration> containing a <like attribute> must have a <level> and cannot be followed by a <defactored declaration> whose <level> is greater than its own. This constraint ensures that expansion of a <like attribute> produces <defactored declaration>s of all members of the structure.

Any <level>s copied by the expansion of a <like attribute> are adjusted so that they are normalized with respect to the <level> of the <defactored declaration> containing the <like attribute>.

Example:

```
declare 1 X, 2 Y, 2 Z;

declare 1 S, 2 R, 3 T like X;
```

The expanded declaration of T is:

```
declare 1 S, 2 R, 3 T, 4 X, 4 Y;
```

5.2.3 Establishment of Explicit Declarations

5.2.3.1 Declare Statements

After <declare statement>s have been defactored and <like attribute>s have been expanded, a single declaration is established for each <declared name> in each <declare statement>. These declarations are established in the <block> that immediately contains the <declare statement>. The <attribute set> of each declaration contains only those <attribute>s explicitly specified in the <defactored declaration>. Such declarations are known as explicit declarations.

5.2.3.1.1 Declarations of Scalars

A <defactored declaration> that is neither an array declaration as described in paragraph 5.2.3.1.2 nor a structure declaration as described in paragraph 5.2.3.1.3 is a declaration of a scalar variable, scalar constant, builtin function, generic function, or condition name.

5.2.3.1.2 Declarations of Arrays

If the <attribute set> of a <defactored declaration> contains a <dimension attribute> the <declared name> is declared as an array whose dimensionality and <bound>s are given by the <dimension attribute>.

Members of dimensioned structures are arrays whose dimensionality and <bound>s include the dimensionality and <bound>s given in their own <dimension attribute>, as well as those inherited from their containing structures. Unless the member has its own <dimension attribute>, the member acquires the dimensionality and <bound>s of its containing structures, but does not acquire the <dimension attribute> for the purposes of later <default statement> evaluation.

Examples:

```
declare A(10,10);
declare 1 S(10),2 X(10),2 Y;
```

X and A are ten-by-ten two-dimensional arrays, while S and Y are ten element one-dimensional arrays. A <default statement> whose <predicate> contained the keyword "dimension" would apply to the declarations of A, S and X, but not to Y.

5.2.3.1.3 Declarations of Structures

A <defactored declaration> is a declaration of a structure if it has a <level> and is followed by one or more <defactored declaration>s whose <level>s are greater than its own. The <structure attribute> may be explicitly written in the <attribute set> of a structure declaration, but the declaration must also have a <level> and must be followed by one or more <defactored declaration> whose <level> is greater than its own.

A <defactored declaration> with a <level> greater than one is a member of the nearest <defactored declaration> to its left, whose <level> is less than its own. The <member attribute> may be explicitly written in the <attribute set> of a member declaration.

Example:

```
declare 1 S,2 A,2 B;
```

is equivalent to:

```
declare 1 S structure,2 A member,2 B member;
```

5.2.3.2 Label Prefixes

An explicit declaration is also established for each <declared name> that appears in a <label prefix>. The declaration is established in the <block> immediately containing the <label prefix>. The <declared name> appearing in the <label prefix> of an <entry statement>, <procedure statement>, or <begin statement> is declared in the <block> that immediately contains the <procedure> or <begin block>. The declarations produced by <label prefix>s on the <entry statement>s or <procedure statement> of an <external procedure> are established in an imaginary outer <block> that contains the <external procedure>.

Example:

```
A: proc;
B: ____
     C: proc;
     D: ____
     E: entry;
     F: ____
     G: end C;
H: ____
I: end A;
```

The names B,C,E,H,I are declared in <procedure> A, and since they are not redeclared in <procedure> C, their scope includes both A and C. The names D,F,G are declared in <procedure> C and their scope is <procedure> C. The name A is declared in an imaginary outer <block> and its scope includes both <procedure>s A and C.

5.2.3.2.1 Format Constants

A <declared name> appearing in a <label prefix> of a <format statement> is declared in the immediately containing <block> with the <format attribute>, <constant attribute>, and <internal attribute>.

The <label prefix> of a <format statement> cannot contain a <prefix subscript>.

5.2.3.2.2 Label Constants

A <declared name> appearing in the <label prefix> of any <statement>, other than an <entry statement>, <procedure statement>, or <format statement>, is declared with the <label attribute>, <constant attribute> and the <internal attribute>. If the <label prefix> contains a <prefix subscript>, the declaration is given a <dimension attribute> of the form (L:H) where L is the lowest <prefix subscript> used in any occurrence of this name in a <label prefix> within the <block> and H is the highest <prefix subscript> used in any occurrence of this name in a <label prefix> in the <block>. In Multics PL/I, a label constant array cannot have more than one dimension.

Example:

```
A: begin;
L(1): ____
_________
L(-2): ____
_________
L(4): _____
_________
end A;
```

L is declared in <block> A as a label constant array with a lower <bound> of -2 and an upper <bound> of 4.

It is an error to reference any element of a label constant array that is not defined by a <label prefix>.

5.2.3.2.3 Entry Constants

A <declared name> appearing in a <label prefix> of an <entry statement> or <procedure statement> is declared in the immediately containing <block> with the <entry attribute> and <constant attribute>s, either the <internal attribute> or the <external attribute>, and, optionally, with the <returns attribute>, <reducible attribute> or <irreducible attribute>.

The <returns attribute> and the <reducible attribute> or the <irreducible attribute> are copied from the <entry statement> or <procedure statement>. The <external attribute> is supplied if the <label prefix> appears on an <entry statement> or <procedure statement> defining an entry to an <external procedure>; otherwise, the <internal attribute> is supplied.

After <default statement>s have been evaluated and the language default rules applied to all declarations, a set of <parameter descriptor>s (p1,p2,...,pn) is created for each entry declaration produced by a <label prefix>. The <parameter descriptor>s are constructed by examining the declaration of each parameter of the entry. A <parameter descriptor> p(k) produced by this examination contains all of the <attribute>s of the kth parameter in the <parameter list> of the <entry statement> or <procedure statement>, except the <variable attribute> and <parameter attribute>.

The <label prefix> of an <entry statement> or <procedure statement> cannot contain a <prefix subscript>.

After <default statement>s have been evaluated and the language default rules applied to all declarations, the <returns attribute> of each declaration produced by a <label prefix> is copied onto the <entry statement> or <procedure statement> for use during execution of <return statement>s in that <procedure>. The original <returns attribute>, if any, on the <statement> is replaced by this copy.

Example:

```
P: proc(a) returns(pointer);
   ____________
   declare a pointer;
   ____________
   Inner: proc(x) returns(bit(1));
          ____________
          declare x pointer;
          ____________
          end;
   end;
```

The <label prefix> Inner produces a declaration equivalent to:

```
declare Inner entry(pointer) returns(bit(1)) internal;
```

The declaration is established in the <external procedure> P. The <label prefix> P produces a declaration equivalent to:

```
declare P entry(pointer) returns(pointer) external;
```

The declaration is established in an imaginary outer <block> that contains the <external procedure> P.

5.2.4 Establishment of Contextual Declarations

Any name not explicitly declared by a <declare statement> or <label prefix> is contextually declared if it appears in any of the following contexts. Unless otherwise noted, the declaration is established in the <external procedure>.

1. Area: An undeclared name is contextually declared with the <area attribute> and the <variable attribute> if it appears as the <reference> of an <in option> or <offset attribute>.

2. Builtin: An undeclared name is contextually declared with the <builtin attribute> if it appears followed by an <argument list> and is one of the names listed in Section 13. If it is not listed in Section 13 and it appears with an <argument list> or as the <entry reference> of a <call statement>, the program is in error.

3. Condition: An undeclared name is contextually declared with the <condition attribute> if it appears as a <condition name> in a <signal statement>, <revert statement>, or <on statement>.

4. File: An undeclared name is contextually declared with the <file attribute> and the <constant attribute> if it appears as the <reference> of a <file option> or <copy option> of an input/output <statement>, or as the <reference> of an input/output <condition name>.

5. Parameter: A name not declared, except as a structure member, in the <block> in which it is used in a <parameter list> is declared with the <parameter attribute> and <variable attribute>. The declaration is made in the <block> immediately containing the <parameter list>.

6. Pointer: An undeclared name is contextually declared with the <pointer attribute> and the <variable attribute> if it appears as the <locator qualifier> of a <based attribute> or <locator qualified reference>, or if it appears as the <reference> in a <set option>.

5.2.5 Contextually Derived Attributes

Contextual declarations acquire <attribute>s which depend on the context that produced the declaration. Any additional <attribute>s are supplied later when <default statement>s are evaluated and the language default rules applied.

Explicit declarations do not acquire any <attribute>s, other than the <parameter attribute>, from the usage of the name in one of the above mentioned contexts. The <attribute>s of an explicit declaration are acquired when the declaration is established and when defaults are supplied.

An explicit declaration of a name, other than a member of a structure, is given the <parameter attribute> if it appears in a <parameter list> of the <procedure> in which it is declared.

5.2.6 Establishment of Implicit Declarations

A name that is neither explicitly nor contextually declared is implicitly declared in the <external procedure> with no attributes.

Each <descriptor> appearing in an <entry attribute> or <returns attribute> is implicitly declared in the <block> in which the entry declaration is established.

Each <literal constant> is implicitly declared in the <block> that immediately contains it and is given the <constant attribute>. Declarations of <bit-string constant>s are given the <bit attribute>. Declarations of <character-string constant>s are given the <character attribute>. Declarations of <arithmetic constant>s are given the <float attribute> if they contain an e, and are given the <fixed attribute> if they contain an f. They are given the <complex attribute> if they contain an i; otherwise, they are given the <real attribute>. The rest of their <attribute>s are supplied by <default statement>s and by the language default rules. Note that <default statement>s are not applied to <bit-string constant>s or <character-string constant>s. Also, <default statements> are not applied to <arithmetic constants> containing p.

5.3 Completion of Attribute Sets

Unless a declaration was produced by a <declare statement> that explicitly provided all <attribute>s, the declaration has an incomplete <attribute set>. The <attribute set> of each declaration is completed by performing the following steps in the indicated order:

1. If the declaration contains a <precision attribute> containing a <scale factor>, the <fixed attribute> is given to the declaration.

2. If the declared item is a member of a structure and has neither the <aligned attribute> nor the <unaligned attribute>, the <aligned attribute> and <unaligned attribute> of its immediately containing structure are given to the declaration. If the immediately containing structure does not have either of these <attribute>s, the members of the structure acquire one of the alignment <attribute>s from the application of defaults as described in steps 4 and 5.

3. If the declared item is a member of a structure, it is given the <member attribute> and the <internal attribute>. If it is a structure, it is given the <structure attribute>. If the item has a <dimension attribute> or <precision attribute> without a keyword, the keyword is supplied.

4. Beginning in the <block> of declaration, all <default statement>s are evaluated in the order in which they appear in the <block>. When all <default statement>s in a given <block> have been evaluated, the <default statement>s in the immediately containing <block> are evaluated in the order in which they appear in that <block>. This process is continued until the <default statement>s of the <external procedure> have been evaluated.

5. The language defaults are supplied by evaluating the <default statement>s listed in paragraph 5.3.3 as if they were written in a <block> containing the <external procedure>.

6. Each entry declaration produced by a <label prefix> is given a set of <parameter descriptor>s derived from the declaration of the parameters of the entry.

If the entry declaration contained a <returns attribute>, its <returns descriptor> was processed by step 4 as if its <block> of declaration was the <block> that immediately contained the <entry statement> or <procedure statement> from which the declaration of the entry was derived. This ensures that the <attributes> of the <returns descriptor> are those that apply to the inner <block>, not the <block> in which the entry declaration was made.

7. The declaration of each <arithmetic constant> that does not have either the <decimal attribute> or <binary attribute> is given the <decimal attribute>, unless it contains a b, in which case it is given the <binary attribute>. If it has neither the <fixed attribute> nor the <float attribute>, it is given the <fixed attribute>.

   If the declaration of an <arithmetic constant> does not have a <precision attribute>, it acquires the <precision attribute> obtained by converting the source precision to the base and type specified by its <attribute set>. The source precision of an <arithmetic constant> is the number of digits in the mantissa, including leading and trailing zeros. If the <arithmetic constant> does not contain a <scale type> of e, it has a <scale factor> of j-k, where j is the number of fractional digits in the mantissa, or 0 if there are none, and k is the value of the exponent, or 0 if there is none. Refer to paragraph 8.2.10 for a discussion of the conversion rules.

8. Any declaration that has the <area attribute> and does not have an <area size> is given an <area size> of 1024. Any declaration that has the <character attribute> or <bit attribute> and does not have a <length> is given a <length> of 1.

Note that although file description attributes can be added to a declaration by a <default statement>, the file description attribute sets are not fully completed until program execution and that they depend on how the file is opened. Refer to paragraph 11.3 for a discussion of file opening. The file description attributes are: input, output, update, record, stream, sequential, direct, keyed, print, and environment.

5.3.1 Default Statement

The <default statement> enables the programmer to determine what <attribute>s shall be supplied to declarations whose <attribute set>s are incomplete. It allows <attribute>s to be supplied on the basis of the <attribute>s already acquired or on the basis of the spelling of the declared name.

Syntax:

```
<default statement>::= [<label prefix>]...{default|dft}
          {system|none|<user defaults>};

<user defaults>::= (<predicate>){error|<attribute set>[,<attribute set>]...}

<attribute set>::= <attribute>...

<predicate>::= <predicate one>|
          <predicate>|<predicate one>

<predicate one>::= <predicate two>|
          <predicate one>&<predicate two>

<predicate two>::= <predicate three>|^<predicate two>

<predicate three>::= (<predicate>)|<attribute keyword>|
          <range>
```

```
<range>::= range(*)|range(<identifier>)|
           range(<letter>:<letter>)

<attribute keyword>::= <identifier>
```

An <attribute keyword> must be the keyword or abbreviated keyword used to designate any <attribute> except the <like attribute>. Keywords and abbreviated keywords are equivalent.

The <like attribute> cannot be applied by a <default statement> because <like attribute>s are expanded before the application of <default statement>s.

5.3.2 Evaluation of Default Statements

Each <default statement> is evaluated by the compiler and behaves like a <null statement> when executed.

A <default statement> is evaluated by evaluating its <predicate> and if the <predicate> is true with respect to a given declaration, copying the <attribute set>s specified by the <default statement> in left-to-right order into the <attribute set> already acquired by the declaration.

Just as it is possible to write an inconsistent <attribute set> in a <declare statement>, it is possible to produce a declaration with an inconsistent <attribute set> by the use of a <default statement>. The <predicate> of a <default statement> should be sufficiently selective to avoid applying default <attribute>s to declarations that should not receive them. The Multics PL/I compiler copies each <attribute set> of a <default statement> in left-to-right order into the <attribute set> already acquired by the declaration. If the compiler detects inconsistencies between the <attribute>s in an <attribute set> about to be copied and these already acquired by the declaration, it does not copy the <attribute set> into the declaration. If all <attribute set>s specified by a <default statement> are inconsistent with those already acquired by the declaration, the compiler issues a warning diagnostic. Refer to paragraph 5.5 for a precise definition of attribute consistency.

A <predicate> yields a value of "true" or "false" when applied to a declaration. The infix operator "|" yields a value "true" only if either or both of its operands are "true". The infix operator "&" yields a value "true" only if both of its operands are "true". The prefix operator "^" yields a value "true" only when its operand is "false".

Each <attribute keyword> or <range> operand of the <predicate> yields a "true" or "false" value with respect to a given declaration. An <attribute keyword> yields a "true" value only if the declaration contains the <attribute> identified by the <attribute keyword>. A <range> operand yields a value "true" only if the declaration is a declaration of a name whose spelling satisfies the <range> operand.

An exception exists for options; options is not considered to yield "true" if "constant" was specified.

A <range> operand of the form range(*) is satisfied by any name. A <range> operand of the form range(<identifier>) is satisfied only by names which begin with the same sequence of characters as the <identifier> given in the <range> operand. A <range> operand of the form range(<letter>:<letter>) is satisfied only by names whose first letter is in the English alphabetical sequence between and including the first and second <letter>s. The first <letter> must occur in the alphabet before the second <letter> or both must be the same letter.

Note that declarations of <literal constant>s, <parameter descriptor>s or <returns descriptor>s never satisfy a <range> operand because such declarations are not declarations of names.

This page intentionally left blank.

Example:

```
default(bit) bit(1);
default(fixed) binary(15);
declare b bit(5), f entry() returns(bit);
declare a fixed;
```

is equivalent to:

```
declare b bit(5) bit(1), f entry() returns(bit bit(1));
declare a fixed binary(15);
```

Note that the declaration of b is an invalid declaration.

Example:

```
default(range(*)&^(automatic|based|controlled|defined|
     parameter|member|generic|builtin|condition|constant))
     internal static;
declare a pointer;
```

is equivalent to:

```
declare a pointer internal static;
```

The interested reader may wish to study the language defaults as expressed by the <default statement>s given in paragraph 5.3.3.

5.3.2.1 Special Cases of the Default Statement

A <default statement> of the form:

```
default system;
```

causes the language defaults to be applied as if the set of <default statement>s given in paragraph 5.3.3 were written at this point in the <procedure>.

A <default statement> of the form:

```
default none;
```

causes no further defaults to be supplied either from <default statement>s remaining in the program or by the application of language defaults.

A <default statement> of the form:

```
default (<predicate>) error;
```

causes any declaration within the scope of the <default statement> for which the <predicate> is true to be considered in error. The Multics PL/I compiler issues a diagnostic for each such declaration.

5.3.3 Language Default Rules

Entry Defaults

```
default (returns|reducible|irreducible|options) entry;
default (entry&^reducible) irreducible;
```

File Default

```
default(input|output|update|stream|record|print|keyed|direct|
     sequential|environment) file;
```

Arithmetic Defaults

```
default(^(character|bit|pointer|offset|area|label|format|entry|file|
     fixed|float|picture|binary|decimal|real|complex|
     builtin|generic|condition|constant)) fixed binary real;
default((real|complex)&^(picture|float|constant)) fixed;
default((binary|decimal)&^(float|constant)) fixed;
default((fixed|float)&^(complex|constant)) real;
default((fixed|float)&^(decimal|constant)) binary;
default(fixed&binary&^precision&^constant) precision(17,0);
default(fixed&decimal&^precision&^constant) precision(7,0);
default(float&binary&^precision&^constant) precision(27);
default(float&decimal&^precision&^constant) precision(10);
```

String Default

```
default((character|bit)&^(varying|constant)) nonvarying;
```

Scope and Storage Class Defaults

```
default((entry|file)&(automatic|based|static|parameter|
     defined|controlled|member|aligned|unaligned|
     initial) variable;

default((entry|file)&range(*)&^variable) constant;
default(^(constant|builtin|generic|condition)&range(*))
     variable;
default((file|entry)&range(*)&constant&^internal) external;
default(condition) external;
default(^external&range(*)) internal;
default(variable&external&^controlled) static;
default(variable&^(based|controlled|static|defined|parameter|
     member)) automatic;
```

Storage Mapping Defaults

```
default((character|bit|picture|structure)&^(aligned|constant))
     unaligned;
default(^(constant|builtin|generic|unaligned)) aligned;
default ((fixed|float)&^unsigned) signed;
```

Example:

```
declare i fixed;
declare j float;
declare a;
declare X external;
declare E entry returns(fixed);
```

After application of the language defaults, these declarations are:

```
declare i fixed binary real precision(17,0)
     aligned variable automatic internal signed;
declare j float binary real precision(27)
     aligned variable automatic internal signed;
declare a fixed binary real precision(17,0)
     aligned variable automatic internal signed;
declare X fixed binary real precision(17,0)
     aligned variable static external signed;
declare E entry constant external irreducible
     returns(fixed binary real precision(17,0) aligned signed);
```

5.4 Syntax and Semantics of Attributes

The <attribute>s described in this section are used in <attribute sets> of <declare statement>s, <default statement>s, <descriptor>s, and in <open statement>s to describe variables, constants, functions and conditions. The discussion of each <attribute> assumes that <attribute set>s have been completed. See paragraph 5.3 for a discussion of <attribute set> completion. In the discussion of each <attribute>, item refers to a declaration of a name, a <parameter descriptor>, or a <returns descriptor>.

The description of each <attribute> gives constraints that apply to the <attribute>. Section 5.5 gives a concise syntax that shows which <attribute>s can and must appear in the same completed <attribute set>.

5.4.1 Aligned

Syntax:

<aligned attribute>::= aligned

The <aligned attribute> is used in an implementation-defined manner to influence the representation of values in storage. In Multics PL/I, aligned data is allocated on a word or multiple word storage boundary, and the amount of storage is an integral number of words.

When a generation of storage is to be shared or accessed by more than one name, all names used to access the generation must have the same alignment <attribute>. Refer to paragraphs 4.3.1 and 4.3.3.

5.4.2 Area

Syntax:

<area attribute>::= area[(<area size>)]

<area size>::= <extent expression>|*

<extent expression>::= <expression>[<refer option>]

<refer option>::= refer(<reference>)

An item declared with the <area attribute> represents area values whose size is given by the <area size>.

Evaluation of the <expression> of an <extent expression> must yield a scalar value suitable for conversion to a fixed-point, binary, real, integer. If the <refer option> is given the value of the <expression> must also be suitable for conversion to the data type of the variable identified by the <reference> in the <refer option>.

If the item has the <static attribute>, the <area size> must be an unsigned <decimal integer>. If the item has the <parameter attribute> or is part of a <descriptor>, the <area size> must be an unsigned <decimal integer> or an asterisk. If the item does not have the <parameter attribute> or is not part of a <descriptor>, the <area size> cannot be an asterisk. If the item does not have the <based attribute>, it cannot contain a <refer option>. Refer to paragraph 4.3.2.5 for a discussion of based storage and the <refer option>.

5.4.3 Automatic

Syntax:

<automatic attribute>::= automatic|auto

A name declared with the <automatic attribute> is a variable whose storage class is automatic. Refer to paragraph 4.3.2 for a discussion of storage classes.

5.4.4 Based

Syntax:

<based attribute>::= based[(<locator qualifier>)]

<locator qualifier>::= <reference>

A name declared with the <based attribute> is a variable whose storage class is based. Evaluation of the <locator qualifier> must yield a scalar locator value. If the <locator qualifier> is omitted, all <reference>s to the based variable except, <allocation reference>s or the <reference>s of <refer option>s, must be <locator qualified reference>s as defined in paragraph 6.6. All <reference>s to based variables without locator qualification, except <allocation reference>s or the <reference>s of <refer options>, are implicitly qualified by the <locator qualifier>. Refer to paragraph 4.3.2 for a discussion of storage classes, and to paragraph 6 for a discussion of <reference>s.

5.4.5 Binary

Syntax:

<binary attribute>::= binary|bin

An item or <literal constant> declared with the <binary attribute> represents a binary arithmetic value or values.

5.4.6 Bit

Syntax:

<bit attribute>::= bit[(<length>)]

<length>::= <extent expression>|*

<extent expression>::= <expression>[<refer option>]

<refer option>::= refer(<reference>)

An item or <literal constant> declared with the <bit attribute> represents a bit-string value or values.

If the item also has the <varying attribute>, the <length> is the maximum number of bits that the item can represent; otherwise, it is the number of bits in each value that the item represents. Refer to paragraph 4.1 for a discussion of data types.

The <length> must satisfy the constraints given in paragraph 5.4.2 for <area size>.

5.4.7 Builtin

Syntax:

<builtin attribute>::= builtin

A name declared with the <builtin attribute> must be one of the names listed in Section 13. Such a name represents a function whose definition is an intrinsic part of the PL/I language.

5.4.8 Character

Syntax:

<character attribute>::= {character|char}[(<length>)]

<length>::= <extent expression>|*

<extent expression>::= <expression>[<refer option>]

<refer option>::= refer(<reference>)

An item or <literal constant> declared with the <character attribute> represents a character-string value or values.

If the item also has the <varying attribute>, the <length> is the maximum number of characters that the item can represent; otherwise, it is the number of characters in each value that the item represents.

The <length> must satisfy the constraints given in paragraph 5.4.2 for <area size>.

5.4.9 Complex

Syntax:

<complex attribute>::= complex|cplx

Unless it also has a <picture attribute>, an item or <literal constant> declared with the <complex attribute> represents a complex arithmetic value or values. If the item has a <picture attribute>, it represents character-string value or values as described in paragraphs 4.1 and 5.4.39.

5.4.10 Condition

Syntax:

<condition attribute>::= condition|cond

A name declared with the <condition attribute> is a <condition name>. Refer to Section 10 for a discussion of conditions.

5.4.11 Constant

Syntax:

<constant attribute>::= constant

A name declared with the <constant attribute> is a named constant. A <literal constant> is always declared with the <constant attribute>. Constants cannot be assigned values during program execution.

5.4.12 Controlled

Syntax:

<controlled attribute>::= controlled|ctl

A name declared with the <controlled attribute> is a variable whose storage class is controlled. Refer to paragraph 4.3.2 for a discussion of storage classes.

5.4.13 Decimal

Syntax:

<decimal attribute>::= decimal|dec

An item or <literal constant> declared with the <decimal attribute> represents a decimal arithmetic value or values.

5.4.14 Defined

Syntax:

<defined attribute>::= {defined|def}<base reference>

<base reference>::= (<reference>)|<reference>

A name declared with the <defined attribute> is a variable whose generation of storage is identified by the <base reference>. Refer to paragraph 4.3.3.3 for a discussion of storage sharing through the use of defined variables.

5.4.15 Dimension

Syntax:

<dimension attribute>::= [<dim key>][(<bound>[,<bound>]...)]

<dim key>::= dimension|dim

<bound>::= {[<extent expression>:]<extent expression>}|*

<extent expression>::= <expression>[<refer option>]

<refer option>::= refer(<reference>)

If the <dim key> is omitted, the <dimension attribute> must be the first <attribute> in the <attribute set> of a <descriptor>, <declare statement> or <default statement>, and the parenthesized list of <bound>s cannot be omitted.

An item declared with the <dimension attribute> represents array values. If only one <extent expression> is given, let L be 1 and let H be the <extent expression>; otherwise let L be the first <extent expression> and let H be the second <extent expression>. The number of elements in the dimension is H-L+1, where H must be greater than or equal to L.

If the item has the <static attribute>, each <bound> must be an optionally signed <decimal integer>.

If the item has the <parameter attribute> or is part of a <descriptor>, each <bound> must be an optionally signed <decimal integer> or an asterisk.

If the item does not have the <parameter attribute> or is not part of a <descriptor>, it cannot have an asterisk <bound>.

If the name does not have the <based attribute>, the <extent expression> cannot have a <refer option>.

Evaluation of the <expression> in an <extent expression> must yield a scalar value suitable for conversion to a fixed-point, binary, real, integer. If a <refer-option> is given, the value of the <expression> must also be suitable for conversion to the data type of the variable identified by the <reference> in the <refer option>.

If a completed <attribute set> contains a <dimension attribute>, it must contain exactly one <dimension attribute> with a parenthesized list of <bound>s.

5.4.16 Direct

Syntax:

```
<direct attribute>::= direct
```

A file constant declared with the <direct attribute> causes the file-state block that it identifies to be opened with the <direct attribute>. A file-state block with the <direct attribute> selects the records of its associated data set by means of character-string valued keys. Refer to Section 11 for a discussion of input/output.

5.4.17 Entry

Syntax:

```
<entry attribute>::= entry[([<parameter descriptor list>])]

<parameter descriptor list>::= <parameter descriptor>
          [,<parameter descriptor>]...

<parameter descriptor>::= <descriptor>

<descriptor>::= <level>[<attribute set>]|
          [<level>]<attribute set>

<attribute set>::= <attribute>...
```

An item declared with the <entry attribute> represents entry values.

If the <parameter descriptor list> is omitted, an entry value represented by the item is invoked only when it is identified by the <entry reference> of a <call statement> or when it is identified by the <entry reference> of a <function reference> with a null <argument list>.

An <entry attribute> of the form "entry()" is equivalent to an <entry attribute> of the form "entry", except that the former is a complete <attribute> and the latter is an incomplete <attribute>. The significance of this difference is shown in paragraph 5.2.1.2 and paragraph 5.3.2.

A <parameter descriptor list> does not restrict the values that may be represented by the item. A <parameter descriptor list> is significant only when an entry value represented by the item is invoked.

The <parameter descriptor list> must produce a declaration for each <parameter descriptor> that is equivalent to the actual declaration of each parameter in the entry invoked by each invocation of the entry values represented by this item. Such declarations are equivalent only if they contain exactly the same <attribute set>s, except that the <parameter descriptor> cannot have: the <parameter attribute> or <internal attribute>.

An <attribute set> of a <descriptor> must be consistent. An <attribute set> of a <descriptor> is consistent only if it can be transformed into a <descriptor set> as described in paragraph 5.5.

A <descriptor> of a structure has exactly the same syntax as a <defactored declaration> of a structure variable, except that it has no name. Its members are declared exactly like the members of a structure variable, except that they have no names.

Example:

```
declare F entry(1,2 fixed,2 pointer,1,
     2 bit(1),2 bit(4),(10,10) pointer);
```

The entry F has three parameters. The first is a structure containing an integer and a pointer. The second is a structure containing two bit-strings, and the third is a ten-by-ten array of pointers.

5.4.18 Environment

Syntax:

<environment attribute>::= {environment|env}[(interactive)|(stringvalue)]

A file constant declared with an <environment attribute> causes the file-state block that it identifies to be opened with the <environment attribute>.

If a completed <attribute set> contains an <environment attribute>, it must contain exactly one <environment attribute> with a parenthesized keyword which may be "interactive" or "stringvalue."

A file-state block with an <environment attribute> specifying "interactive" causes the execution of each <put statement> that references the file to finish its output by writing a linemark. This form of <environment attribute> is normally used when the data stream attached to the file-state block is an interactive device used for both input and output.

If a file-state block has an <environment attribute> specifying "stringvalue," the execution of a <read statement>, <rewrite statement>, or <write statement> is affected as follows. If a <read statement> has an <into option> referencing a scalar variable with the <character attribute> and the <varying attribute>, the complete record in the file is treated as a character-string value and is assigned to the variable by a normal string assignment. If a <rewrite statement> or <write statement> has a <from option> referencing a scalar variable with the <character attribute> and the <varying attribute>, the record placed in the file will be a character string that is equal to the current value of the variable. If a <read statement> has an <into option> referencing a scalar variable with the <bit attribute> and the <varying attribute>, the complete record in the file is treated as a bit-string value and is assigned to the variable by a normal string assignment. If a <rewrite statement> or <write statement> has a <form option> referencing a scalar variable with the <bit attribute> and the <varying attribute>, the record placed in the file will be a bit string that is equal to the current value of the variable. This form of the <environment> attribute is useful for processing a file containing strings of different lengths, especially when the file was not created using PL/I record output.

5.4.19 External

Syntax:

<external attribute>::= external|ext

A name declared with the <external attribute> has external scope and is known in all <block>s in which the same name is declared with the <external attribute> and in all contained <block>s, except those <block>s in which the name is redeclared with the <internal attribute>. All declarations of an external name must have equivalent <attribute set>s, and all such declarations refer to the same generation of storage, the same constant, or the same condition.

5.4.20 File

Syntax:

<file attribute>::= file

An item declared with the <file attribute> represents file values.

5.4.21 Fixed

Syntax:

<fixed attribute>::= fixed

An item or <literal constant> declared with the <fixed attribute> represents a fixed-point arithmetic value or values.

5.4.22 Float

Syntax:

<float attribute>::= float

An item or <literal constant> declared with the <float attribute> represents a floating-point arithmetic value or values.

5.4.23 Format

Syntax:

<format attribute>::= format

An item declared with the <format attribute> represents format values.

5.4.24 Generic

Syntax:

<generic attribute>::= generic[(<alternative list>)]

<alternative list>::= <alternative>[,<alternative>]...

<alternative>::= <entry reference>when([<selector>])

<entry reference>::= <reference>

<selector>::= <arg selector>[,<arg selector>]...

<arg selector>::= *|[<level>]<attribute set>|
<level>[<attribute set>]

<attribute set>::= <attribute>...

A name declared with a <generic attribute> is the name of a set of entry variables and entry constants. Refer to paragraph 6.9 for a discussion of <reference>s to generic names.

If a completed <attribute set> contains a <generic attribute>, it must contain exactly one <generic attribute> with an <alternative list>.

The <attribute>s used in an <arg selector> are restricted to the <attribute>s allowed in a <parameter descriptor> as described in paragraph 5.4.17, except that these two additional rules apply:

All <extent expression>s must be asterisks.

The <precision attribute> has an extended syntax that permits a range of precision values to be specified.

<precision attribute>::= [precision|prec]
(<low prec>[:<high prec>][,<low scale>
[:<high scale>]])

Both <low prec> and <high prec> must be <decimal integer>s and the value of <low prec> must be less than the value of <high prec>. Both <low scale> and <high scale> are optionally signed <decimal integer>s and the value of <low scale> must be less than the value of <high scale>. Note that this extended form of the <precision attribute> is only permitted in an <arg selector> of a <generic attribute>.

5.4.25 Initial

Syntax:

```
<initial attribute>::= {initial|init}[<initial list>]

<initial list>::= (<initial item>[,<initial item>]...)

<initial item>::= <factor><initial list>|
     [<factor>]<initial value>|(<expression>)

<initial value>::= [+|-|^]<literal constant>|
     [+|-]<real constant>{+|-}<imaginary constant>|
     [+|-|^]<reference>|*

<factor>::= (<expression>)
```

If a completed <attribute set> contains an <initial attribute>, it must contain exactly one <initial attribute> with an <initial list>.

If the declaration also has the <static attribute>, the <factor> must be a <decimal integer>, the <initial item> must not be (<expression>), and the <initial value> must be <literal constant>s or <reference>s to the null and empty built-in functions.

Evaluation of the <factor> must yield a scalar arithmetic or string value. The value of the <factor> is converted to a real, fixed-point, binary integer whose value must be greater than zero. Evaluation of each <expression> or <reference> in the <initial value> or <initial item> must yield scalar values.

An <initial attribute> provides an ordered sequence of scalar values that are assigned to the scalar components of each generation of the variable when the generation is allocated. Note that the elements of an array are stored in row-major order and that the scalar values of the <initial attribute> are assigned to the elements in row-major order. Refer to Section 4 for a discussion of array storage and allocation.

The program is in error if the number of elements in the array is not equal to the number of scalar values given in the <initial attribute>. It is also in error if a scalar variable is declared with an <initial attribute> that specifies more than one value.

An asterisk <initial value> causes the scalar variable to which it applies to not be initialized.

During compilation of an <external procedure>, the lexical level syntax rules are applied before any high level syntax rules are applied. Consequently, a <character string constant> or a <bit string constant> beginning with a parenthesized <decimal integer> is expanded into a single constant as described in paragraph 2.6.

Example:

```
declare x(5) bit(5) initial((5)"1"b);
```

is equivalent to:

```
declare x(5) bit(5) initial("11111"b);
```

To set all 5 elements of the array to all ones we must write:

```
declare x(5) bit(5) initial((5)(5)"1"b);
```

or:

```
declare x(5) bit(5) initial((5)(1)"11111"b);
```

5.4.26 Input

Syntax:

<input attribute>::= input

A file constant declared with the <input attribute> causes the file state block that it identifies to be opened with the <input attribute>.

It is an error to execute a <write statement>, <locate statement>, <rewrite statement>, <delete statement>, or <put statement> whose <file option> identifies a file-state block that has an <input attribute>. Refer to Section 11 for a discussion of input/output.

5.4.27 Internal

Syntax:

<internal attribute>::= internal|int

A name declared with the <internal attribute> has internal scope and is known only in the <block> in which it is declared and all contained <block>s, except those <block>s in which it is redeclared.

5.4.28 Irreducible

Syntax:

<irreducible attribute>::= irreducible|irred

An item declared with the <irreducible attribute> represents entry values. When an entry value represented by a name declared with an <irreducible attribute> is invoked, it is assumed to designate an irreducible entry as described in paragraph 6.11.

An <irreducible attribute> does not restrict the entry values represented by the item; its only significance is to force the entry values represented by the item to be invoked once for each evaluation of an <entry reference> in a <call statement> or <function reference>.

5.4.29 Keyed

Syntax:

<keyed attribute>::= keyed

A file constant declared with the <keyed attribute> causes the file-state block that it identifies to be opened with the <keyed attribute>. A file-state block with the <keyed attribute> may select the records of its associated data set by means of character-string valued keys. Refer to Section 11 for a discussion of input/output.

5.4.30 Label

Syntax:

<label attribute>::= label

An item declared with the <label attribute> represents label values.

5.4.31 Like

Syntax:

<like attribute>::= like<like reference>

<like reference>::= <identifier>[.<identifier>]...

The <like attribute> is a macro-attribute and is fully described in paragraph 5.2.2. The <like attribute> cannot appear in a <default statement>, in a <descriptor>, or in a <generic attribute>.

5.4.32 Local

Syntax:

<local attribute>::= local

An item declared with the <local attribute> represents either label values or format values.

When a name declared with the <local attribute> is referenced during evaluation of a <goto statement>, its value must be a label value derived from a <label prefix> immediately contained in the same <block> that immediately contains the declaration of the name.

When a name declared with the <local attribute> is referenced during evaluation of a <remote format>, its value must be a format value derived from a <label prefix> immediately contained in the same <block> that immediately contains the declaration of the name.

A <local attribute> does not restrict the values represented by the item, except when the item is referenced by a <goto statement> or <remote format>.

5.4.33 Member

Syntax:

<member attribute>::= member

An item declared with the <member attribute> must be a member of a structure as described in paragraph 5.2.3.1.3.

5.4.34 Nonvarying

Syntax:

```
<nonvarying attribute>::= nonvarying|nonvar
```

An item declared with the <nonvarying attribute> represents string values that all have the same length. The length is given by the <length> specified in the <bit attribute> or <character attribute>.

5.4.35 Offset

Syntax:

```
<offset attribute>::= offset[(<reference>)]
```

An item declared with the <offset attribute> represents offset values. Evaluation of the <reference> must yield a scalar area.

5.4.36 Options

Syntax:

```
<options attribute>::=
options{(constant)|(<option specification>[,<option specification>]...)}

<option specification>::= {variable|non_quick|support|main|separate_static
     |packed_decimal}
```

The <options attribute> is used to provide nonstandard information about variables and entry values. Unless the keyword "constant" is specified, an item declared with an <options attribute> represents entry values.

When an entry value represented by a name declared with the <options attribute> with the keyword "variable" is invoked, it is assumed to designate a nonstandard Multics entry that requires full run-time argument descriptions. An entry is nonstandard if it accepts a variable number of arguments or allows a given argument to have different <attribute>s each time the entry is invoked. Standard PL/I <procedure>s described by this document never need this <attribute>. The Multics Programmers' Manual identifies all Multics entries that must be declared with the <options attribute> specifying "variable."

An <options attribute> specifying "variable" does not restrict the entry values represented by the item; its only significance is to force all invocations of the entry values represented by the item to have complete run-time argument descriptions as required by nonstandard Multics entries.

If a <procedure statement> or <entry statement> contains an <options attribute> specifying "variable", that <statement> identifies a nonstandard Multics entry requiring complete run-time argument descriptions.

If the <procedure statement> heading an <external procedure> contains an <options attribute> specifying "support", that <external procedure> is considered to be a Multics runtime support <procedure>. This <option specification> should only be used by systems programmers; its use affects error messages printed by Multics. The keyword "support" may only be used with an <options attribute> that is contained in the <procedure statement> heading an <external procedure>.

The <options attribute> may specify the keyword "non_quick" if it is contained within a <begin statement> or within a <procedure statement> that does not head an <external procedure>. This <option specification> forces the compiler to obtain a new stack frame for the program when the <block> headed by the aforementioned <procedure statement> or <begin statement> is activated. If this <option specification> is not used, the compiler may attempt to have this <block>'s activation record share the stack frame of another <block>.

If the <procedure statement> heading an <external procedure> contains an <options attribute> specifying "main", and the program is running within a run unit, that <external procedure> is considered the main <procedure> of the run unit. Since this only affects the semantics of the <return statement> and <end statement>, this <option specification> should be used only to identify the first non-system <procedure> of the run unit when it is important to affect the semantics of the aforementioned <statement>s. The keyword "main" may be used only with an <options attribute> that is contained in the <procedure statement> heading an <external procedure>.

If the <procedure statement> heading an <external procedure> contains an <options attribute> specifying "separate_static", the compiler produces an object segment with a static section separate from the linkage section. Since this degrades the efficiency of the object code, this <option specification> should be used only with prelinked subsystems. The keyword "separate_static" may be used only with an <options attribute> that is contained in the <procedure statement> heading an <external procedure>.

If the <procedure statement> heading an <external procedure> contains an <options attribute> specifying "packed_decimal", the compiler prints no warning message for using unaligned decimal (packed decimal) variables. Otherwise a warning message is printed if unaligned decimal variables are used. Release 25 of the compiler allocates unaligned decimal variables by packing 2 digits per byte. Previous releases of the compiler allocated unaligned decimal at only 1 digit per byte. Specifying "packed_decimal" in the <options attribute> identifies <external procedure>s designed to be compiled with Release 25 or later release of the compiler. In a future release, the need to specify "packed_decimal" in an <options attribute> on a <procedure statement> to suppress this warning will be eliminated.

This page intentionally left blank.

The <options attribute> may specify the parenthesized keyword "constant" for items that are not entry values. Specification of "constant" causes the variable to be allocated in the text section of the object segment. The completed <attribute set> of a variable for which "options(constant)" has been specified must contain the <internal attribute>, the <static attribute>, and either the <initial attribute> or the <structure attribute>. If the latter is true, all nonstructure members of the structure must contain the <initial attribute>. Except for its allocation, the variable is treated semantically as any other internal static variable. It is an error to change the value of a variable for which "options(constant)" has been specified during execution of a program.

If the <options attribute> is contained within a <procedure statement> or <begin statement>, one or more <option specification>s must be specified.

If a completed <attribute set> contains an <options attribute>, it must contain exactly one <options attribute> with either the parenthesized keyword "constant", or with a parenthesized list of one or more <option specification>s.

5.4.37 Output

Syntax:

<output attribute>::= output

A file constant declared with the <output attribute> causes the file-state block that it identifies to be opened with the <output attribute>. It is an error to execute a <read statement>, <get statement>, <rewrite statement>, or <delete statement> whose <file option> identifies a file-state block that has an <output attribute>. Refer to Section 11 for a discussion of input/output.

5.4.38 Parameter

Syntax:

<parameter attribute>::= parameter|parm

A name declared with the <parameter attribute> must appear as a parameter in a <parameter list> of the <procedure> in which it is declared.

5.4.39 Picture

Syntax:

<picture attribute>::= {picture|pic}["<picture>"]

An item declared with the <picture attribute> represents character-string values whose conversion to arithmetic values or other character-string values is controlled by the <picture>. Refer to paragraph 8.2.12 for a discussion of picture controlled conversion and the syntax of <picture>s.

If a completed <attribute set> contains a <picture attribute>, it must contain exactly one <picture attribute> with a <picture>.

5.4.40 Pointer

Syntax:

<pointer attribute>::= pointer|ptr

An item declared with the <pointer attribute> represents pointer values.

5.4.41 Position

Syntax:

<position attribute>::= {position|pos}[(<position>)]

<position>::= <expression>

A name declared with the <position attribute> must be a defined variable suitable for string overlay defining as described in paragraph 4.3.3.6.

If a completed <attribute set> contains a <position attribute>, it must contain exactly one <position attribute> with a <position>.

5.4.42 Precision

Syntax:

<precision attribute>::= [<precision key>][(<precision>
[,<scale factor>])]

<precision key>::= precision|prec

<precision>::= <decimal integer>

<scale factor>::= [+|-]<decimal integer>

An item declared with the <precision attribute> represents arithmetic values.

If the <precision key> is omitted, the <precision attribute> must immediately follow either the <fixed attribute>, <float attribute>, <binary attribute>, <decimal attribute>, <real attribute>, or the <complex attribute>. If the <precision key> is omitted, the remaining part of the <attribute> must be given. If the <scale factor> is present, the item must have the <fixed attribute> and not the <float attribute>.

The <precision> specifies the number of digits that is sufficient to express all values represented by this item. The <scale factor> defines the position of the decimal or binary point. The point is located k digits to the left of the rightmost digit when the <scale factor> is positive, and -k digits to the right of the rightmost digit when the <scale factor> is negative, where k is the value of the <scale factor> and must be in the range $-128 \leq k \leq 127$. The <precision> is restricted to a nonzero value ≤ 59 if the item has the <decimal attribute>, to a nonzero value ≤ 71 if the item has the <binary attribute> and the <fixed attribute>, and to a nonzero value ≤ 63 if the the item has the <binary attribute> and the <float attribute>.

When the <precision attribute> is used in a <generic attribute>, it has an extended syntax as shown in paragraph 5.4.24.

If a completed <attribute set> contains a <precision attribute>, it must contain exactly one <precision attribute> with a <precision>.

5.4.43 Print

Syntax:

<print attribute>::= print

A file constant declared with the <print attribute> causes the file-state block that it identifies to be opened with the <print attribute>. A file-state block with the <print attribute> writes data into its data stream as if it were a printer. Refer to Section 11 for a discussion of input/output.

5.4.44 Real

Syntax:

<real attribute>::= real

Unless it also has a <picture attribute>, an item or <literal constant> declared with the <real attribute> represents a real arithmetic value or values. If the item has a <picture attribute>, it represents character-string values as described in paragraphs 4.1.6 and 5.4.41.

5.4.45 Record

Syntax:

<record attribute>::= record

A file constant declared with the <record attribute> causes the file-state block that it identifies to be opened with the <record attribute>. A file-state block with the <record attribute> can only be attached to a record data set and cannot be attached to a stream data set. Refer to Section 11 for a discussion of input/output.

5.4.46 Reducible

Syntax:

<reducible attribute>::= reducible|red

An item declared with the <reducible attribute> represents entry values. When an entry value represented by a name declared with a <reducible attribute> is invoked, it is assumed to designate a reducible function as described in paragraph 6.11.

A <reducible attribute> does not restrict the entry values represented by the item, its only significance is to possibly reduce the number of invocations of the entry values represented by the item.

5.4.47 Returns

Syntax:

<returns attribute>::= returns[([<returns descriptor>])]

<returns descriptor>::= <descriptor>[,<descriptor>]...

<descriptor>::= <level>[<attribute set>]|
[<level>]<attribute set>

<attribute set>::= <attribute>...

An item declared with a <returns attribute> represents entry values. An entry value represented by a name declared with a <returns attribute> cannot be invoked by the execution of a <call statement>.

A <returns attribute> does not restrict the entry values represented by the item, its only significance is to limit the invocation of such entry values to the evaluation of <function reference>s.

The <extent expression>s that appear in a <returns descriptor> must be <decimal integer>s or asterisks. If given as <decimal integer>s, the extents of all values returned by the function are the same. If given by asterisks, each invocation of the function can return a value whose extents may differ from the previous invocation.

The <varying attribute>, <nonvarying attribute>, <aligned attribute>, and <unaligned attribute> can appear in a <returns descriptor>. The use of these <attribute>s in a <returns descriptor> provides optimization information to the compiler but has no effect on the semantics of the language.

The <returns descriptor> must produce a declaration that is identical to the declaration produced by the <returns descriptor> of the <returns attribute> written on the <procedure statement> or <entry statement> invoked by each invocation of the entry values represented by the item.

An <attribute set> of a <descriptor> must be consistent. An <attribute set> is consistent only if it can be transformed into a <descriptor set> as described in paragraph 5.5.

If a completed <attribute set> contains a <returns attribute>, it must contain exactly one <returns attribute> with a <returns descriptor>.

5.4.48 Sequential

Syntax:

<sequential attribute>::= sequential|seql

A file constant declared with the <sequential attribute> causes the file-state block that it identifies to be opened with the <sequential attribute>. A file-state block with the <sequential attribute> selects the records of its associated data set in the order in which the records are recorded, unless the file-state block also has the <keyed attribute> in which case the input/output <statement> may supply a character-string valued key that is used to select a record from the data set. Refer to Section 11 for a discussion of input/output.

5.4.48a Signed

Syntax:

<signed attribute>::= signed

The <signed attribute> is a nonstandard <attribute> and its use makes programs dependent on Multics PL/I. The <signed attribute> influences the representation of values in storage. A signed arithmetic variable always contains storage to represent the sign of its value.

If a generation of storage is to be shared or accessed by more than one name, and one of those names is declared with the <signed attribute>, then none of the other names may be declared with the <unsigned attribute>.

5.4.49 Static

Syntax:

<static attribute>::= static

A name declared with the <static attribute> is a variable whose storage class is static. Refer to paragraph 4.3.2 for a discussion of storage classes.

5.4.50 Stream

Syntax:

<stream attribute>::= stream

A file constant declared with the <stream attribute> causes the file-state block that it identifies to be opened with the <stream attribute>. A file-state block opened with the <stream attribute> can be attached only to a stream data set and cannot be attached to a record data set. Refer to Section 11 for a discussion of input/output.

5.4.51 Structure

Syntax:

<structure attribute>::= structure

An item declared with the <structure attribute> must be a structure as described in paragraph 5.2.3.1.3.

This page intentionally left blank.

5.4.52 Unaligned

Syntax:

<unaligned attribute>::= unaligned|unal

The <unaligned attribute> is used in an implementation-defined manner to influence the representation of values in storage. In Multics PL/I, unaligned nonvarying bit-string values, unaligned binary arithmetic values, or unaligned pointer values are aligned on bit boundaries. Unaligned nonvarying character-string or decimal arithmetic values are aligned on 9-bit byte boundaries.

When a generation of storage is to be shared or accessed by more than one name, all names used to access the generation must have the same alignment <attribute>. Refer to paragraphs 4.3.1 and 4.3.3.

5.4.52a Unsigned

Syntax:

<unsigned attribute>::= unsigned|uns

The <unsigned attribute> is a nonstandard <attribute> and its use makes programs dependent on Multics PL/I. The <unsigned attribute> influences the representation of values in storage. A packed unsigned arithmetic variable does not contain storage to represent the sign of its value. An unsigned variable may store only nonnegative values.

If a generation of storage is to be shared or accessed by more than one name, and one of those names is declared with the <unsigned attribute>, then all of the other names must be declared with the <unsigned attribute>.

5.4.53 Update

Syntax:

<update attribute>::= update

A file constant declared with the <update attribute> causes the file-state block that it identifies to be opened with the <update attribute>. It is an error to execute a <get statement> or <put statement> whose <file option> identifies a file-state block that has an <update attribute>. Refer to Section 11 for a discussion of input/output.

5.4.54 Variable

Syntax:

<variable attribute>::= variable

A name declared with the <variable attribute> is a variable.

5.4.55 Varying

Syntax:

<varying attribute>::= varying|var

An item declared with the <varying attribute> represents string values whose lengths may be any value, k, such that $0 \leq k \leq n$, where n is the <length> specified in the <bit attribute> or <character attribute>.

5.5 Attribute Consistency

To check the consistency of a completed <attribute set> consider the <attribute set> to be an ordered set of keywords formed by removing all parenthesized and auxiliary parts of each <attribute>. Replace any abbreviated keyword by its full keyword and replace all multiple occurrences of a given keyword by a single occurrence of the keyword. Let the order of the keywords be the order implied by these syntax rules. For this discussion, consider the options keyword to apply to the <options attribute> with one or more <entry value options> specified.

If the resulting ordered set of keywords is not described by these syntax rules, it is an inconsistent set and the program is in error. If it is described by these syntax rules and the constraints are satisfied, it is a consistent set.

Because file description attributes may be supplied during file opening as described in paragraph 11.3, an <attribute set> containing one or more file description attributes is considered a consistent set even though its file description attributes are only a subset of those described by <consistent file description>. If the file description attributes of an <attribute set> are not a subset of those described by <consistent file description>, the <attribute set> is an inconsistent set.

The validity of <expression>s, <reference>s, or <decimal integer>s that are part of an <attribute> are not considered when determining the consistency of an <attribute set>. Constraints of that type are described in paragraph 5.4 where the semantics of each <attribute> are given.

Each <literal constant set> must be produced by the declaration of a <literal constant> and each <descriptor set> must be produced by the declaration derived from a <descriptor>. A <named constant set>, other than an external entry constant or file constant, must be produced by a declaration derived from a <label prefix>.

Syntax:

```
<consistent attribute set>::= <condition set>|<builtin set>|
     <generic set>|<literal constant set>|
     <named constant set>|<descriptor set>|
     <variable set>

<condition set>::= external condition

<builtin set>::= internal builtin

<generic set>::= internal generic

<literal constant set>::= {<arithmetic>|bit|character}
     constant

<named constant set>::= internal label constant[dimension]|
     internal format constant|<scope><entry>constant|
     <scope>file<consistent file description>constant

<descriptor set>::= <data type><alignment>[dimension][member] [<sign type>]

<variable set>::= variable<data type><alignment>[dimension]
     <scope class>[initial] [<sign type>]

<data type>::= <arithmetic>|<string>|<entry>|structure[like]|
     pointer|offset|area|label[local]|format[local]|file

<arithmetic>::= {fixed|float}{binary|decimal}{real|complex}
     precision

<string>::= picture[real|complex]|
     {bit|character}{varying|nonvarying}

<entry>::= entry[options]
     {reducible returns|irreducible[returns]}

<alignment>::= aligned|unaligned

<scope>::= internal|external

<scope class>::= automatic internal|based internal|
     static<scope>|controlled<scope>|parameter internal|
     defined internal[position]|member internal

<consistent file description>::= <stream description>|
     <record description>

<stream description>::= stream{input|output[print]
     [environment]}

<record description>::= record{input|output|update}
     {<sequential description>|<direct description>}[environment]

<sequential description>::= sequential[keyed]

<direct description>::= direct keyed

<sign type>::= signed|unsigned
```

Constraints:

An <attribute set> of a member of a parameter structure or defined structure cannot have an <initial attribute>.

An <attribute set> containing the <parameter attribute> or the <defined attribute> may not contain an <initial attribute>.

An <initial attribute> cannot be given in the <attribute set> of a structure.

An <attribute set> containing the <signed attribute> must contain either the <fixed attribute> or the <float attribute>.

An <attribute set> containing the <unsigned attribute> must contain the <fixed attribute>, the <binary attribute>, and the <real attribute>.

Note: The <options attribute> is consistent with entry unless "constant" is specified, in which case it is consistent with static internal.

SECTION 6

REFERENCES

The value and storage of a variable, the value of a named constant, the value returned by a function, or the condition identified by a condition name is represented in the text of an <external procedure> by a <reference> to a declared name. A declaration establishes the meaning of a name within a given region of the program known as the scope of the declaration. Refer to Section 5 for a discussion of declarations and scope.

A <reference> is resolved by finding the declaration to which it refers. A <reference> is evaluated by locating the generation of storage or value represented by the declared name. A <reference> is resolved by the compiler and is evaluated during program execution.

Syntax:

```
<reference>::= <simple reference>|
     <subscripted reference>|
     <structure qualified reference>|
     <locator qualified reference>|
     <function reference>
```

The meaning of a <reference> depends on the syntax of the <reference>, the <attribute>s of the declared name and the context in which the <reference> occurs.

Evaluation of a <reference> that identifies a variable either: yields a generation of storage of the variable, yields the current value stored in a generation of storage of the variable, or yields an identification of the variable's declaration. A <reference> yields a value unless it occurs in one of the following contexts:

In the following contexts, a <reference> yields a generation of storage:

1. A <target> of an <assignment statement>.
2. An <index> of a <multiple do>.
3. An argument, passed by-reference, by a <call statement> or <function reference>.
4. An argument to the "addr" built-in function, unless it identifies an unallocated controlled variable.
5. A <reference> in a <string option> of a <put statement>.
6. A <free reference> of a <free statement>.
7. A <reference> in a <pseudo-variable>, other than a <pageno pseudo>.
8. A <reference> in a <set option>, <in option>, <into option>, <from option> or <keyto option>.
9. A <target> of a <get item> in a <get statement> or a <get data ref> in a <get statement>.

10. A <reference> in a <refer option> evaluated as the target of the assignment of its evaluated extent.

In the following contexts, a <reference> is only an identification of its declaration and yields neither a value nor a generation of storage:

1. An <allocation reference> in a <allocate statement> or <locate statement>.

2. An argument of the "size" or "allocation" built-in function, or the first argument of the "convert" built-in function.

3. A <reference> to an unallocated controlled variable used as an argument to the "addr" built-in function.

Except in these contexts, it is an error to reference a variable whose generation of storage has not been allocated. It is always an error to reference the value of a variable if no value has been assigned to the variable. Refer to paragraph 4.3.2 for a discussion of storage allocation.

The order of evaluation of the components of a <reference> is undefined and any program that depends on the order is in error.

A <reference>, R, contained in a declaration of X is resolved in the <block> that immediately contains the declaration of X, and is evaluated when X is referenced or allocated. R is evaluated as if it were referenced in the <block> that immediately contains the <reference> to X or caused the allocation of X.

A <reference>, R, to a defined variable, X, whose <base reference> contains isubs or asterisks is mapped as described in paragraph 4.3.3.4 or 4.3.3.5 during evaluation of R. Any <reference>s in the <subscript>s of R are resolved in the <block> that immediately contains R, but all <reference>s in the <base reference> are resolved in the <block> that immediately contains X. R is evaluated in its immediately containing <block>.

6.1 Simple References

Syntax:

```
<simple reference>::= <identifier>
```

6.2 Subscripted References

Syntax:

```
<subscripted reference>::= <identifier>(<subscript>
     [,<subscript>]...)

<subscript>::= <expression>|*
```

Each <expression> must be a scalar value suitable for conversion to a fixed-point, binary, real, integer as described in Section 8. The number of <subscript>s must be equal to the number of dimensions declared for the name. Refer to paragraph 4.2 for a discussion of array data, and to paragraph 5.4.15 for a description of the <dimension attribute>.

6.3 Cross-Section References

A <subscripted reference> containing one or more asterisk <subscript>s is a cross-section <reference>. It is a <reference> to the array formed by the planes, indicated by asterisks, of the array identified by the reference. The number of dimensions of the array is the number of asterisks in the reference.

A cross-section <reference> containing only asterisk <subscript>s is equivalent to a <reference> to the entire array.

Cross-section <reference>s, other than those equivalent to a <reference> to the entire array, are usually <reference>s to unconnected arrays. As such they cannot be used as the argument to the "addr" built-in function and cannot be passed as arguments to array parameters, unless the parameters are declared with asterisk bounds. Otherwise, a cross-section <reference> can be used any place in a program that an array <reference> is permitted. Refer to paragraph 4.3.1.3 for a discussion of unconnected arrays.

Multics PL/I requires that parameters that receive unconnected arguments be declared with asterisk bounds. Henceforth in this document, the word "array" should be taken to include array values of cross-section <reference>s.

Example:

```
declare A(7,5), B(5,10), C(5);

A(I,*) = B(*,3)+C(*);
```

This <statement> computes a one-dimensional array of five values by adding the values of the 3rd column of B to the values of C and then assigns them to the Ith row of A. Note that a <reference> to C(*) is equivalent to a <reference> to C.

6.4 Structure Qualified References

The name of a member of a structure can have a scope which overlaps another declaration of the same name. To resolve ambiguity of <reference>s to such names, the <reference> must be qualified by <containing reference>s to one or more of its containing structures. Refer to paragraph 4.2 for a discussion of structure data and to paragraph 5.2.3.1.3 for a description of structure declarations.

Syntax:

```
<structure qualified reference>::=
     {<containing reference>.}...<member reference>

<containing reference>::= <simple reference>|
                          <subscripted reference>

<member reference>::= <simple reference>|
     <subscripted reference>
```

Examples:

```
A.B.C
X(I).Y.Z(5,K)
NODE.ELEMENT
```

The rightmost <reference> identifies the variable being referenced. It is contained within the structure identified by the immediately preceding <containing reference>, which in turn is contained within the structure identified by the immediately preceding <containing reference>, etc.

The rightmost <reference> is said to be fully qualified if it is qualified by a <containing reference> to each of its containing structures. A name declared at level n in a substructure has n-1 <containing reference>s if it is fully qualified. A <simple reference> or simple <subscripted reference> to a name declared by a level-one declaration is considered a fully qualified <reference> to that name. A fully qualified <reference> is never ambiguous.

The rightmost <reference> is said to be partially qualified if it has fewer <containing reference>s than it has containing structures.

The <subscript>s used in a <structure qualified reference> do not have to appear immediately following the names to which they are to apply. As long as the order of the <subscript>s is preserved, the <subscript>s may be moved to the left or the right and written after any of the names in the <reference>. Use of this feature obscures the meaning of a program and should be avoided.

The number of <subscript>s must equal the number of dimensions of the referenced name including all inherited dimensions.

Example:

```
declare 1 S(3),
        2 A(4),
        2 B;
```

A <subscripted reference> to A must contain two <subscripts>, and <subscripted reference>s to S or B must have one <subscript>.

Asterisk <subscript>s may be used to refer to cross-sections of arrays of structures or array structure members.

Example:

```
declare 1 S(3),
        2 A(4),
        2 B;

declare 1 X,
        2 Y(5);
```

The following are all valid cross-section <reference>s:

```
S(K).A(*)   S(*).A(*)   S(*).B   S(*)   X.Y(*)
```

6.5 Reference Resolution and Ambiguity

A declaration is applicable to a <structure qualified reference> if it is a declaration of a structure member some or all of whose containing structures have the same names and the same order as the <containing reference>s of the <structure qualified reference>. A declaration is applicable to a <simple reference> or <subscripted reference> if it is a declaration of the name given in the <simple reference> or <subscripted reference>.

References are resolved by looking for an applicable declaration in the <block> that immediately contains the <reference>. If no applicable declaration exists in that <block>, the next containing <block> is searched until a <block> containing an applicable declaration is found. The <reference> is in error if no applicable declaration exists in a containing <block>.

If only one applicable declaration exists in the <block> containing the first applicable declaration, the <reference> identifies that declaration. However, if the <block> contains more than one applicable declaration, the <reference> must be a fully qualified <reference> to one of the declarations in that <block>, and is resolved to identify that declaration. If it is not a fully qualified <reference> to one of the declarations in that <block>, the <reference> is ambiguous and in error.

The presence or absence of <subscript> lists or <argument list>s does not affect the resolution of <reference>s and cannot resolve otherwise ambiguous <reference>s.

6.6 Locator Qualified References

Syntax:

```
<locator qualified reference>::= <locator qualifier>-><based reference>

<locator qualifier>::= <reference>

<based reference>::= <simple reference>|
     <subscripted reference>|
     <structure qualified reference>
```

The <locator qualifier> must be a <reference> to a scalar locator variable or scalar locator-valued function. Its value identifies a generation of storage whose data and aggregate type are described by the declaration identified by the <based reference>. Refer to paragraph 4.3.2.5 for a full discussion of based variables.

It is an error to use a <locator-qualifier> to qualify a <reference> to a nonbased variable. Implicit qualification derived from the <based attribute> or explicit qualification by an arrow operator is required for all <reference>s to based variables except the <allocation reference> of an <allocate statement> or <locate statement>, and the <reference> of a <refer option>.

Examples:

```
P->X
S.ITEM(I,J)->TABLE.ENTRY
F(X+Y)->GAMMA(K)
HEAD->LIST.NEXT->LIST.VALUE
```

The last example shows a based <locator qualified reference>, HEAD->LIST.NEXT, used as the <locator qualifier> of LIST.VALUE.

6.7 Function References

The evaluation of a <function reference> results in the invocation of an entry value. The value of the <function reference> is the value returned by the invoked entry.

The syntax of a <function reference> differs from that of a <subscripted reference> only when the <function reference> has multiple or empty <argument list>s. In order to recognize a <function reference> with a single, nonempty, <argument list>, the compiler examines the <subscripted reference>. If the declared name identified by the <subscripted reference> has no dimensions, the <subscripted reference> is a <function reference>; otherwise, it is a <subscripted reference>.

A <structure qualified reference> containing two or more <subscripted reference>s is examined to determine if it is a <function reference>. If the declared name identified by the <structure qualified reference> has n dimensions then the leftmost n <subscript>s are considered part of the <structure qualified reference>, and any other parenthesized lists must follow the rightmost name. If a parenthesized list follows the <structure qualified reference> it is an <argument list> of the <function reference>. Arguments and <subscript>s cannot appear in the same parenthesized list.

Syntax:

```
<function reference>::= <entry reference><argument list>

<entry reference>::= <reference>

<argument list>::= ([<expression>[,<expression>]...])
```

Evaluation of the <entry reference> must yield a scalar entry value. A <function reference> is distinguished from a <reference> to an entry value by the presence of an <argument list>. The <argument list> is empty only if the entry has no parameters.

Examples:

```
declare F entry() returns(ptr);
declare G entry(fixed) returns(bit(1));
```

A <reference> to F is a <reference> to the entry value of F and is not a <function reference>. A <function reference> to the value returned by the invocation of F is written as F(). Similarly a <function reference> to G is written as G(K) while a <reference> to the entry value of G is written as G.

Since the entry value may itself be a <function reference>, it is possible to have multiple <argument list>s.

Example:

```
declare F entry(ptr) returns(entry(fixed) returns(float));
```

A <reference> to F(P)(I) is a <function reference> which returns a floating-point value. A <reference> to F(P) is a <function reference> which returns an entry value. A <reference> to F is a <reference> to the entry value of F and does not invoke F.

The entry value may be any <reference> including <locator qualified reference>s and <structure qualified reference>s with or without <subscript>s. The only restriction on the <reference> is that it must yield a scalar entry value.

Example:

```
declare F(5,6) entry(fixed,fixed) returns(ptr);
```

A <reference> to F(I,J) is a <reference> to the entry value of the (I,J) element of the array F. A <reference> to F(I,J)(K,L) is a <reference> to the pointer value returned by the invocation of the (I,J) element of the array F.

Example:

```
declare 1 S,2 B(N),3 E entry() returns(char(*));
```

A <reference> to S.B(I).E or any equivalent <reference> such as S.B.E(I), S(I).B.E, or S.E(I), are all <reference>s to the entry value of E and are not <reference>s to the value returned by E. S.B(I).E(), S.B.E(I)(), or S(I).B.E() are <function reference>s that invoke E.

The declaration of an entry variable or constant must contain an <entry attribute> giving <descriptor>s of all parameters, and a <returns attribute> describing the return value if the entry is to be invoked as a function. Refer to Section 5 for a discussion of declarations.

6.8 Built-in Function References

A built-in function is an intrinsic part of the PL/I language. All <reference>s to built-in function names are <function reference>s in that they refer to the value returned by the function. A built-in function name has no entry value and cannot be used in contexts that require entry values. Built-in functions that take no arguments may be referenced with or without an empty <argument list>. Refer to Section 13 for a complete discussion of all built-in functions.

6.9 Generic References

If the name identified by an <entry reference> in a <function reference> or in a <call statement> is declared with a <generic attribute>, the <entry reference> is a generic reference. The compiler transforms a generic reference into an <entry reference> to one of the entries specified by the <generic attribute>. The proper entry is selected by matching the <alternative>s specified by the <generic attribute> against the arguments of the generic reference. For descriptive convenience the syntax of the <generic attribute> is repeated here. Refer to Section 5 for a discussion of declarations.

Syntax:

```
<generic attribute>::= generic[(<alternative list>)]

<alternative list>::= <alternative>[,<alternative>]...

<alternative>::= <entry reference>when([<selector>])

<entry reference>::= <reference>

<selector>::= <arg selector>[,<arg selector>]...

<arg selector>::= *|[<level>]<attribute set>|
        <level>[<attribute set>]

<attribute set>::= <attribute>...
```

The <arg selector>s of the leftmost <alternative> are matched against the arguments of the generic reference. An asterisk <arg selector> matches any argument; otherwise, an <arg selector> only matches an argument if the argument has every <attribute> found in the <arg selector>. The argument's precision is considered to match the <arg descriptor> only if it lies within the range specified by the <precision attribute> given in the <arg selector>. An asterisk <extent expression> is considered to match any <extent expression> of the argument.

A structure parameter of a generic entry must be represented by a set of two or more <arg selector>s containing <level>s. In this case, the set of <arg selector>s must satisfy the constraints on <level>s given for <descriptor>s of an <entry attribute> in paragraph 5.4.17. If an <arg selector> does not represent a structure parameter, it cannot have a <level>.

A structure argument matches a set of <arg selector>s with <level>s only if the structure and each member of the structure match the corresponding <arg selector> in the set. If all <arg selector>s of an <alternative> match the arguments of the generic reference, the generic reference is transformed into an <entry reference> to the <entry reference> given by that <alternative>; otherwise, the next <alternative> is tried. The program is in error if no <alternative> matches the generic reference. A generic reference with no arguments is transformed into the leftmost <alternative> that has no <arg selector>s.

Example:

```
declare F generic(F1 when(binary,pointer),
     F2 when(decimal,pointer));

declare F1 entry(fixed binary,pointer) returns(fixed);

declare F2 entry(fixed decimal,pointer) returns(fixed);

declare X fixed binary, Y pointer;

A = F(X,Y);
```

In this example, the generic reference F(X,Y) is transformed into the <function reference> F1(X,Y).

6.10 Parameters and Arguments

An argument is an <expression> used in the <argument list> of a <call statement> or <function reference>. A parameter is a name declared in the invoked procedure and used by the invoked <procedure> to reference an argument. The ith argument in an <argument list> corresponds to the ith parameter specified in the <parameter list> of the invoked entry. The correspondence between an argument and a parameter lasts until the block activation that established the correspondence is deactivated by a <return statement> or nonlocal goto.

6.10.1 Argument Passing By-value or By-reference

When an argument is passed by-value, it is evaluated and assigned to a unique generation of storage in the calling <procedure> and that generation is associated with the parameter. Since the generation of storage associated with the parameter is not the generation occupied by the original argument, assignments of values to the parameter by the invoked procedure do not affect the value of the argument in any way. Similarly, changes made to the value of the original argument, while the block activation created by the invocation of the entry is still active, do not affect the value of the parameter.

When an argument is passed by-reference, its generation of storage is associated with the parameter. The parameter thus shares the same generation of storage as the original argument and either can be used to assign values to the generation.

An argument is passed by-reference only when it is a <reference> to a variable whose <attribute>s and extents match the <attribute>s and extents declared for the parameter. The following <attribute>s must match:

> <fixed attribute>, <float attribute>, <binary attribute>, <decimal attribute>, <real attribute>, <complex attribute>, <precision attribute>, <bit attribute>, <character attribute>, <picture attribute>, <pointer attribute>, <offset attribute>, <area attribute>, <label attribute>, <format attribute>, <entry attribute>, <file attribute>, <varying attribute>, <nonvarying attribute>, <aligned attribute>, <unaligned attribute>, <signed attribute>, <unsigned attribute>.
>
> The <parameter descriptor list> of an <entry attribute> and the <reference> of an <offset attribute> are not involved in the matching process.
>
> Attributes not included in the above list are not considered in the matching operation, but if the argument is an array, then the <dimension attribute> of the parameter must give the same dimensionality as the array argument.

It is an error to pass an unconnected array by-reference to a parameter declared with constant extents.

If an argument is a <reference> to a variable that does not match the parameter, it cannot be passed by-reference and is passed by-value. A <literal constant>, a <reference> to a named constant, a <reference> enclosed in parentheses, and a <reference> to an isub-defined array (not scalar elements of an isub-defined array) are considered <expression>s and are passed by-value.

6.10.2 Argument Conversion and Promotion

The evaluation of an argument passed by-value includes the conversion and promotion necessary to force it to conform to the data type and aggregate type of the parameter. If the argument cannot be converted or promoted to conform to the parameter the program is in error. Refer to Sections 8 and 9 for discussions of conversion and promotion.

6.10.3 Asterisk and Constant Extents of Parameters

A parameter may be declared with either constant or asterisk extents. (An extent is an array <bound>, string <length>, or <area size>). If a parameter is declared with asterisk extents the asterisks are replaced by the extents of the argument that corresponds to the parameter. This replacement occurs each time the parameter is associated with an argument and holds only so long as the parameter remains associated with the argument.

For the purpose of determining whether an argument is to be passed by-value or by-reference, an asterisk extent is considered to "match" any extent of the argument.

An array parameter that corresponds to an unconnected array argument must be declared with asterisk bounds. (Most cross-section <reference>s refer to unconnected array values.)

If a parameter is declared with constant extents, only arguments that have identical constant extents are considered to match the parameter.

6.10.4 Storage of a Parameter

Since the generation of storage associated with a parameter is always supplied by its corresponding argument, parameters have no <initial attribute> and are never allocated a generation of storage. The scope of a parameter is always internal to the <block> in which the name appears as a parameter.

It is an error to reference a parameter that is not associated with an argument.

6.11 Reducibility of Functions

If each invocation of an entry produces no side-effects, returns a value that depends only on the values of the arguments passed to that invocation, and invokes only reducible functions, the entry is a reducible function. A side-effect is any change in the value of any variable, file-state block, or data set known outside of the invoked entry or any of its dynamic descendents. Any entry that is not reducible is irreducible.

During evaluation of an <expression> the order of evaluation of <reference>s to irreducible functions is not defined, but each such <reference> is evaluated. A <reference> to a reducible function might not be evaluated if the compiler detects that such an evaluation would yield the same value as some previous evaluation. A <reference> to a reducible function might be evaluated before the <statement> in which it is written is executed, but a <reference> to an irreducible entry is always evaluated during the execution of the <statement> in which it is written. Refer to paragraph 5.4 for a discussion of the <reducible attribute> and <irreducible attribute>.

SECTION 7

EXPRESSIONS

There are three kinds of <expression>s: primitive expressions, prefix expressions and infix expressions. A primitive expression is a <reference> or a <literal constant>, a prefix expression is a prefix operator preceding one operand, while an infix expression is an infix operator between two operands.

An operand is one of the three kinds of <expression>s.

The value yielded by an <expression> has a data type and an aggregate type. Since <expression>s always yield values of the same data type and aggregate type, except for changing array-extents, they are characterized by their data type and aggregate type, and are referred to as: scalar <expression>s, array <expression>s, scalar pointer <expression>s, etc.

The data type of an <expression> is one of the data types described in paragraph 4.1, and the aggregate type is one of the aggregate types described in paragraph 4.2.

7.1 Evaluation of Expressions

7.1.1 Evaluation of Primitive Expressions

Since primitive expressions contain no operators, their evaluation consists solely of evaluating a <reference> or a <literal constant>.

The aggregate type and data type of the value yielded by evaluation of a primitive expression are determined by the declaration of the name identified by the <reference> or the declaration of the <literal constant>. If the primitive expression is a <function reference>, the aggregate type and data type are the aggregate type and data type of the value returned by the function.

The value yielded by evaluation of a primitive expression is the value of the variable or constant identified by the <reference> or <literal constant>. If the primitive expression is a <function reference>, the value of the evaluated primitive expression is the value returned by the function.

7.1.2 Evaluation of Prefix Expressions

The evaluation of a prefix expression consists of:

1. The evaluation of the operand.
2. The conversion of the value of the evaluated operand to the data type required by the operator. If the operand is an aggregate, each scalar component is converted to the required data type.

3. The application of the operator to the converted value of the operand. If the operand is an aggregate, the operator is applied to each scalar component of the aggregate.

The result of the evaluation of a prefix expression is a value whose aggregate type is the aggregate type of the evaluated operand. The data type of each scalar component of the result is the data type of the corresponding scalar component of the converted operand.

7.1.3 Evaluation of Infix Expressions

The evaluation of an infix expression consists of:

1. The evaluation of both operands.

2. The promotion of the two operands to the highest common aggregate type as described in Section 9.

3. The conversion of the operands to data types acceptable to the operator. If the operands are aggregates, corresponding scalar components are converted to the required data types.

4. The application of the operator to the converted values of the promoted operands. If the operands are aggregates, the operator is applied to each pair of corresponding scalar components of the operands.

The result of the evaluation of an infix expression is a value whose aggregate type is the common aggregate type of the promoted operands. The data type of each scalar component of the result is determined by the data types of the corresponding scalar components of the converted and promoted operands.

7.1.4 Order of Evaluation

The order of evaluation of an <expression> is determined by the precedence of the PL/I operators and by the parenthesization of subexpressions.

Within a pair of parentheses, operators are evaluated according to their precedence. Operators of higher precedence are evaluated before operators of lower precedence.

Operators of equal precedence are evaluated from left-to-right, except for the exponentiation and prefix operators which are evaluated from right to left.

The precedence of PL/I operators is:

```
Highest
   ¦   **  ^  prefix +  prefix -
   ¦   *  /
   ¦   infix +  infix -
   ¦   ||
   ¦   =  ^=  <  ^<  >  ^>  <=  >=
   ¦   &
   V   ¦
Lowest
```

The order of <expression> evaluation is determined by the precedence of operators and by parenthesization. However, within these constraints, the order of evaluation is not defined. A program that depends on any of the following is in error and the results of its execution are not defined:

1. The order in which operands are promoted to higher aggregate types.

2. The order in which operands are converted to different data types.

3. The order in which the scalar components of aggregate operands are converted and operated upon by infix or prefix operators.

4. The order in which <basic expression>s are evaluated.

5. The frequency with which a reducible <function reference> is evaluated. An irreducible <function reference> is evaluated once for each occurrence of the <function reference> in the text of an <external procedure> except as explained in 7.1.5.

7.1.5 Optional Evaluation

If the result of an operator can be determined without evaluation of one or more of its operands and no operand contains irreducible <function reference>s, the operands are not necessarily evaluated.

A program that depends on the full evaluation of all operands is in error and the result of its execution are undefined. Similarly, a program that depends on an operand not being evaluated is in error.

The following are invalid:

```
if p=null | p->x=5 then
if x=0 | y/x then
```

If the value of an <expression> or <reference> does not depend on the value of a contained <function reference>, then that <function reference> is not necessarily evaluated.

7.1.6 Expression Evaluation and Conditions

The <on statement> and <condition prefix> allow a program to detect and respond to various states of the executing program known as conditions. Refer to Section 10 for a full discussion of conditions, signals and <on unit>s.

In general, the order in which conditions are detected and the frequency with which they occur are not defined. This is due to the fact that the order of <expression> evaluation is not strictly defined.

Conditions that are signalled during <expression> evaluation cause the evaluation to be suspended and control to enter an <on unit>. In most cases, the program is in error if the <on unit> returns control to the point where the condition was detected. Refer to the description of each condition in paragraph 10.4 to see if a particular <on unit> can return control.

An <on unit> entered as a result of a condition signalled during <expression> evaluation cannot access variables that are assigned values by the interrupted <statement>. Similarly, the <on unit> cannot assign a value to any generation of storage accessible at the point where the signal occurred, nor can it allocate or free such a generation.

Example:

```
on zerodivide begin;
                .
                .
                .
              end;
C,B = 0;
C = A/B + A/B
```

In this example:

1. the value of C is not defined upon entry to the <on unit>.

2. if the <on unit> does a normal return the result of the program is undefined, regardless of whether or not the <on unit> assigned a new value to B.

3. the number of times the zerodivide condition is signaled is not defined, but at least one will be signalled.

7.2 Formal Syntax of Expressions

The syntax given in this section defines the precedence of operators and an order of evaluation of <expression>s. The actual order of evaluation may differ from the order expressed by these syntax rules only in the following cases:

1. If the result of an operator can be determined without the evaluation of one or more of its operands, and no operand contains an irreducible <function reference>, the operands need not be evaluated.

2. The order of evaluation of the <basic expression>s contained within an <expression> is not defined.

Syntax:

```
<expression>::= <expression seven>|
     <expression>|<expression seven>

<expression seven>::= <expression six>|
     <expression seven>&<expression six>

<expression six>::= <expression five>|
     <expression six>{=|^=|<|^<|<=|>|^>|>=}<expression five>

<expression five>::= <expression four>|
     <expression five>||<expression four>

<expression four>::= <expression three>|
     <expression four>{+|-}<expression three>

<expression three>::= <expression two>|
     <expression three>{*|/}<expression two>

<expression two>::= <basic expression>|<simple expression>|
     <parenthesized expression>|<expression one>

<expression one>::= {<basic expression>|
     <parenthesized expression>}**<expression two>

<simple expression>::= {+|-|^}<expression two>

<parenthesized expression>::= (<expression>)

<basic expression>::= <reference>|<literal constant>|<isub>
```

Note that an <isub> is valid only in <expression>s that are part of the <base reference> of a <defined attribute>.

7.3 Operators

7.3.1 Arithmetic Operators

The prefix arithmetic operators are:

+ plus
- minus

The infix arithmetic operators are:

+ add
- subtract
* multiply
/ divide
** exponentiate

7.3.1.1 Operand Conversion for Arithmetic Operators

Arithmetic operators require arithmetic operands and force conversion of their operands. The conversions are performed according to the rules given in Section 8, and the target for the conversions is given by the following rules:

1. A character-string operand, X, is converted to an arithmetic value, X', where the data type of X' is the data type that a fixed-point, decimal, real value of precision (59,0) would have been converted to, had it appeared in place of X.

2. A bit-string operand, X, is converted to an arithmetic value, X', where the data type of X' is the data type that a fixed-point, binary, real value of precision (71,0) would have been converted to, had it appeared in place of X.

3. If the operands of an arithmetic infix operator differ in mode, base, or type, the operands are converted to the target mode, base, and type given by the following rules.

   The target attributes are: complex, if the modes differ; binary, if the bases differ; and floating-point, if the types differ; otherwise, the target has the common mode, base, or type. The precision of the two operands may differ without causing any conversion. If a conversion occurs due to a difference in mode, base or type, the precision of the converted operand is given by the rules in paragraph 8.2.10.

   The exponentiation operator is an exception to these rules. See Section 7.3.1.2.3.

   Example:

   ```
   declare A character(5), B float binary(27);

   C = A+B;
   ```

   In this example, A is converted as if it were a real, fixed-point, decimal, integer of precision (59,0). The target mode is real, the target base is binary, and the target type is floating-point. Because B already has the target attributes, it is not converted, but A is converted to a real, floating-point, binary value of precision (63).

7.3.1.2 Results of Arithmetic Operators

After the operands have been converted, the operation is performed. The result is an arithmetic value whose type, base, mode and precision are determined by the converted operands and the operator as described in the following sections.

A decimal floating-point result of precision (p) contains only the most significant p digits of the true arithmetic result rounded at the (p+1)th digit.

A binary floating-point result of precision (p) contains the most significant n digits of the true arithmetic result, where n is 27 if $p \leq 27$ and n is 63 if $p > 27$. When the final result of the evaluation of an <expression> is assigned to a variable or to a generation of storage to be passed as an argument, n significant digits are stored if the target is unpacked, but p significant digits are stored if the target is packed. In the latter case, excess low order digits are truncated.

The precision rules of fixed-point operations are such that no high order digits of the true arithmetic result are lost. Unless the operation is division or the result precision has reached the limits of the machine (59 for decimal or 71 for binary), no low order digits of the true arithmetic result are lost. In the latter case, the precision rules given below indicate exactly when low order digits are lost.

7.3.1.2.1 Prefix Operations

The prefix operators plus and minus yield a result having the type, base, mode and precision of the converted operand. The value of the result of a plus operator is the value of the converted operand. The value of the result of a minus operator for a real operand is the value of the converted operand with its sign reversed. The value of the result of a minus operator for a complex operand is the value of the converted operand with the signs of the real and imaginary parts reversed.

7.3.1.2.2 Infix Operations

If the operation is exponentiation, see Section 7.3.1.2.3.

If the converted operands are floating-point values, the result is a floating-point value. The base and mode are the common base and mode of the converted operands, while the precision of the result is the greater of the precisions of the two operands.

If the converted operands are fixed-point values, the result depends on the operator and the converted operands as described by the following:

> Let N be 71 if the common base is binary and let N be 59 if the common base is decimal.
>
> Let (p,q) be the precision of the first operand, and (r,s) be the precision of the second operand.
>
> If the operation is addition or subtraction the result is a fixed-point value whose base and mode are the common base and mode of the converted operands. The precision of the result is:
>
> (min(N,max(p-q,r-s)+max(q,s)+1),max(q,s))
>
> The value of the result is the sum or the difference of the two operands.

If the operation is multiplication, the result is a fixed-point value whose base and mode are the common base and mode of the converted operands. The precision of the result is:

(min(N,p+r+1),q+s)

The value of the result is the product of the two operands.

If the operation is division, the result is a fixed-point value whose base and mode are the common base and mode of the converted operands. The precision of the result is:

(N,N-p+q-s)

The value of the result is the quotient of the first operand divided by the second. If the quotient exceeds the precision of the result, the least significant digits of the quotient are truncated to form the result. Note that the result always has the maximum precision allowed by the common base, and that as many fractional digits are preserved as is allowed by the machine. Use of these values as operands of other fixed-point computations can easily lead to situations that produce the fixedoverflow or size conditions.

Example:

1/3+25

This example produces fixedoverflow because the division yields 0.333...3 and when the addition aligns the decimal points, the sum exceeds the limit, N, of the machine.

The divide built-in function described in paragraph 13.2.8 can be used to control the precision of the result of fixed-point division.

*

7.3.1.2.3 Exponentiation

For the exponentiation operator, determine the target attributes and convert the operands as follows:

If the first operand is fixed-point, let (p,q) be its precision. Let N be 71 if the base of the first operand is binary, and let N be 59 if the base of the first operand is decimal.

1. If the first operand is fixed-point, the second operand is a fixed-point <real constant> with a scale factor of zero, the value, E, of the second operand is positive, and $(p+1)*E-1 \leq N$, then the result is fixed-point with the base and mode of the first operand. The precision of the result is:

   ((p+1)*E-1,q*E)

   No conversion is performed on the operands.

2. If the second operand is real, fixed-point, with precision (r,0), and case 1 does not apply, then the result is floating-point with the base and mode of the first operand. The first operand is converted to floating-point. The precision of the result is the precision of the converted first operand. No conversion is performed on the second operand.

3. If neither case 1 nor case 2 applies, then the result is floating-point. The base and mode of the result are determined according to the target attribute rules in section 7.3.1.1. The operands are converted to the target mode, base, and type. The precision of the result is the greater of the precisions of the converted operands.

The result of the exponentiation operation is normally a machine-dependent approximation to X raised to the power Y, where X is the first operand and Y is the second operand. However, there are cases for which X**Y is defined as follows:

If X and Y are real values:

If X<0 and neither case 1 nor case 2 above applies, the error condition is signalled.

If X=0 and Y≤0, the error condition is signalled.

If X=0 and Y>0, the result is 0.

If X>0 and Y=0, the result is 1.

If X is complex and Y is real:

If X=0 and Y>0, the result is 0.

If X=0 and Y≤0, the error condition is signalled.

If X≠0 and Y=0, the result is 1.

If Y is a complex value:

If X=0, the real part of Y>0, and the imaginary part of Y=0, the result is 0.

If X=0 and if the real part of Y≤0 or the imaginary part of Y≠0, the error condition is signalled.

If X≠0 and Y=0, the result is 1.

7.3.2 Bit-string Operators

The bit-string operators are:

^	complement
|	inclusive or
&	and

7.3.2.1 Operand Conversion for Bit-string Operators

Bit-string operators require bit-string operands and force conversion of their operands to bit-strings according to the rules given in Section 8. The lengths of the converted operands are defined by the following rules:

A character-string operand is converted to bit-string of the same length as the character-string.

An arithmetic operand is converted to a bit-string whose length is defined in paragraph 8.2.8.

7.3.2.2 Results of Bit-string Operators

The result of the complement operator is a bit-string value whose length is the length of the converted operand. The result value is the complement of the value of the converted operand (each 1 becomes a 0, and each 0 becomes a 1).

The bit-string infix operators produce a bit-string value whose length is the maximum of the lengths of the two converted operands. Prior to evaluation of the operator, the shorter operand is effectively padded on the right with zero bits until it is the length of the longer operand. Each bit of the result is developed by performing the indicated logical operation on the corresponding bits of the two operands. The following table defines the logical operations for a given bit.

This page intentionally left blank.

First Operand	Second Operand	Result of And	Result of Or
1	0	0	1
1	1	1	1
0	0	0	0
0	1	0	1

7.3.3 Concatenate Operator

The concatenate operator is ||. It is an infix operator that yields either a bit-string or character-string.

7.3.3.1 Operand Conversion for Concatenation

If both operands are bit-strings, no conversion occurs and the result is a bit-string; otherwise, the result is a character-string and both operands are converted to character-strings according to the rules given in Section 8. The lengths of the converted operands are defined by the following rules:

An arithmetic operand is converted to character-string according to the conversion rules given in paragraph 8.2.7.

A bit-string operand is converted to a character-string whose length is the length of the bit-string.

7.3.3.2 Result of Concatenation

The result is a string whose type is the common type of the converted operands and whose length is the sum of the lengths of the converted operands.

The value of the result is the converted value of the first operand concatenated with the converted value of the second operand.

7.3.4 Relational Operators

The relational operators are:

```
=  equal
^= not equal
<  less than
^< not less than
<= less than or equal

>  greater than
^> not greater than
>= greater than or equal
```

7.3.4.1 Operand Conversion for Relational Operators

Comparison is performed between values of the same data type. If the operands are of different types, they are converted according to the following rules:

If either operand is arithmetic or declared with a <numeric picture>, the operands are converted as if the operator were an arithmetic infix operator.

If one operand is a character-string and the other is a bit-string, the bit-string is converted to a character-string whose length is that of the bit-string.

If one operand is an offset value and the other is a pointer value, the offset value is converted to a pointer value.

All conversions are performed according to the rules given in Section 8. No other conversions are performed.

7.3.4.2 Types of Comparison

Comparison is defined for all data types except area data. Except for those cases given in paragraph 7.3.4.1, both operands must be of the same data type.

Character-string, bit-string, and real arithmetic values may be compared using any relational operator. Complex arithmetic, label, format, entry, pointer, offset, and file values can only be compared using the equal and not equal operators.

Arithmetic values and character-string values declared with a <numeric picture> are compared algebraically.

Character-string values, other than those declared with a <numeric picture>, are compared by extending the shorter operand to the length of the longer operand by padding the shorter on the right with blank characters. The two strings are then compared from left-to-right using the ASCII collating sequence as given in the MPM Reference Guide.

Bit-string values are compared by extending the shorter operand to the length of the longer operand by padding the shorter on the right with zero bits. The two operands are then compared from left-to-right with 0 comparing less than 1.

Label values compare equal only when they identify the same <statement> and the same block activation record. Refer to Section 3 and paragraph 4.1. Note that a label value that identifies a <label prefix> on a <null statement> does not compare equal to a label value that identifies any other <statement>.

Format values compare equal only when they identify the same <statement> and the same block activation record. Refer to Section 3 and paragraph 4.1.

Entry values compare equal only when they identify the same entry and the same block activation record. Note that multiple <label prefix>s on an <entry statement> or <procedure statement> do not produce entry values that compare equal because each <label prefix> results in the creation of a unique <entry statement>.

Pointer values compare equal only when they identify the same generation of storage, or when they are both null.

Offset values compare equal when they identify the same generation of storage in a given area. They also compare equal if they identify generations of storage in two different areas whose entire history of allocation and freeing is identical. Two areas have identical histories only if one has been assigned to the other and no subsequent allocate or free operations were performed on either area, or if identical sequences of allocate and free operations have been performed on the areas. Two offset values also compare equal when they are both null.

A locator, pointer or offset, value identifying a generation of a structure variable or area variable does not necessarily compare equal to a locator value identifying the first member of the structure or first generation allocated in the area. A locator, pointer or offset, value identifying a generation of an array or array of structures does not necessarily compare equal to a locator value that identifies the first element in the array. Programs that depend on such comparisons are in error.

File values compare equal only if they identify the same file-state block.

7.3.4.3 Results of Relational Operators

Relational operators compare the values of their operands and yield a bit-string of length 1. The value of the result is "1"b if the relationship is true; otherwise, the value of the result is "0"b.

SECTION 8

CONVERSION OF DATA TYPES

As defined in Section 4, a data type is a set of values. Each value is a member of only one such set. A value conforms to a data type if it is a member of that set. If a value does not conform to the data type required by the context in which the value appears, it is converted to the required data type. If conversion from the original data type to the required data type is not defined, the program is in error.

8.1 Contexts That Force Conversion

Each of these contexts forces values to be converted. The resultant data type is called the target data type.

1. The value of the <expression> of an <assignment statement> is converted to a value that conforms to the data type of the <target> of the <assignment statement>.

2. The value of an argument of a <function reference> or <call statement> is converted to conform to the data type given in the corresponding <parameter descriptor> of the entry declaration.

3. The value of an argument to a built-in function is converted to conform to the data type required by the function. Since some built-in functions are generic and others do not allow conversion, some built-in functions do not convert their arguments. Refer to Sections 13 for a description of built-in functions.

4. The operands of an operator are converted to conform to a data type determined by the operator. Infix operators convert their operands to a data type determined by the unconverted data types of both operands. Refer to Sections 7 for the rules used to determine the target data type for operand conversion.

5. A value of a <subscript>, <pagesize option>, <linesize option>, <skip option>, <line option>, <area size>, string <length>, <ignore option>, <position>, or array <bound> is converted to a fixed-point, binary, real, integer.

6. The value of a <locator qualifier> is converted to a pointer value.

7. The value of a <string option> of a <get statement> is converted to a character-string.

8. The value of a <key option> or <keyfrom option> is converted to a character-string.

9. The value of a <title option> is converted to a character-string.

10. A value placed in an output data stream by a <put statement> is converted to a character-string.

11. A value extracted from an input data stream by a <get statement> is converted to conform to the data type of the list element to which it is assigned. If the conversion is controlled by a <data format>, the character-string value from the data stream is first converted to the data type specified by the <data format> and is then converted to the data type specified by the list element to which it is assigned.

12. The values of all <expression>s in a <format specification list> are converted to fixed-point, binary, real, integers.

13. The value of a <return value> is converted to conform to the data type given in the <returns attribute> of the <entry statement> or <procedure statement> whose execution created the current block activation.

14. The value of the <expression> in an <if statement> is converted to a bit-string.

15. The value of a <while expression> is converted to a bit-string.

16. Each value assigned to the <index> of a <multiple do> is converted to conform to the data type of the <index>.

17. The value of the <expression> in an <extent expression> is converted to conform to the data type of the <reference> in the <refer option> of the <extent expression>.

18. The value of the <expression> in an <extent expression> is converted to a fixed-point, binary, real, integer.

19. The value of each <expression> in a <substr pseudo> is converted to a fixed-point, binary, real, integer.

20. The value of a <factor> in an <initial attribute> is converted to a fixed-point, binary, real, integer.

21. The value of an <initial value> in an <initial attribute> is converted to conform to the data type of the variable of which it is an initial value.

8.2 Conversion Rules

The language defines the following kinds of conversion:

Pointer to offset
Offset to pointer
Character-string to arithmetic
Character-string to bit-string
Bit-String to arithmetic
Bit-String to character-string
Arithmetic to character-string
Arithmetic to bit-string
Arithmetic mode conversion
Arithmetic type, base, and precision conversion
Format controlled conversion
Picture controlled conversion

No conversions are defined for label, entry, format, file, or area data.

8.2.1 Pointer to Offset Conversion

A pointer value identifying a generation of a based variable allocated within an area, A, is converted to an offset value identifying that same generation. In order for the conversion to occur, either the offset must have been declared with an <offset attribute> containing a <reference> that identifies A, or the conversion must have resulted from a <reference> to the "offset" built-in function whose second argument was A.

8.2.2 Offset to Pointer Conversion

An offset value identifying a generation of a based variable allocated within an area, A, is converted to a pointer value identifying that same generation. In order for the conversion to occur, either the offset must have been declared with an <offset attribute> containing a <reference> that identifies A, or the conversion must have resulted from a <reference> to the "pointer" built-in function whose second argument was A.

8.2.3 Character-String to Arithmetic Conversion

If the target has any of the type, base or mode omitted, the missing <attribute>s are taken from the set: fixed, decimal, real. If the target precision is omitted and the target is fixed-point binary, the precision of the target is 71. If the target precision is omitted and the target is floating-point binary, the target precision is 63. If the target precision is omitted and the target is fixed or floating-point decimal, the target precision is 59.

If the string is a null string or contains only blank characters, the value of the result is zero.

If the string is not null or all blank, it must be described by:

<valid string>::= [<blank>...]<numeric constant>[<blank>...]

<numeric constant>::= [+|-]<arithmetic constant>|
[+|-]<real constant>{+|-}<imaginary constant>

If the string is not null, blank, or described by this syntax, the conversion condition occurs.

The character-string value is converted to its intrinsic arithmetic value. That value is then converted to conform to the type, base, mode and precision of the target.

During the conversion from character-string to arithmetic, the conversion, size, overflow or underflow conditions may occur. The conversion condition occurs when the character-string is invalid as previously described. The size condition occurs when the target data type is fixed-point and its precision is insufficient to represent all of the integral digits of the converted value. The underflow or overflow conditions occur when the target data type is floating-point and the value is too small or too large to be represented. Refer to Sections 10 for a full discussion of conditions.

Examples:

Character-string	Result
"5.63"	5 or 5.63 depending on the target
""	0
"10e"	conversion condition

8.2.4 Character-String to Bit-String Conversion

Let X be the string to be converted.

If X is a null character-string, it is converted to a null bit-string, X'; otherwise, it is converted to X', where X' is a bit-string of length n, where n is the length of X. For k=1,2...,n, the kth bit of X' is 0 if the kth character of X is 0, and the kth bit of X' is 1 if the kth character of X is 1. If the kth character of X is neither 0 nor 1, the conversion condition occurs.

If no target length is given, the result is X'.

If the target length is greater than n, X' is extended on the right with zeros until it is the length of the target. The result is the extended value of X'.

If the target length is less than n, the stringsize condition occurs. If the <on unit> returns to the point where the condition was detected, the result is formed by truncating the rightmost n-m bits of X', where m is the length of the target.

Examples:

Character-string	Result
"1011"	"1011"b
""	""b
"10Ø"	conversion condition

8.2.5 Bit-string to Arithmetic Conversion

If the target has any of the type, base or mode omitted, the missing <attribute>s are supplied from the set: fixed, binary, real. If the target precision is omitted and the target is fixed-point binary, the precision of the target is 71. If the target precision is omitted and the target is floating-point binary, the target precision is 63. If the target precision is omitted and the target is fixed or floating-point decimal, the target precision is 59.

If the string is a null string, the value of the result is zero.

If the string is not a null string, the rightmost bit of the bit-string is considered to be the units position of an unsigned binary integer of precision n, where n is the length of the string. The value of that integer is then converted to conform to the type, base, mode and precision of the target. If the target is fixed-point and has insufficient precision to represent the integral digits of the value, the size condition occurs.

Examples:

Bit-String	Result
"101"b	5
""b	0
"0000000"b	0

8.2.6 Bit-string to Character-String Conversion

Let X be the bit-string to be converted.

If X is a null bit-string, it is converted to a null character-string X'; otherwise, it is converted to X', where X' is a character-string of length n, where n is the length of X. For k=1,2,...,n, the kth character of X' is 0 if the kth bit of X is 0 and the kth character of X' is 1 if the kth bit of X is 1.

If no target length is given, the result is X'.

If the target length is greater than n, X' is extended on the right with blanks until it is the length of the target. The result is the extended value of X'.

If the target length is less than n, the stringsize condition occurs. If the <on unit> returns to the point where the condition was detected, the result is formed by truncating the rightmost n-m characters of X', where m is the length of the target.

Examples:

Bit-String	Result
"101"b	"101"
""b	""

8.2.7 Arithmetic to Character-String Conversion

Let X be the arithmetic value to be converted.

If the base of X is decimal, let X' be X; otherwise, convert X to X', where the type and mode of X' are the type and mode of X, and the base of X' is decimal. The precision of X' is given by the rules for base conversion described in paragraph 8.2.10.

Let the precision of X' be (p) if the type of X' is floating-point, and let (p,q) be the precision of X' if the type of X' is fixed-point. Let an intermediate result, S, be defined as a character-string whose value is determined by the following:

If the mode of X' is complex, S is formed by converting the real part of X' to a string, S1, and converting the imaginary part of X' to a string, S2, as if they were real numbers. If the imaginary part of X' is ≥ 0, S is formed by:

S1||"+"||S2

Otherwise, S is formed by:

S1||S2

Before concatenation, an "i" is appended to S2 and all leading blanks in S2 are removed by shifting the nonblank characters of S2 to the left and filling the vacated character positions with blanks.

Example:

ББ-2.9 becomes -2.9iББ

The following rules describe the conversion of X' when its mode is real:

If the type of X' is floating-point, the value of S is the value produced by converting X' under control of a picture of the form:

"-9.v(p-1)9es999"

If the type of X' is fixed-point, the value of S is given by one of the following three cases:

For q=0, the length of S is p+3 and its value is the value of X' converted under control of a picture of the form:

"(p+2)-9v"

For p≥q>0, the length of S is p+3 and its value is the value of X' converted under control of a picture of the form:

"(p-q+1)-9.v(q)9"

For q<0 or q>p, the length of S is p+3+k, where k is the number of digits necessary to represent the value of q with no leading zeros. To form the result S, let S' be the value of X' converted under control of a picture of the form:

"(p)-9vf(-q)"

Let E be the value of -q converted under control of a picture of the form:

"s(k)9"

The result S is formed by S'||"f"||E.

Refer to paragraph 8.2.12 for a discussion of picture controlled conversion.

If the target length is not given, the result is S.

Let n be the length of S and let m be the target length.

If m≥n, m-n blanks are appended to the right of S to form the result.

If m<n, the stringsize condition occurs. If detection of the condition is not enabled or if the <on unit> returns control to the point where the condition was detected, the rightmost n-m characters of S are removed to form the result.

Examples:

Type of X'	Precision of X'	Value of X'	Result
float	(4)	0	ƀ0.000e+000
float	(4)	1.23	ƀ1.230e+000
fixed	(4,0)	0	ƀƀƀƀƀ0
fixed	(4,0)	25	ƀƀƀƀ25
fixed	(4,2)	0	ƀƀƀ0.00
fixed	(4,2)	-12.34	ƀ-12.34
fixed	(4,-2)	0	ƀƀƀƀ0f+2
fixed	(4,-2)	123000	ƀ1230f+2
fixed	(5,6)	0	ƀƀƀƀƀ0f-6
fixed	(5,6)	-.01	-10000f-6
fixed	(3,3)	0	ƀ0.000
fixed	(3,3)	-.01	-0.010

8.2.8 Arithmetic to Bit-string Conversion

Let X be the arithmetic value to be converted.

Let X' be a real, fixed-point, binary value of precision (p,0), where p is given by the following:

Attributes of X	Value of p
binary fixed (r,s)	min(71,max(r-s,0))
decimal fixed (r,s)	min(71,max(ceil((r-s)*3.32),0))
binary float (r)	min(71,r)
decimal float (r)	min(71,ceil(r*3.32))

The functions "min", "max" and "ceil" are described in Section 13.

The value of X' is the absolute value of the real part of X.

The size condition occurs if the precision of X' is such that it cannot represent the integral digits of the real part of X.

Let S be a bit-string of length p whose value is the string of binary digits that represent the value of X'.

If the target length is not given, the result is S.

Let n be the target length.

If n≥p, n-p zero bits are appended to the right of S to form the result.

If n<p, the stringsize condition occurs. If the <on unit> returns control to the point where the condition was detected, the rightmost p-n bits of S are removed to form the result.

Examples:

Value of X	Value of X'	Precision of X'	Result
5	5	(4,0)	"0101"b
-4	4	(4,0)	"0100"b
0.7	0	(4,0)	"0000"b
.7	0	(0,0)	""b
01b	1	(2,0)	"01"b

8.2.9 Arithmetic Mode Conversion

If a complex value is converted to a real value, the result is the real part of the complex value.

If a real value is converted to a complex value, the result is a complex value whose real part is the unconverted real value and whose imaginary part is zero.

If the base, type or precision of the converted value is not that of the target, it is converted to conform to the target according to the rules for base, type, and precision conversion.

Examples:

5+2i	becomes	5
5	becomes	5+0i

8.2.10 Arithmetic Type, Base and Precision Conversion

Let X be the arithmetic value to be converted. If the type of X is fixed-point, let (p,q) be the precision of X; otherwise, let (p) be the precision of X.

Let the result X' have the type and base of the target. If the type of the target is not given, let X' have the type of X. If the base of the target is not given, let X' have the base of X. If the precision of the target is given, let the precision of X' be the precision of the target; otherwise, the precision of X' is given by the following table:

Attributes of X	Attributes of X'	Precision of X'
fixed binary	fixed binary	(p,q)
fixed decimal	fixed binary	(min(ceil(p*3.32)+1,71), ceil(q*3.32))
float binary	fixed binary	(p,0)
float decimal	fixed binary	(p,0)
fixed binary	fixed decimal	(min(ceil(p/3.32)+1,59), ceil(q/3.32))
fixed decimal	fixed decimal	(p,q)
float binary	fixed decimal	(min(ceil(p/3.32)+1,59),0)
float decimal	fixed decimal	(p,0)
fixed binary	float binary	(min(p;63))
fixed decimal	float binary	(min(ceil(p*3.32),63))
float binary	float binary	(p)
float decimal	float binary	(min(ceil(p*3.32),63))
fixed binary	float decimal	(min(ceil(p/3.32),59))
fixed decimal	float decimal	(min(p,59))
float binary	float decimal	(min(ceil(p/3.32),59))
float decimal	float decimal	(p)

The value of X' is the value of X converted to the data type of X'. In most cases, the value of X' is the same value as X, but if the base of X' differs from the base of X, the value of X' is an approximation to the value of X. If the base of X' differs from the base of X, rounding occurs if X' is floating point, while truncation occurs if X' is fixed point.

The overflow or underflow condition occurs if X' is floating-point binary and X is a decimal number too large or too small to be represented by binary floating-point. The size condition occurs if X' is fixed-point and has insufficient precision to represent the integral digits of the value.

Examples:

Attributes of X	Attributes of X'
fixed decimal prec(7,0)	fixed binary prec(25,0)
fixed binary prec(17,0)	fixed decimal prec(7,0)
float decimal prec(10)	float binary prec(34)
float binary prec(27)	float decimal prec(9)

8.2.11 Format Controlled Conversion

Format controlled conversion occurs only when a <get statement> or <put statement> containing a <get edit> or <put edit> is executed.

When a <format specification> is used to control conversion from a character-string, it is described as input conversion, and when it is used to control conversion to a character-string, it is described as output conversion.

The result of an input conversion is assigned to a list element and, consequently, is converted to conform to the data type of the list element. Refer to Section 12 for the syntax and semantics of <statement>s.

8.2.11.1 Fixed-Point Format

Syntax:

<fixed-point format>::= f(<w>[,<d>[,<k>]])

<w>::= <expression>

<d>::= <expression>

<k>::= <expression>

Evaluation of the width, <w>, the decimal location, <d>, and the scale factor, <k>, must yield scalar arithmetic or string values that are converted to real binary integers. Let w, d and k be the converted values. Both w and d must be nonnegative.

8.2.11.1.1 Fixed-Point Input Conversion

Let S be the character-string to be converted to the result X. The length of S is w.

If w=0 or S is a string of all blanks, the result is a real, fixed decimal 0 with precision 1 and scale 0.

If w≠0 and S is not all blanks, S must be described by:

[<blank>]...[+|-]<decimal number>[<blank>]...

The string, S, is converted to a fixed-point, decimal, real number, X, whose value, V, and precision, (p,q), are determined as follows:

V is the value of the integer represented by S. The decimal point, if any, is ignored for this purpose.

p is the number of digits in S.

q is calculated as j-k where:

j is the number of digits in S following the decimal point, if one occurs; or j is the value of d, if it appears; or j is 0.

k is given by the <fixed-point format> or is zero.

The value of q must be in the range $-128 \leq q \leq 127$.

The result is X.

Otherwise, the conversion condition occurs. The value of the onsource built-in is S. The value of the onchar builtin is the leftmost character in S that does not meet the syntax for fixed-point input conversion.

Examples:

Value of S	Format	Result
ß7.2ß	f(5)	7.2
ßßßßß	f(5)	0
-7ßßß	f(5)	-7
ß10.5	f(5,2)	10.5
ß100ß	f(5,2)	1.00
ßßßß7	f(5,0,2)	700
ß100ß	f(5,4,-2)	.0001

8.2.11.1.2 Fixed-Point Output Conversion

Let X be the value to be converted to the result string S. The length of S is w. If d is omitted, let d be zero. If any expression in any of the following <picture>s is negative, the size condition occurs.

If w=0, S is a null string.

If d=0 and X<0, let s be min(59,w-1). The value of S is the value of X converted under control of a <picture> of the form:

"(w-s-1)b(s-1)--9v"

If d=0 and X>0, let s be min(59,w). The value of S is the value of X converted under control of a <picture> of the form:

"(w-s)b(s-1)z9v"

If d≠0 and X<0, let s be min(59,w-2). The value of S is the value of X converted under control of a <picture> of the form:

"(w-s-2)b(s-d-1)--9.v(d)9"

If d≠0 and X≥0, let s be min(59,w-1). The value of S is the value of X converted under control of a <picture> of the form:

"(w-s-1)b(s-d-1)z9.v(d)9"

Although the description of <fixed-point picture> editing in paragraph 8.2.12.3.1 specifies that low order digits are truncated when the conversion is performed for a <fixed-point picture>, the remaining low order digit is rounded if it is followed by a digit ≥ 5. If k is given, the conversion to decimal performed by the <fixed-point picture> effectively multiplies the decimal value X' to be edited into the picture by 10**k.

Note that the scaling performed by a <fixed-point format> effectively multiplies the value being converted by a power of ten for both input and output conversions. It differs from the scaling performed by the <picture scale factor> which effectively multiplies by a power of ten for input and divides by a power of ten on output. Refer to paragraph 8.2.12.

The result is S.

Examples:

Value of X	Format	Result
7.5	f(5,2)	␣7.50
0	f(5,2)	␣0.00
-3	f(5,2)	-3.00
3.5	f(5)	␣␣␣4
0	f(5)	␣␣␣0
-7.5	f(5)	␣␣-8
12	f(5,0,2)	␣1200

8.2.11.2 Floating-Point Format

Syntax:

<floating-point format>::= e(<w>[,<d>[,<s>]])

<w>::= <expression>

<d>::= <expression>

<s>::= <expression>

Evaluation of the width, <w>, the decimal location, <d>, and the number of significant digits, <s>, must yield scalar arithmetic or string values that are converted to real binary integers. Let w, d and s be the converted values. All three values must be nonnegative. If given, w, d, and s must satisfy $0<s\leq 59$, $s\geq d\geq 0$, $w\geq 0$.

8.2.11.2.1 Floating-Point Input Conversion

Let S be the character-string to be converted to the result X. The length of S is w.

If w=0 or S is a string of all blanks, the result is a real, fixed, decimal 0 with precision 1.

If w≠0 and S is not all blank, S must be described by:

[<blank>]...[+|-]<decimal number>
[{[e]{+|-}|e}<decimal integer>][<blank>]...

The string, S, is converted to a floating-point, decimal, real number, X, whose mantissa, f, exponent, e, and precision, p, are determined as follows:

p is the number of digits in the <decimal number>.

f is the integer value of the <decimal number>, ignoring the decimal point, if any.

e is calculated as k-q where:

k is the value of the <decimal integer> or is zero.

q is the number of digits following the decimal point in S, if one appears; or q is the value of d, if it is given; or q is zero.

The value of e must be in the range $-128\leq e\leq 127$.

The result is X.

Otherwise, the conversion condition occurs. The value of the onsource built-in is S. The value of the onchar builtin is the leftmost character in S that does not meet the syntax for floating-point input conversion.

Examples:

Value of S	Format	Result
ßßßßß	e(5,3)	0
1.3e7	e(5,3)	1.3e+7
12345	e(5,3)	12.345e+0
-52+4	e(5)	-52.0e+4
ßß5ßß	e(5)	5.0e+0

8.2.11.2.2 Floating-Point Output Conversion

Let X be the value to be converted to the result string S. The length of S is w.

If s is omitted, let s be d+1. If d is omitted, let d be p-1 and let s be p, where p is the precision of X after conversion to a floating-point, decimal, real number according to the rules given in paragraph 8.2.10.

The value of S is determined by one of the following cases:

If w=0, S is a null string.

If d<s and d≠0 and X<0, the value of S is the value of X converted under control of a <picture> of the form:

"(w-s-7)b(s-d)-9.v(d)9es999"

If d<s and d≠0 and X≥0, the value of S is the value of X converted under control of a <picture> of the form:

"(w-s-6)b(s-d-1)z9.v(d)9es999"

If d=0 and X<0, the value of S is the value of X converted under control of a <picture> of the form:

"(w-s-6)b(s)-9ves999"

If d=0 and X≥0, the value of S is the value of X converted under control of a <picture> of the form:

"(w-s-5)b(s-1)z9ves999"

If d=s and d≠0 and X<0, the value of S is the value of X converted under control of a <picture> of the form:

"(w-d-8)b-..v(d)9es999"

with the resulting ".." replaced by "0.".

If d=s and d≠0 and X≥0, the value of S is the value of X converted under control of a <picture> of the form:

"(w-d-7)b..v(d)9es999"

with the resulting ".." replaced by "0.".

If the leading expression in any of the preceding <picture>s is negative, the size condition occurs.

The result is S.

Examples:

Value of X	Format	Result
7.5	e(11,3)	ß7.500e+000
-7.5	e(11,3)	-7.500e+000
75	e(11,3)	ß7.500e+001
0	e(11,3)	ß0.000e+000
.008	e(11,2,4)	ß80.00e-004

8.2.11.3 Complex Format

Syntax:

<complex format>::= c(<format part>[,<format part>])

<format part>::= <picture format>|<fixed-point format>|
<floating-point format>

If only one <format part> is given, it is used to control the conversion of both the real and imaginary parts of the complex number. If two <format part>s are given, the first controls the conversion of the real part of the complex number and the second controls the imaginary part of the complex number. The conversions of the two parts are performed independently as described for real numbers in this section.

If a <format part> is a <picture format>, the <picture> must be a <numeric picture>.

Note that an "i" does not appear in the character-string representations of complex numbers processed by a <complex format>.

Examples:

Value	Format	Input Result
ßß3ßß2	c(f(3))	3+2i
-2ßßß1	c(f(3,1),f(3))	-.2+1i

Value	Format	Output Result
5+2i	c(f(3))	ßß5ßß2
5.2-3.1i	c(f(4,1),f(5,2))	ß5.2-3.10

8.2.11.4 Character-String Format

Syntax:

<character-string format>::= a[(<w>)]

<w>::= <expression>

Evaluation of <w> must yield a scalar arithmetic or string value that is converted to real binary integers. Let w be the converted value. If specified w must be nonnegative.

For input conversion, w must be given. No conversion is performed and the result is a character-string of length w.

For output conversion, let X be the value to be converted to the result string S. It is converted to a character-string, S', according to the rules given in section 8.2. If w is not given, the result is S'. If w is given, let n be the length of S'. If n≤w, the result is S' with w-n blanks appended to its right. If n>w, the stringsize condition occurs. If detection of the condition is disabled or if the <on unit> returns to the point where the condition was detected, the rightmost n-w characters are removed from S' to form the result.

Examples:

Value	Format	Input Result
ßßßß	a(4)	"ßßßß"
ß2.5	a(4)	"ß2.5"

Value	Format	Output Result
"abc"	a	abc
"abc"	a(4)	abcß
""	a(4)	ßßßß

8.2.11.5 Bit-string Format

Syntax:

<bit-string format>::= <radix factor>[(<w>)]

<radix factor>::= {b|b1|b2|b3|b4}

<w>::= <expression>

Evaluation of <w> must yield a scalar arithmetic or string value that is converted to real binary integers. Let w be the converted value. If specified, w must be nonnegative.

8.2.11.5.1 Bit-string Input Conversion

For input conversion, w must be specified. Let S be the character-string of length w that is to be converted. S must be described by the following:

[<blank>...][<character>...][<blank>...]

The <character>s in the above description must come from the <character>s in the table in paragraph 2.6.2.1 corresponding to the specified <radix factor>. If S does not satisfy this syntax and constraint, the conversion condition occurs. Let S' be S with all of its leading and trailing blanks removed and let n be the length of S'.

Let m be 1 if the <radix factor> is "b", or the number in the <radix factor> otherwise. If S' is a null character-string, it is converted to a null bit-string, R; otherwise, it is converted to R, where R is a bit-string of length m*n. For k=1,2,...,n, bits k*m-m+1,...,k*m are obtained from the table in paragraph 2.6.2.1. If the kth character of S is invalid, the conversion condition occurs.

Examples:

Value	Format	Input Result
010	b(3)	"010"b
ßßß	b(3)	""b
000	b(3)	"000"b
ß1ß	b(3)	"1"b
407	b3(3)	"100000111"b
cd5	b4(3)	"110011010101"b

8.2.11.5.2 Bit-string Output Conversion

For output conversion, w is optional. Let X be the value to be converted to the resulting character-string S. X is converted to a bit-string, B, according to the rules given in paragraph 8.2. Let n be the length of B. Let m be the number specified in the <radix factor> or 1 if no number was specified. Let n'=n. If n is not a multiple of m, let n' be the next higher multiple of m, and extend B by appending n'-n zero bits to the right of B. Let K=n'/m. If w is not specified, let w be K.

B is converted to a character-string S of length w as follows. If B is a null bit-string it is converted to a null character-string S; otherwise, it is converted to S, where S is a character-string of length K. For i=1,2,...,K, bits i*m-m+1,...,i*m are converted to the ith character by using the table in paragraph 2.6.2.1.

If w is greater than K, S is extended on the right with blanks until its length is w.

If K is greater than w, the stringsize condition occurs. If the <on unit> returns to the point where the condition was detected, the result is formed by truncating the rightmost K-w characters of S.

Examples:

Value	Format	Output Result
"00"b	b	00
"1"b	b(4)	1ßßß
""b	b(4)	ßßßß
"10101"b	b3(3)	52ß
"11111"b	b4(2)	F8

8.2.11.6 Picture Format

Syntax:

<picture format>::= p"<picture>"

For input conversion, the character-string to be converted by a <picture format> must be a valid string as defined in paragraph 8.2.12. If this constraint is not satisfied, the conversion condition occurs. For valid strings, no actual conversion occurs and the result of input conversion is the original character-string.

This page intentionally left blank.

Note that for input conversion the result of the format controlled conversion is considered a pictured value and is converted as such when it is assigned to the list element.

For output conversion, let X be the value to be converted to the pictured string S. X is converted to S as described in the next section.

8.2.12 Picture Controlled Conversion

The following sections describe the conversion that occurs when a value is assigned to a pictured variable or output through a <picture format> as editing. The conversion that occurs when a pictured value is converted to an arithmetic value is described as encoding.

The pictured character-string value described by a <picture> consists of n characters, where n is the number of <picture char>s in the <normal picture> excluding any "v", "k", or <picture scale factor>, but including all <insertion character>s.

The result of editing is a pictured character-string of length n whose value is determined by the <picture>. The result of encoding is a decimal arithmetic value whose type and precision are determined by the <picture>. If the encoded value does not conform to the data type of the target, the encoded value is converted to conform to the target.

The character-string value to be encoded must be a valid string. A valid string is one of the strings that could have been produced by editing values through the <picture>, except that the set of strings acceptable to the <mantissa field> of a <floating-point picture> is the set of strings that could have been produced by editing values through the <mantissa field> as if it were a <fixed-point picture>.

If a pictured variable or function value is declared with the <complex attribute>, the encoding and editing operations are performed on the real and imaginary parts of the complex values as if they were real numbers. The single <picture> is effectively a pair of identical <picture>s.

8.2.12.1 Syntax of Pictures

Syntax:

```
<picture>::= {[(<decimal integer>)]<picture char>}...
     [<picture scale factor>]

<picture char>::= a|b|c|d|e|k|r|s|v|x|y|z|$|9|+|-|.|,|/|*

<picture scale factor>::= f([+|-]<decimal integer>)
```

This syntax describes all valid <picture>s, but is too permissive in that it also describes many invalid <picture>s. In order to describe only valid <picture>s, <picture>s must be translated into <normal picture>s. This translation is accomplished by copying each <picture char> k times, where k is the value of the parenthesized <decimal integer> that immediately precedes the <picture char>. If no such parenthesized <decimal integer> appears, the <picture char> is not repeated. If k=0, the <picture char> is removed.

Example:

```
(5)9v(2)9      becomes   99999v99
(3)-9.(4)9     becomes   ---9.9999
(0)-99         becomes   99
```

Normalized pictures must be described by the following syntax as amended by the discussion of <insertion character>s that follows below:

Syntax:

```
<normal picture>::= <character picture>|<numeric picture>

<character picture>::= [9...]{a|x}[a|x|9]...

<numeric picture>::= {<fixed-point picture>|
     <floating-point picture>}[<picture scale factor>]

<picture scale factor>::= f([+|-]<decimal integer>)

<fixed-point picture>::= <fixed field>|<drifting field>

<fixed field>::= <digit positions>[s|+|-][$]|
     <digit positions>[$][s|+|-]|
     [s|+|-]<digit positions>[$]|
     [s|+|-][$]<digit positions>|
     [$][s|+|-]<digit positions>|
     [$]<digit positions>[s|+|-]|
     <digit positions>[$]{cr|db}|
     [$]<digit positions>{cr|db}

<digit positions>::= <digits>[v[<digits>]]|
     v<digits>|
     z...[<digits>][v[<digits>]]|
     [z...]v z...|
     *...[<digits>][v[<digits>]]|
     [*...]v *...

<drifting field>::= <drifting sign>[$]|
     [$]<drifting sign>|
     <drifting dollar>[s|+|-]|
     [s|+|-]<drifting dollar>|
     <drifting dollar>{cr|db}

<drifting sign>::= <signs>[<digits>][v[<digits>]]|
     s...v s...|+...v +...|-...v -...

<drifting dollar>::= $ $...[<digits>][v[<digits>]]|$...v $...

<digits>::= {9|y}...

<signs>::= s s...|+ +...|- -...

<floating-point picture>::= <mantissa field>
     {e|k}<exponent field>

<mantissa field>::= [s|+|-]<digit positions>|<drifting sign>

<exponent field>::= [s|+|-][[9][9]9|[z][9]9|[z][z]9|[z][z]z}
```

If a <picture> can be translated into a <character picture> by expanding all repeated <picture char>s, or into a <numeric picture> by expanding all repeated <picture char>s and removing all <insertion character>s, it is a valid <picture>; otherwise, it is not valid and the program is in error.

```
<insertion character>::= .|,|/|b
```

Although the presence of <insertion character>s is described informally and not by syntax rules, they are part of the <fixed-point picture> or <floating-point picture> and occupy positions in the pictured character-string value described by the <picture>.

8.2.12.2 Character Picture Conversion

A <character picture> can contain only "9", "a" or "x" <picture char>s and it must contain at least one "a" or "x".

8.2.12.2.1 Character Picture Editing

Let X be the value to be edited into the pictured character-string P.

X is converted to a character-string value X' according to the rules for the conversion to character-string given in paragraph 8.2. The value of X' is then edited into the pictured character-string as follows:

Let n be the length of P and let m be the length of X'. If m<n, X' is extended on the right by n-m blanks. If m>n, the stringsize condition occurs. If detection of the condition is disabled or if the <on unit> returns control to the point where the condition was detected, the last m-n characters of X' are ignored.

The value of P is the leftmost n characters of the extended value of X'.

For k=1,2,...,n, the kth character of X' is checked for conformance to the kth <picture char> and is assigned to the kth character position of P. Only the characters "0","1",...,"9" or blank conform to a "9" <picture char>. Only the characters "a","b",...,"z" or "A","B",...,"Z" or blank conform to an "a" <picture char>. Any character conforms to an "x" <picture char>.

If any character of X' does not conform, the conversion condition occurs. The value of P is not defined when this condition occurs.

Examples:

Value of X'	Picture	Result
abc	aaa	abc
123	x99	123
1e2	x99	conversion condition

8.2.12.2.2 Character Picture Encoding

The pictured character-string value is converted to an arithmetic value according to the rules for character-string to arithmetic conversion given in paragraph 8.2.3.

8.2.12.3 Fixed-Point Picture Conversion

A <fixed-point picture> cannot contain a "k", "e", "a" or "x" <picture char>.

There are two kinds of <fixed-point picture>s, <fixed field> pictures and <drifting field> pictures.

8.2.12.3.1 Fixed-Point Picture Editing

Let X be the value to be edited into the pictured character-string, P.

X is converted to a fixed-point, decimal, real, value, X', of precision (n,m), where (n,m) is determined according to the rules for <fixed-point picture> encoding given in paragraph 8.2.12.3.2.

If this precision is insufficient to retain all digits to the left of the decimal point, the size condition occurs. If fractional digits are lost, they are truncated.

The value of X' is converted to a character-string by the following:

Let D be the string of n decimal digits that represents the absolute value of X'.

Let P be a copy of the <normal picture> with the "v" and the <picture scale factor>, if any, removed. Let N be the number of <picture char>s in P, let j be 1, and let zero suppression be off.

For k=1,2,...,N, select the kth <picture char> from P and perform the action indicated for this <picture char>. If the kth <picture char> in the original <normal picture> is a "v" and zero suppression is on and X'≠0, turn zero suppression off before performing the action indicated for the kth <picture char> of P.

s If X'<0, replace the "s" with a "-"; otherwise, replace it with a "+". If additional "s" characters remain, replace each of them with a "z" and turn zero suppression on.

+ If X'<0, replace the "+" with a blank; otherwise, it remains unchanged. If additional "+" characters remain, replace each of them with a "z" and turn zero suppression on.

- If X'<0, the "-" is unchanged; otherwise, replace it with a blank. If additional "-" characters remain, replace each of them with a "z" and turn zero suppression on.

$ Leave this character unchanged. If additional "$" characters remain, replace each of them with a "z" and turn zero suppression on.

9 Replace the "9" by the jth digit of D and turn zero suppression off. Let j be j+1.

y Turn zero suppression off. If the jth digit of D is a zero, replace the "y" by a blank; otherwise, replace the "y" by the jth digit of D. Let j be j+1.

z If this is the first "z" and it occurs to the left of the "v" in the original <normal picture>, turn zero suppression on. If zero suppression is on and the jth digit of D is a zero, replace the "z" by a blank; otherwise, turn zero suppression off and replace the "z" by the jth digit of D. Let j be j+1.

* If this is the first "*" and it occurs to the left of the "v" in the original <normal picture>, turn zero suppression on. If zero suppression is on and the jth digit of D is a zero, leave the "*" unchanged; otherwise, turn zero suppression off and replace the "*" by the jth digit of D. Let j be j+1.

c The next <picture char> must be an "r". If X'<0, leave both the "c" and the "r" unchanged; otherwise, replace them by two blanks.

d The next <picture char> must be a "b". If X'<0, leave both the "d" and the "b" unchanged; otherwise, replace them by two blanks.

, If zero suppression is on and the previous character in P is now an "*", replace the "," by an "*". If zero suppression is on and the previous character in P is not an "*", replace the "," by a blank. If zero suppression is off, leave the "," unchanged.

/ Process like a comma.

. Process like a comma.

b Process like a comma, except when zero suppression is off replace the "b" by a blank.

Before obtaining the result, the longest subfield contained within a <drifting sign> or <drifting dollar> satisfying this syntax:

{+|-|$}<blank>...

has its first and last characters interchanged.

If no digits were edited into P, set P to all blanks. The result is the character-string P.

Examples:

Value of X	Picture	Result
5.2	99v99	0520
5.2	99.99	00.05
5.2	9.9.99	0.0.05
5.2	sssv99	ß+520
5.2	sssv.99	ß+5.20
5.2	---v.99	ßß5.20
5.2	+++v.99	ß+5.20
-5.2	sssv.99	ß-5.20
-5.2	---v.99	ß-5.20
-5.2	+++v.99	ßß5.20
-5.2	$$$v.99cr	ßß$5.20cr
5.2	$$$v.99cr	ßß$5.20ßß
5.2	zzzvzz	ßß520
.01	zzzvzz	ßßß01
0	zzzvzz	ßßßßß
1234	z,zzv	1,234
900	z,zzv	ßß900

8.2.12.3.2 Fixed-Point Picture Encoding

Let X be the pictured character-string value to be encoded and let (n,m) be the precision of the encoded value, Y, where (n,m) are determined as follows:

If the <fixed-point picture> is a <fixed field>, let n be the number of <picture char>s in the <digit positions> excluding any <insertion character>s or the "v". Let m be the number of <picture char>s in the <digit positions> following the "v" and excluding any <insertion character>s. If the "v" is omitted, m=0.

If the <fixed-point picture> is a <drifting field>, let n be the number of <picture char>s in the <drifting sign> or <drifting dollar> excluding: the first sign of a <drifting sign>, the first "$" of a <drifting dollar>, any <insertion char>s, and the "v". Let m be the number of <picture char>s in the <drifting sign> or <drifting dollar> following the "v" and excluding any <insertion character>s. If the "v" is omitted, m=0.

The resulting values of n and m must satisfy The relationship $m \leq n \leq 59$ or the program is in error.

If the <picture> has a <picture scale factor>, m is changed to m-q, where q is the value of the <picture scale factor>. The final value of m must be in the range $-128 \leq m \leq 127$.

Let D be the string of decimal digits contained within X. If D is a null string, let D' be zero; otherwise, let D' be the decimal integer represented by D. To form the result, Y, let the absolute value of Y be D' and let the sign of Y be minus if X contains a "-", "cr" or "db"; otherwise, the sign of Y is plus. The data type of Y is fixed-point, real decimal, of precision (n,m).

The value of Y is the result of encoding X.

Examples:

Value of X	Picture	Result
12,345	zz,zzz	12345
123456	zz,zzz	program in error
ßßß900	zz,zzz	900
ßß5.00	zzzv.zz	5.00
ßß5.00	zzz.zz	500
ß-5.00	sssv.99	-5.00
ß+5.00	sssv.99	5.00
ß-5.00	---v.99	-5.00
ß+5.00	---v.99	program in error
ß+5.00	+++v.99	5.00
ß-5.00	+++v.99	program in error
ßß$5.2	$$$9v.9	5.2
ßßß5.2	$$$9v.9	program in error
12.23cr	zzv.99cr	-12.23
12.23ßß	zzv.99cr	12.23

8.2.12.4 Floating-Point Picture Conversion

A <floating-point picture> consists of two subfields, one describing the mantissa and one describing the exponent. A <floating-point picture> cannot contain an "a" or "x" <picture char>.

8.2.12.4.1 Floating-Point Picture Editing

Let X be the value to be edited into the pictured character-string P.

X is converted to a floating-point, decimal, real value, X' of precision (n), where (n) is determined according to the rules for <floating-point picture> encoding given in paragraph 8.2.12.4.2.

If a <picture scale factor> is specified for P, the value of X' is changed to X'*10**-k, where k is the value of the <picture scale factor>.

If digits are lost by this conversion, the least significant remaining digit is rounded if it is followed by a digit ≥ 5.

The absolute value of the mantissa of X' is represented as a fixed-point, decimal, real number, D, of precision(n,n) adjusted to lie in the range $(1/10) \leq f < 1$, or is zero. The exponent is adjusted to reflect that fact that the mantissa is adjusted. The exponent is zero if X' is zero.

Consider the <mantissa field> to be a <fixed-point picture> and the mantissa of X' to be a fixed-point, decimal, real value of precision (n,m), where n and m are given by the rules for <fixed-point picture> encoding given in paragraph 8.2.12.3.2. Edit the mantissa of X' into a copy of the <mantissa field> as if it were a <fixed-point picture>. Let M be the result of this edit operation.

Adjust the exponent of X' to reflect the location of the "v" or implied "v" within the original <mantissa field>, and then convert it to a fixed-point, decimal, real number of precision(3,0). Edit the adjusted and converted exponent into a copy of the <exponent field> of the <floating-point picture> as if the <exponent field> were a <fixed-point picture>. Let E be the result of this edit operation.

If the <floating-point picture> contained a "k", the result is M||E. If the <floating-point picture> contained an "e" and E is all blanks, the result is M||"ß"||E. If the <floating-point picture> contained an "e" and E is not all blanks, the result is M||"e"||E.

Examples:

Value of X'	Picture	Result
5.2	9v.99ks99	5.20+00
5.2	9v.99k-99	5.20ß00
5.2	9v.99k+99	5.20+00
5.2	9v.99es99	5.20e+00
5.2	9v.99e9	5.20e0
-5.2	s9v.99e9	-5.20e0
-5.2	+9v.99e9	ß5.20e0
-5.2	-9v.99e9	-5.20e0
-5.2	---v.9es9	-52.0e-1
1234.5	9,999v.e9	1,235.e0
-5.2	---v.---es99	-52.000e-01

8.2.12.4.2 Floating-Point Picture Encoding

Let X be the pictured character-string value to be encoded, and let (n) be the precision of the encoded value, Y', where (n) is determined as follows:

> If the <mantissa field> is a <drifting sign>, n is the number of <picture char>s in the <drifting sign>, excluding the "v", the first sign character, and any <insertion character>s.
>
> If the <mantissa field> is not a <drifting sign>, n is the number of <picture char>s in the <digit positions>, excluding the "v" and any <insertion character>s.
>
> Let m be the number of <picture char>s in the <mantissa field> following the "v", but excluding any <insertion character>s.
>
> The resulting values of n and m must satisfy the relationship $m \leq n \leq 59$ or the program is in error.

Let D be the string of decimal digits contained in X. If D is a null string, let D' be zero; otherwise, let D' be the absolute value of the decimal integer represented by D. To form the result, Y, let Y be a real, decimal, fixed-point value of precision (n,m) whose absolute value is given by D' and whose sign is minus if the first N characters of X contain a "-", and is otherwise plus.

Let I be the value derived by encoding the <exponent field> of X as if it were a <fixed-point picture>. Let I' be I+s, where s is the value of the <picture scale factor>, if there is one, or is 1.

The result, Y', is a floating-point, decimal, real, value of precision (n) whose value is Y*10**I'.

Examples:

Value of X	Picture	Result
1,234.56+0	9,999.v99ks9	1234.56e0
ßß900.00+4	z,zzz.v99ks9	900.00e4
ß-1.234e00	--9.v999e99	-1.234e0
ßßßßßß	9v9999k9	program in error

SECTION 9

PROMOTION OF AGGREGATE TYPES

As defined in paragraph 4.2, an aggregate type is the dimensionality, array-extents and structuring of a set of scalar values. A value conforms to an aggregate type if it has the dimensionality, array-extents and structuring specified by the aggregate type. When a value does not conform to the aggregate type required by the context in which the value appears, it is promoted to the required aggregate type. If promotion from the original aggregate type to the required aggregate type is not defined, the program is in error.

9.1 Contexts That Force Promotion

1. The value of the <expression> of an <assignment statement> is promoted to conform to the aggregate type of the <target> of the <assignment statement>.

2. The value of an argument of a <function reference> or <call statement> is promoted to conform to the aggregate type of the corresponding <parameter descriptor> of the entry declaration.

3. Operands of infix operators are promoted to the higher of their two aggregate types.

4. The value of a <return value> is promoted to conform to the aggregate type specified by the <returns attribute> of the <entry statement> or <procedure statement> whose execution created the current block activation.

5. The arguments of certain built-in functions are promoted to the highest aggregate type of all the given arguments. Refer to Section 13 to see which built-in functions force promotion, and which arguments are promoted.

6. The <expression>s of a <substr pseudo> are promoted to the highest common aggregate type of the operands of the <substr pseudo>.

All of these contexts supply the dimensionality and structuring of the resultant aggregate type. All contexts, except the <argument list> context and the <return value> context, supply the array-extents of the result. If a <parameter descriptor> or a <returns descriptor> specifies asterisk array-extents, the resultant aggregate has an array-extent of one in each dimension; otherwise, the constant array-extents of the <parameter descriptor> or <returns descriptor> supply the array-extents of the result.

Example:

```
declare f entry(dimension(*) fixed);

call f(5);
```

In this example, the scalar 5 is promoted to a one-dimensional array of one element whose value is 5.

9.2 Types of Promotion

The language defines promotion from:

scalar to array
scalar to structure
scalar to array of structures
structure to array of structures

The word "promotion" implies a ranking of aggregate types, and the promotion of the operands of infix operators utilizes this ranking. The aggregate types are ranked as follows:

array of structures	highest
array or structure	equal
scalar	lowest

9.3 Promotion Rules

1. Scalars become arrays by forming an array whose elements each have the scalar value.

2. Scalars become structures by forming a structure whose members each have the scalar value.

3. Scalars become arrays of structures by forming an array of structures whose scalar components each have the scalar value.

4. Structures become arrays of structures by forming an array of structures whose array elements each have the value of the structure.

An array cannot be promoted to a scalar, to an array of different dimensionality or extent, nor can it be promoted to a structure or to an array of structures. However, since the <bound>s of an array valued <expression> are always normalized, arrays of identical extents and dimensionality, but with differing <bound>s can be used in any of the contexts that force promotion without causing promotion to occur. Refer to paragraph 4.2 for a discussion of array <bound>s and normalization.

Example:

```
declare A(5),B(4),C(2,2);
```

In this example, there are no valid promotions between A, B and C.

A structure cannot be promoted to a scalar, to a structure of different shape, nor can it be promoted to an array. It can be promoted to an array of structures. However, since <level>s are normalized, structures of identical shape, but with differing <level>s can be used in any of the contexts that force promotion without causing promotion to occur. Refer to paragraph 4.2 and paragraph 5.2.1.3 for a discussion of the normalization of <level>s.

Example:

```
declare 1 S,2 A,2 B;

declare 1 T,2 X,3 Y,3 Z;
```

In this example, there are no valid promotions between T and S, but X and S have identical structuring and consequently have identical aggregate type. (Their adjusted <level>s are equal.)

The fact that two aggregates may map into equivalent patterns of values in storage has no affect on the rules of aggregate promotion.

Example:

```
declare A(3);

declare 1 S,2 X,2 Y,2 Z;
```

In some implementations, A and S may map into storage in the same manner, but their aggregate types are not compatible and cannot be promoted to a common aggregate type.

SECTION 10

CONDITIONS, SIGNALS AND ON-UNITS

10.1 Conditions and Condition Names

A condition is a state of the executing program. A condition name is a name that identifies a condition. For example, division by zero is a condition identified by the condition name "zerodivide". The language defines a set of condition names each of which identifies a specific condition which can be detected during program execution. The complete list of PL/I conditions is given in paragraph 10.4.

10.2 Condition Prefixes

The <condition prefix> is an optimization/debugging feature that allows the programmer to disable or enable the detection of some of the PL/I conditions.

Syntax:

```
<condition prefix>::= (<prefix name>[,<prefix name>]...):

<prefix name>::= <disabled condition>|<enabled condition>

<enabled condition>::= {conversion|conv}|{fixedoverflow|fofl}|
    {overflow|ofl}|size|{stringrange|strg}|{stringsize|strz}|
    {subscriptrange|subrg}|{underflow|ufl}|{zerodivide|zdiv}

<disabled condition>::= {noconversion|noconv}|{nofixedoverflow|
    nofofl}|{nooverflow|noofl}|nosize|{nostringrange|nostrg}|
    {nostringsize|nostrz}|{nosubscriptrange|nosubrg}|
    {nounderflow|noufl}|{nozerodivide|nozdiv}
```

A <condition prefix> is in error if it contains a <disabled condition> and an <enabled condition> that identify the same condition.

The region of an <external procedure> affected by a <prefix name> is known as the scope of the <prefix name>. The scope of a <prefix name> specified in a <condition prefix> attached to a <begin statement> or <procedure statement> is all <statement>s contained in the <block> defined by the <begin statement> or <procedure statement>, except <statement>s or <block>s that lie within the scope of another <prefix name> identifying the same condition and contained in the same <block>.

The scope of a <prefix name> specified in a <condition prefix> attached to a <statement> other than a <begin statement> or <procedure statement> is restricted to that <statement> and does not include any <block>s or <statement>s that are part of an <if statement> or <on statement>. The scope of a <prefix name> specified in a <condition prefix> attached to a <do statement> is restricted to the <do statement> and does not include the <group> headed by the <do statement>.

A <condition prefix> attached to a <format statement> controls the detection of conditions resulting from the evaluation of the <format specification list>, but

has no effect on the detection of conditions resulting from the execution of the <get statement> or <put statement>.

A <condition prefix> cannot be attached to a <declare statement> or <default statement>. Any <reference>s or <expression>s in a <declare statement> or <default statement> are part of the declarations of one or more names. When a name is referenced during the execution of a <statement>, the <condition prefix> that applies to that <statement> is used to control the detection of conditions during evaluation of <reference>s and <expression>s in the declaration of the name.

The detection of all PL/I conditions is enabled unless it has been explicitly disabled. The detection of a condition is said to be disabled for all <statement>s that lie within the scope of a <prefix name> that identifies the condition with a <disabled condition> name.

If a condition occurs during the execution of a <statement> within the scope of a <condition prefix> that has disabled detection of the condition, the program is in error and the results of further execution are undefined.

The imaginary outer <block> that contains an <external procedure> has a <condition prefix> of the form:

```
(nosize,nostringsize,nostringrange,nosubscriptrange):
```

This establishes a default <condition prefix> that applies to the entire <external procedure>.

10.3 Signals and On-units

When a condition is detected the condition is signalled. A signal causes a <block> activation of the <on unit> most recently established for the condition. The execution of a <signal statement> also signals a condition and has the same effect on the flow of control as the detection of a condition. The execution of a <signal statement> affects the values of some condition built-in functions as described in paragraph 12.27.

An <on unit> is a <begin block> or <independent statement> executed when a condition is signalled. An <on unit> is established by the execution of an <on statement> and is reverted by the execution of a <revert statement> or by termination of the block activation that established it.

Each block activation is capable of establishing a single <on unit> for each condition. If a block activation attempts to establish a second <on unit> for a given condition, the second replaces the first. Each block activation is capable of reverting only those <on unit>s that it established. If a block activation attempts to revert an <on unit> which it did not establish, the <revert statement> behaves like a <null statement>. Refer to Section 12 for the syntax and semantics of the <on statement>, <signal statement>, and <revert statement>.

Example:

```
L1: on zerodivide go to A;

    begin;
    L2: on zerodivide go to B;

    L3: on zerodivide go to C;

    revert zerodivide;
    end;
```

Statement L1 establishes an <on unit> of "go to A" for the zerodivide condition. Statement L2 establishes a new <on unit> of "go to B" for the same condition, and because L2 is part of a different block activation, its <on unit> does not replace that established by L1. It is effectively stacked on top of the <on unit> established by L1. Statement L3 replaces the <on unit> established by L2 because L2 and L3 are <statement>s in the same block activation. The <revert statement> reverts the <on unit> established by L3 and causes the <on unit> established by L1 to be the current <on unit> for the condition.

If no <on unit> has been established for a condition and the condition is signalled, a default <on unit> is invoked which performs the default action described for that condition in paragraph 10.4. A default <on unit> is explicitly established by an <on statement> of the form:

```
on <condition list> system;
```

An <on unit> is invoked as if it were a <procedure>. When control reaches the end of the <on unit> it returns to the point where the condition was detected.

10.3.1 Restrictions

The program is in error and the results of continued execution are undefined if an <on unit> invoked for any of the following conditions returns to the point where the condition was detected.

```
area (if caused by assignment)
error
fixedoverflow
overflow
size
storage (if caused by stack overflow)
stringrange
subscriptrange
zerodivide
```

If a condition is signalled during evaluation of an <expression>, but not during execution of an irreducible function invoked by the <expression>, and the responding <on unit> returns to the point where the condition was signalled, then the <on unit> must not have allocated, freed, or assigned a value to any generation of storage known at the point where the condition was signalled.

This effectively means that conditions are considered to be unexpected side effects of <expression> evaluation and their <on unit>s cannot change the values of variables being used by the interrupted <block> unless the <on unit> executes a <goto statement> to return to the interrupted <block>.

An <on unit> invoked as a result of a condition detected during evaluation of a <statement> cannot access the value of a variable whose value is changed by the execution of the <statement>.

Example:

```
on zerodivide begin;
                X = A;
                go to trouble;
                end;
A = B/C;
```

The value of A is not defined upon entry to the <on unit> and, therefore, cannot be accessed by the <on unit>. Programs which access such values are in error

and the results of continued execution are undefined. This example would be valid if it were rewritten as follows:

```
on zerodivide begin;
              A = X;
              go to trouble;
              end;
A = B/C;
```

This example is now valid because the <on unit> does not access the value of A; it only accesses the generation of storage of A.

10.4 PL/I Conditions

In the following discussion, a <reference> is understood to be a <reference> to a file value. Refer to Section 11 for a description of the relationship between file values, file-state blocks, and data sets.

In the following discussion, error_output is understood to be the Multics error_output I/O switch.

Although the description of each condition states when the condition occurs, the following conditions may occur anytime during execution of the program:

underflow
overflow
fixedoverflow
zerodivide
size
stringsize
storage
area
error

These conditions occur when the compiled code or any of its supporting subroutines exceed one or more of their limitations or when they detect an error. Execution of a valid program does not normally cause these unexpected conditions to occur.

10.4.1 Area Condition

Syntax:

<area condition name>::= area

This condition occurs when an <allocate statement> attempts to allocate a generation of a based variable in an area whose size is insufficient to contain the generation, or when an <assignment statement> assigns an area to an area whose size is insufficient to contain the assigned area. If an <on unit> returns to the point where the condition was detected and the condition was signalled by the execution of an <assignment statement>, the program is in error. If the condition was signalled by the execution of an <allocate statement>, the allocation is retried including reevaluation of the <in option> of the <allocation>. Unless the <on unit> has freed sufficient storage in the area or caused the value of the <in option> to change to an area that has sufficient storage, the condition will occur again.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.2 Conversion Condition

Syntax:

<conversion condition name>::= conversion|conv

This condition occurs when an invalid character-string or character-pictured value is converted to an arithmetic or bit-string value. Refer to Section 8 for a discussion of character-string conversion.

Just before the condition is signalled, the current values of the onsource and onchar built-in functions are pushed down and the value being converted is assigned to "onsource". The leftmost character for which the conversion failed is assigned to "onchar". If the conversion is being performed by stream input/output, the current value of the onfile built-in function is also pushed down and the current file name is assigned to "onfile". Refer to Section 11 and paragraph 13.5.

If an <on unit> returns to the point where the condition was detected, the conversion is retried using the value of the current generation of "onsource". Unless the <on unit> has assigned a new value to the "onsource" or "onchar" pseudo-variable, the condition will occur again.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.3 Endfile Condition

Syntax:

<endfile condition-name>::= endfile(<reference>)

This condition occurs when a <get statement> or <read statement> attempts to read past the end of the data set attached to the file-state block identified by the file value of the <reference>.

Just before the condition is signalled, the current value of the onfile built-in function is pushed down and the current file name is assigned to "onfile". If the file-state block identified by the file value of the <reference> has the <keyed attribute>, the current value of the onkey built-in function is also pushed down and the current key value is assigned to "onkey". Refer to Section 11 and paragraph 13.5.

Repeated attempts to read past the end of the data set cause the condition to be signalled for each attempt. If an <on unit> returns to the point where the condition was detected, control returns to the <statement> following the <get statement> or <read statement>.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.4 Endpage Condition

Syntax:

<endpage condition name>::= endpage(<reference>)

Let linenumber and pagesize be control values of the file-state block identified by the value of the <reference>.

This condition occurs when a <put statement> places a linemark into the data stream and the newly updated linenumber equals the pagesize+1.

If the condition occurred as the result of an attempt to write data, then on return from the <on unit>, the data is written. If the condition occurred because of the evaluation of a <line option>, <line format>, <skip option>, or <skip format>, then on return from the <on unit>, the format or option is ignored.

Just before the condition is signalled, the current value of the onfile built-in function is pushed down and the current file name is assigned to "onfile". Refer to Section 11 and paragraph 13.5.4.

When endpage is signalled, the linenumber is one greater than the pagesize. During the execution of the <on unit> or after return from the <on unit> without a <page option> or <page format> having been evaluated, the linenumber may increase indefinitely. However, evaluation of a <line option> or a <line format> that would have caused the endpage condition does not cause the condition, but instead writes a pagemark into the output stream.

The default <on unit> places a pagemark into the data stream, resets the linenumber to 1, and returns to the point where the condition was detected.

10.4.5 Error Condition

Syntax:

<error condition name>::= error

This condition is signalled by the default <on unit>s for several conditions. It is also signalled by the mathematical built-in functions as described in paragraph 13.3 and by the exponentiate operator as described in Section 7.

If an <on unit> attempts to return to the point where the condition was signalled, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and returns to the Multics command processor. If the start command is typed on the console, then control returns to the point where the condition was signalled, but the program is in error and the effects of continued execution are undefined.

10.4.6 Finish Condition

Syntax:

<finish condition name>:: = finish

This condition occurs when the process or run unit has attempted to terminate.

The default on-unit returns to the point where the condition was detected.

If process or run unit termination results from partial destruction of the process or run unit, or exhaustion of process resources, the signal may or may not occur and the correct execution of the <on-unit> may or may not occur.

10.4.7 Fixedoverflow Condition

Syntax:

<fixedoverflow condition name>::= fixedoverflow|fofl

This condition occurs when the result of a binary fixed-point computation exceeds 71 binary digits. If an <on unit> returns to the point where the condition was detected, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.8 Key Condition

Syntax:

<key condition name>::= key(<reference>)

This condition occurs when a <key option> specifies a key value that does not identify any record in the data set attached to the file-state block identified by the by the file value of the <reference>. It also occurs when a <keyfrom option> specifies a key value that identifies a record that already exits in the data set.

Just before the condition is signalled, the currentrecord and nextrecord values of the file-state block are set to undefined values, and the current values of the "onfile" and "onkey" built-in functions are pushed down and the current file name is assigned to "onfile" and the current key value is assigned to "onkey". Refer to Section 11 and paragraph 13.5.

If an <on unit> returns to the point where the condition was detected, control returns to the <statement> following the <statement> that caused the condition to occur.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.9 Name Condition

Syntax:

<name condition name>::= name(<reference>)

This condition occurs when a stream data set is being processed by a <get statement> containing a <get data>. It occurs if the stream contains a <basic reference> that does not identify a variable whose scope of declaration includes the <get statement>, or if the stream contains a <basic reference> that identifies a variable that is not identified by a <get data ref> specified by the <get data>.

Just before the condition is signalled, the current value of the "onfield" and "onfile" built-in functions are pushed down and the current file name is assigned to "onfile". The character-string extracted from the data stream by the <get statement> is assigned to the "onfield" built-in function. Refer to paragraphs 12.14 and 13.5.

If an <on unit> returns to the point where the condition was detected, processing continues with the next input field in the stream.

The default <on unit> writes a comment on error_output and returns to the point where the condition was detected.

10.4.10 Overflow Condition

Syntax:

<overflow condition name>::= overflow|ofl

This condition occurs when the result of a floating-point computation has an exponent that exceeds 127. If an <on unit> returns to the point where the condition was detected, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.11 Record Condition

Syntax:

<record condition name>::= record(<reference>)

This condition occurs when a <read statement> reads a record that is not equal to the size of the variable specified by the <into option>.

Just before the condition is signalled, the current value of the "onfile" built-in function is pushed down and the current file name is assigned to "onfile". If the file-state block identified by the file value of the <reference> has the <keyed attribute> the current value of the "onkey" built-in function is also pushed down and the current key value is assigned to "onkey". Refer to Section 11 and paragraph 13.5.

If an <on unit> returns to the point where the condition was detected, execution continues as described in paragraph 12.23.

The program is in error and the results of continued execution are undefined unless the variable is a valid left part of the record, or the record is a valid left part of the variable. In the former case, excess data in the record is not input. In the later case, only the left part of the variable receives a value. A variable or record is a valid left part of another variable or record if and only if their generations of storage conform to the rules given in paragraph 4.3.3.2 for the sharing of storage by based variables.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.12 Size Condition

Syntax:

<size condition name>::= size

This condition occurs when a value is converted to a fixed-point target value, and the target's precision and scale factor does not provide sufficient digits to the left of the decimal or binary point to represent the integral digits of the converted value.

The size condition also occurs when the result of a decimal fixed-point computation exceeds 59 decimal digits.

The condition also occurs during format controlled output conversion when the output field described by a <fixed-point format> or a <floating-point format> is insufficient to hold the converted value. Refer to paragraphs 8.2.11.1.2 and 8.2.11.2.2.

The condition also occurs when a negative value is assigned to a target whose declaration contains the <unsigned attribute>.

If an <on unit> returns to the point where this condition was detected, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.13 Storage Condition

Syntax:

<storage condition name>::= storage

This condition occurs when the Multics stack segment is about to overflow, or when the "system storage" used to allocate controlled and based variables is full.

If the Multics stack segment is about to overflow, the Multics "stack" condition is signalled. Its default <on unit> signals the PL/I storage condition. In this case, the <on unit> for the storage condition cannot require more than four pages of stack storage. If the <on unit> returns to the point where the condition was detected, the program is in error and the results of continued execution are undefined. Refer to the Multics Programmers' Manual.

If the condition was signalled because "system storage" was full and the <on unit> returns to the point where the condition was detected, the allocation is retried. Unless the <on unit> has freed sufficient storage in "system storage", the condition will occur again.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.14 Stringrange Condition

Syntax:

<stringrange condition name>::= stringrange|strg

This condition occurs when the substr built-in function or <substr pseudo> specify a substring that is not completely contained in the string value that appears as the first argument of the substr reference. If an <on unit> returns to the point where the condition was detected, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.15 Stringsize Condition

Syntax:

<stringsize condition name>::= stringsize|strz

This condition occurs when a value is converted to a string target value and the target's generation of storage is insufficient to contain the string value. If an <on unit> returns to the point where the condition was detected, the string value is assigned to the target from left-to-right until the target is full and any excess characters or bits are truncated.

The value of the target is undefined at the time the condition is signalled.

The default <on unit> returns to the point where the condition was detected.

10.4.16 Subscriptrange Condition

Syntax:

<subscriptrange condition name>::= subscriptrange|subrg

This condition occurs when the value of a <subscript> exceeds the <bound>s of the dimension to which it applies. If an <on unit> returns to the point where the condition was detected, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.17 Transmit Condition

Syntax:

<transmit condition name>::= transmit(<reference>)

This condition occurs when data cannot be reliably transmitted between the data set attached to the file-state block identified by the file value of the <reference> and one or more of the values specified in a <get statement>, <put statement>, <read statement>, <write statement>, <rewrite statement> or <locate statement>.

The value of any datum whose transmission caused the condition is undefined.

Just before the condition is signalled, the current value of the "onfile" built-in function is pushed down and the current file name is assigned to "onfile". If the file-state block identified by the file value of the <reference> has the <keyed attribute> the current value of the "onkey" built-in function is also pushed down and the current key value is assigned to "onkey". Refer to Section 11 and paragraph 13.5.

If an <on unit> returns to the point where the condition was detected, control returns to the <statement> following the <statement> that caused the condition to occur.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.18 Undefinedfile Condition

Syntax:

<undefinedfile condition name>::= {undefinedfile|undf}
(<reference>)

This condition occurs when an attempt to open a file-state block is unsuccessful. If an <on unit> returns to the point where the condition was detected and the <statement> being executed is an <open statement>, execution continues with the evaluation of the next <opening> of the <open statement>. If an <on unit> returns control to the point where the condition was detected and the <statement> being executed is not an <open statement>, the file must have been opened by the <on unit>. If it is not yet open, the error condition is signalled. Just before the condition is signalled, the current value of the "onfile" built-in function is pushed down and the current file name is assigned to "onfile".

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.19 Underflow Condition

Syntax:

```
<underflow condition name>::= underflow|ufl
```

This condition occurs when the result of a floating-point computation has an exponent less than -128. The result of the computation is set to zero before the condition is signalled. If an <on unit> returns to the point where the condition was detected, execution continues using a value of zero.

The default <on unit> writes a comment on error_output and returns to the point where the condition was detected.

10.4.20 Zerodivide Condition

Syntax:

```
<zerodivide condition name>::= zerodivide|zdiv
```

This condition occurs when the divisor of a fixed-point or floating-point computation is zero. If an <on unit> returns to the point where the condition was detected, the program is in error and the results of continued execution are undefined.

The default <on unit> writes a comment on error_output and signals the error condition.

10.4.21 Multics and Programmer Defined Conditions

Syntax:

```
<programmer defined condition name>::=
     {condition|cond}(<identifier>)|<identifier>
```

Any <identifier> not used to designate a PL/I condition and not otherwise declared in the <block>, except as the name of a structure member, can be declared as a condition and used to designate a programmer defined condition. Programmer defined conditions behave just like other PL/I conditions, except that the only way they are signalled is by the execution of a <signal statement>. Refer to Section 5 for a discussion of declarations and the <condition attribute>.

The Multics operating system defines a number of conditions that are not included in the set of PL/I conditions. These conditions are considered to be programmer defined conditions and behave as such, except that they may be

signalled by the Multics operating system or by the execution of a <signal statement>. Refer to the "Multics Programmers' Manual".

The default <on unit> writes a comment on error_output and calls the Multics command processor. If the "start" command is typed control returns to the point where the condition was signalled.

SECTION 11

INPUT/OUTPUT

11.1 Data Sets

A data set is either a stream data set or a record data set.

11.1.1 Stream Data Sets

A stream data set is an ordered sequence of data characters and control characters. A control character is a pagemark or linemark. A data character is any ASCII character, other than those used as control characters. A linemark is an ASCII new line character. A pagemark is an ASCII new page character. The effects of these control characters on the I/O mechanism are discussed in paragraph 11.2.

Stream data sets are operated upon be the execution of <get statement>s and <put statement>s as described in section 12.

11.1.2 Record Data Sets

A record data set is a set of discrete records each of which is the internal representation of a PL/I value. The Multics system supports two kinds of record data sets: sequential data sets and indexed sequential data sets. A sequential data set is an ordered sequence of records without keys. An indexed sequential data set is an ordered sequence of records each identified by a unique key. A key is a character-string whose maximum length is 256 characters.

Records of a sequential data set are in chronological order; that is, the order in which they were written. Records of an indexed sequential data set are in key order, that is record x precedes record y if and only if the key of x is less than the key of y.

Record data sets are operated upon by the execution of: <read statement>s, <write statement>s, <rewrite statement>s, <delete statement>s and <locate statement>s as described in Section 12.

11.2 File Values and File-state Blocks

A file value identifies a file-state block. A file constant always identifies the same file-state block, but a file variable can identify any file-state block. A file-state block, sometimes called a file, is a composite value that describes the relationship between the program and a data set.

A program has as many file-state blocks as it has file constants. The file description attributes declared for a file constant are really properties of the file-state block.

A file-state block consists of:

1. Name of the file-state block. (filename)
2. A data set designator. (title)
3. Current record designator. (currentrecord)
4. Next record designator. (nextrecord)
5. Stream position designator. (streamposition)
6. Initial file description attributes. (initialdescription)
7. Current file description attributes. (filedescription)
8. Input buffer.
9. Output buffer.
10. Open/closed status.
11. Current line size. (linesize)
12. Current page size. (pagesize)
13. Current column position. (columnposition)
14. Current line number. (linenumber)
15. Current page number. (pagenumber)
16. External/internal status.

The parenthesized names are used throughout this document to denote the components of a file-state block. The meaning of each value is discussed below:

The filename is a varying character-string of maximum length 32 that is the name of the file-state block. The <declared name> of the file constant that identifies the file-state block is the filename.

Example:

```
declare f file constant;
```

The file-state block identified by the file value of f has a filename of "f".

The title is a character-string that is used as a Multics I/O attach description as described in paragraph 11.3.

The currentrecord is either null or it designates a record in the data set designated by title.

The nextrecord is either null or it designates the next record in the sequential data set designated by title. Each time that the value of currentrecord is changed, the value of nextrecord is updated. When the currentrecord has been set null by the execution of a <delete statement> as described in paragraph 12.8, the nextrecord is used to find the next record of a sequential data set.

If the <get statement> or <put statement> contains a <string option> instead of a <file option>, data stream is understood to be the string value identified by the <string option>; otherwise, it is understood to be the data set associated with the file.

The streamposition is either null or it designates the current character in a stream data set.

The initialdescription is the set of file description attributes declared for the file constant whose value identifies this file-state block.

Example:

```
declare f file stream input;
```

In this example, the initial file description attributes are "stream" and "input".

The filedescription is the set of attributes that describe the data set. It is formed when the file-state block is opened as described in paragraph 11.3.

The input buffer is used to support the execution of <read statement>s containing a <set option> and is described in paragraph 12.23.

The output buffer is used to support the execution of the <locate statement> and is described in paragraph 12.17.

The open/closed status indicates whether the file-state block is open or closed. Initially it is closed.

The linesize is the maximum number of data characters that may be written between linemarks in a data stream attached to a file-state block that has the <stream attribute> and <output attribute>. During stream output a linemark is output whenever a character is to be output and columnposition = linesize+1. Linemarks are also output by evaluation of <skip option>s, <skip format>s, <line option>s and <line format>s. They can also be output as a result of evaluating a <column format>. Refer to Section 12.

The pagesize is the maximum number of linemarks that may be written between pagemarks in a data stream attached to a file-state block that has the <print attribute> without causing the endpage condition to occur. During output on a file that has the <print attribute>, the endpage condition is signalled whenever linenumber = pagesize + 1. The default <on unit> for the endpage condition writes a pagemark.

The columnposition is the number of data characters input or output since the last linemark plus one. In other words, it is the column into which the next character will be written, or the column from which the next character will be read. The input or output of a linemark sets the columnposition to 1.

The linenumber is a count of the number of linemarks output since the last pagemark plus one. The output of a pagemark sets the linenumber to 1.

The pagenumber is a count of the pagemarks output since the file was opened plus one. The initial value of pagenumber is 1. Note that pagenumber can be set by the <pageno pseudo> described in paragraph 12.2.

The external/internal status indicates the scope of the file constant whose value identifies this file-state block.

11.3 Opening a File

A file is opened by performing the following steps in the indicated order:

1. Form the filedescription by forming the union of the initialdescription and the <opening attribute>s supplied by the <statement> performing the opening. The following table gives the <opening attribute>s for all input/output <statement>s capable of opening a file.

Statement	Opening Attributes
<get statement>	stream input
<put statement>	stream output
<read statement>	record input*
<write statement>	record output*
<rewrite statement>	record update
<locate statement>	record output
<delete statement>	record update
<open statement>	<opening attributes> given in <statement>

   * If the initialdescription specifies an <update attribute>, the <input attribute> or <output attribute> is not one of the <opening attribute>s deduced from a <read statement> or <write statement>.

2. Augment the filedescription with the <attribute>s implied by any <attribute> already in the description.

Attribute	Implied Attribute
direct	record keyed
keyed	record
print	stream output
sequential	record
update	record

3. If, after supplying implied <attribute>s, the filedescription is missing one of the following required <attribute>s, the required <attribute> is supplied by default.

Required Attributes	Default
stream|record	stream
input|output|update	input
sequential|direct	sequential (if record)

4. If the filename is "sysprint" and the file-state block is external and the filedescription contains the <stream attribute> and the <output attribute>, augment the filedescription with the <print attribute>.

5. The filedescription must now be a set of <attribute>s described by the following syntax:

```
<consistent file description>::= <stream description>|
     <record description>

<stream description>::= stream{input|output[print]
     [environment(interactive)]}

<record description>::= record{input|output|update}
     {<sequential description>|<direct description>}
     [environment(stringvalue)]

<sequential description>::= sequential[keyed]

<direct description>::= direct keyed
```

6. If the filedescription contains the <print attribute> and the opening is being performed by the execution of an <open statement> and the <opening> contains a <pagesize option>, pagesize is set to the converted value of the <pagesize option>; otherwise, it is set to a default value that depends on the device or data set to which the stream is attached. If the stream is attached to a terminal, pagesize is set to infinity, thereby preventing an endpage condition from occurring; otherwise it is set to 60.

   If the opening is being performed by the execution of an <open statement> and the <opening> contains a <pagesize option>, the filedescription must contain the <print attribute>.

7. If the filedescription contains a <stream attribute> and an <output attribute>, and the opening is being performed by the execution of an <open statement> and the <opening> contains a <linesize option>, linesize is set to the converted value of the <linesize option>; otherwise, it is set to a default value that depends on the Multics I/O System attachment. If the Multics I/O switch is attached to a terminal, linesize is set to the current linesize of the terminal; otherwise, it is set to 132.

   If the opening is being performed by the execution of an <open statement> and the <opening> contains a <linesize option>, the filedescription must contain the <stream attribute> and the <output attribute>.

8. If the opening is being performed by the execution of an <open statement> and the <opening> contains a <title option>, let t be the converted value of the <title option>; otherwise, t is defined by the following:

   If the filename is "sysin" and the filedescription contains the <stream attribute> and <input attribute>, let t be "syn_ user_input".

If the filename is "sysprint" and the filedescription contains the <stream attribute> and <output attribute>, let t be "syn_ user_output".

If neither of these two cases applies, let t be "vfile_ filename".

The character-string t is passed as an attach description to the Multics I/O system. Refer to the Multics PL/I Reference Manual, for a complete description of the relationship between the Multics PL/I language and the Multics I/O system.

9. If the filedescription contains an <output attribute>, any existing data set designated by the title is normally deleted and a new data set conforming to the file description is created.

   If the filedescription contains an <input attribute> or <update attribute>, the data set designated by the title is checked for conformance with the filedescription. The following table shows all valid filedescriptions for each type of data set.

Data Set	File Description
stream	stream input
sequential	sequential record{input|update}| stream input
indexed sequential	sequential record{input|update}[keyed]| record direct keyed{input|update}

10. If the filedescription contains a <stream attribute>, the columnposition is set to one. If it also contains the <print attribute>, the linenumber and pagenumber are set to one. If it contains the <stream attribute> and the <input attribute>, then the streamposition is set to the first character in the data stream.

    If the filedescription contains the <record attribute> and does not contain the <output attribute>, nextrecord is set to designate the first record in the data set.

    In all other cases, the values of currentrecord, nextrecord or streamposition are null.

11. If steps 1 through 10 were successfully performed, the open/closed status is set "open"; otherwise, the undefinedfile condition is signalled.

11.4 Closing a File

A file is closed by performing the following steps in the indicated order:

1. If there is an output buffer, a new record is created in the data set and the content of the buffer is written as the value of the new record. If there is an evaluated key associated with the buffer, it is associated with the new record as its key. If any record in the data set already has this key, the key condition is signalled.

   If the file does not have the <keyed attribute>, the new record is appended to the end of the data set; otherwise, the new record is inserted into its proper place within the data set as determined by its key. After the record is written, the output buffer is freed. An output buffer exists when the last output operation on the file was the execution of a <locate statement>.

2. If there is an input buffer, it is freed. This circumstance occurs when the last input operation on the file was the execution of a <read statement> containing a <set option>.

3. If the data set was attached by the execution of a PL/I input/output <statement>, the Multics I/O system is called to detach the data set.

4. The open/closed status is set "closed".

Files are closed by the execution of a <close statement>, upon normal termination of a process or run unit, or by the Multics command: close_file.

If process termination results from partial destruction of the process or exhaustion of process resources, the files open in that process, and in contained run units, may or may not be closed. The same applies to run unit termination.

Note that if a process terminates without closing a file, the contents and state of the data set designated by that file are undefined.

11.5 Conditions and Files

Several of the conditions described in Section 10 are detected during the execution of input/output <statement>s. Each I/O condition name contains a <reference> to a scalar file value. The file value identifies a file-state block and effectively qualifies the condition name. An endfile condition for file f is a different condition from an endfile condition for file g. Refer to Section 10 for a full discussion of conditions.

Example:

```
on endfile(f) begin; ... end;

on endfile(g) begin; ... end;
```

Execution of the <statement>s in this example establishes two <on unit>s, one for endfile on file f, the other for endfile on file g.

It is important to realize that the I/O conditions are qualified by the file's state block, not by the <reference> used in the condition name.

Example:

```
declare f file variable, g file constant;

f = g;

on endfile(f) begin; ... end;

on endfile(g) begin; ... end;
```

Execution of the second <on statement> in this example reverts the <on unit> established by the execution of the first <on statement> because f and g both identify the same file-state block.

SECTION 12

SYNTAX AND SEMANTICS OF STATEMENTS

Throughout this section, whenever the execution of a <statement> is described, the evaluation of its options and parts as if the option or part were present is described. Such descriptions are not to be taken as an indication that the described option or <statement> part is required; only the syntax and constraints indicate whether or not an option is required.

12.1 The Allocate Statement

Syntax:

```
<allocate statement>::= [<prefix>]{allocate|alloc}
     <allocation>[,<allocation>]...;

<allocation>::= <allocation reference>
     {[<in option>][<set option>]|
     [<set option>][<in option>]}

<in option>::= in(<reference>)

<set option>::= set(<reference>)

<allocation reference>::= <identifier>
```

Constraints:

Each <allocation reference> must identify a level-one based or controlled variable.

Evaluation of the <reference> in a <set option> must yield a generation of storage of a scalar locator variable.

Evaluation of a <reference> in an <in option> must yield a generation of storage of a scalar area variable.

If the <allocation reference> of an <allocation> identifies a controlled variable, the <in option> and <set option> must be omitted.

If the <allocation reference> of an <allocation> identifies a based variable and the <set option> is omitted, the based variable must be declared with a <based attribute> containing a <locator qualifier>. In that case, the <locator qualifier> is used as a <set option> and it must satisfy the constraints specified for the <set option>.

If the <set option> or derived <set option> of an <allocation> identifies an offset variable, the <in option> must be present or the offset variable must have been declared with an <offset attribute> containing a <reference>. In the latter case, the <reference> is used as the <in option>, and it must satisfy the constraints specified for the <in option>.

If the <set option> or derived <set option> identifies a pointer variable and the <in option> is omitted, a default area called "system storage" is supplied as the <in option>.

Semantics:

An <allocate statement> is executed by evaluating its <allocation>s from left-to-right. Each evaluation consists of performing one of the following:

1. If the <allocation reference> identifies a controlled variable, a new generation of the controlled variable is allocated in "system storage". The newly allocated generation and its evaluated extents are stacked on the previous generation and its extents. References to the variable reference the newly allocated generation and references to the extents of the variable reference the extents of the newly allocated generation. The <initial attribute>s of the controlled variable's declaration are evaluated in an unspecified order and initial values are assigned to the newly allocated generation.

2. If the <allocation reference> identifies a based variable, a new generation of the based variable is allocated in the area identified by the <in option> and a locator value that identifies the generation is assigned to the variable identified by the <set option>. The <initial attribute>s of the based variable's declaration are evaluated in an unspecified order and initial values are assigned to the newly allocated generation. Before initialization, the value of the <expression> of each <refer option> is assigned to the variable identified by the <reference> of the <refer option>.

If insufficient storage exists within the area identified by the <in option>, the area condition occurs. If insufficient storage exists within "system storage", the storage condition occurs. Refer to paragraph 10.4 for a discussion of the area and storage conditions. Refer to paragraph 4.3.2 for a discussion of storage classes and storage allocation.

Examples:

```
allocate X set(P);

allocate X,Y,Z;

allocate X in(A) set(P),Y in(B) set(Q);
```

12.2 The Assignment Statement

Syntax:

```
<assignment statement>::= [<prefix>]<target>[,<target>]...
     =<expression>[,<by-name option>];

<target>::= <reference>|<pseudo-variable>

<pseudo-variable>::= <string pseudo>|<substr pseudo>|
     <unspec pseudo>|<pageno pseudo>|<real pseudo>|
     <imag pseudo>|<onchar pseudo>|<onsource pseudo>

<string pseudo>::= string(<reference>)

<substr pseudo>::= substr(<reference>,
     <expression>[,<expression>])

<unspec pseudo>::= unspec(<reference>)

<pageno pseudo>::= pageno(<reference>)
```

<real pseudo>::= real(<reference>)

<imag pseudo>::= imag(<reference>)

<onchar pseudo>::= onchar[()]

<onsource pseudo>::= onsource[()]

<by-name option>::= by name

Constraints:

If the <expression> is a <reference> to a scalar character-string or bit-string variable, the generation of storage identified by that <reference> cannot overlap the generation of storage identified by any of the <target>s if the generation of storage identified by the <target> starts to the right of the start of the generation of storage identified by the <expression>. Refer to paragraph 4.3.3 for a discussion of storage sharing.

Because of compiler optimizations, if the <expression> is a <reference> to the string, substr, or unspec built-in functions, and the first argument of the <reference> is a <reference> to a variable, the <expression> is considered to be a <reference> to a generation of storage, and the constraint in the previous paragraph applies.

Evaluation of the <expression> cannot: allocate, free, or assign a value to any <target>, or any generation of storage identified by any <reference> contained within any <target>; i.e. <subscript>s, etc.

Evaluation of the <reference> in a <string pseudo> must yield a generation of storage of a scalar or aggregate string variable suitable for string-overlay defining as described in paragraph 4.3.3.6.

Evaluation of the <reference> in a <substr pseudo> must yield a generation of storage of a scalar or aggregate string variable. If any scalar component of the variable was declared with the <varying attribute>, that component must currently have a value.

Evaluation of the <expression>s in a <substr pseudo> must yield scalar or aggregate arithmetic or string values. If either <expression> yields an aggregate value, its aggregate type must not be higher than the aggregate type of the <reference> of the <substr pseudo>.

Evaluation of the <reference> in an <unspec pseudo> must yield a generation of storage of a scalar or aggregate variable whose storage is connected.

Evaluation of the <reference> in a <pageno pseudo> must yield a scalar file value.

Evaluation of the <reference> in a <real pseudo> or <imag pseudo> must yield a generation of storage of a scalar or aggregate complex variable.

If evaluation of the <expression> yields an area value, it must be a scalar area value and each <target> must be a <reference> to a scalar area variable.

If the <assignment statement> contains a <by-name option>, it may contain no <function reference>s yielding non-scalar values unless they are contained inside a <locator qualifier>, a <subscript>, or an <argument list>. This prohibits the use of structure- or array-valued procedure functions and built-in functions when the <by-name option> is specified.

This page intentionally left blank.

If the <assignment statement> contains a <by-name option>, it may not contain any <pseudo-variable>s, and all <target>s must be <reference>s to structures or arrays of structures.

Semantics:

If evaluation of the <expression> yields an area, let A denote that area and assign A to each <target> taking them from left-to-right. If the size of a target area is insufficient to contain A, the area condition occurs. A target area must be large enough to contain all generations currently allocated in A and must be large enough so that each generation can be allocated in the target area with the same offset as it had in A.

If the <assignment statement> contains a <by-name option>, the following steps are performed:

1. For each <target> in the <assignment statement> and for each <reference> to a structure contained in the <expression> but not contained within a <locator qualifier>, <subscript>, or <argument list>, create a list of names of all non-structure members of the referenced structure or contained substructures. Each member's name includes the names of its contained structures up to but not including the name of the structure identified by the structure <reference>.

2. Form the intersection of the lists created in step 1 and call the resulting list the by-name-parts-list. If the by-name-parts-list is empty, treat the <assignment statement> as a <null statement>. This means that only names that are contained in all <target>s and all appropriate <reference>s in the <expression> will be assigned.

3. Determine the aggregate-type for each <target> and for each <reference> to a structure contained in the <expression> but not contained in a <locator qualifier>, <subscript>, or <argument list> by the following method:

   a. Each <reference> is a structure or array of structures depending on its <declaration> and <subscripts>, if any.

   b. The structure or array of structures mentioned in step a contains one non-structure member for each name in the by-name-parts-list. Each member acquires dimensions from two sources: its own <declaration>, and the <declaration> of every containing structure that is an array.

4. Perform the assignment using the new aggregate-types according to the following rules for assignment.

If evaluation of the <expression> does not yield an area, an <assignment statement> is executed as if it were replaced by a set of simple <assignment statement>s of the form:

```
 V = E;
T1 = V;
T2 = V;
   .
   .
   .
Tn = V;
```

where E is the <expression> and V is a variable whose aggregate type and data type are the aggregate type and data type of the <expression>. T1,T2,...,Tn are the <target>s of the original <assignment statement> taken from left-to-right. If Tj is an arithmetic variable, a bit-string variable, or a pictured variable defined by a <numeric picture>, and E is a pictured value defined by a <numeric picture>, the value of E is encoded to an arithmetic value as described in Section 8. In that case, the data type of V is the data type of the encoded value.

The rewritten <assignment statement> is evaluated by performing the following steps in the indicated order:

1. E is evaluated and its value is assigned to V.

2. For j = 1,2,...,n: the value of V is promoted to a value that conforms to the aggregate type of Tj, the promoted value is converted to conform to the data type of Tj, and the promoted and converted value V' is assigned to Tj. The value of V is unaffected by these promotions and conversions. Refer to Sections 8 and 9 for a discussion of conversions and promotions.

If Tj is an aggregate character-string or bit-string variable, the scalar components of V' are assigned to the corresponding scalar components of Tj using the following rules for scalar string assignment.

If Tj is a scalar pictured character-string variable, V' is edited into Tj as described in paragraph 8.2.12.

If Tj is a scalar, nonpictured, character-string or bit-string variable, the following rules apply to the assignment. Let n be the length of V', and let m be the declared length of Tj.

1. If Tj is varying and n<m, assign V' to Tj and set the current length of Tj to n.

2. If Tj is nonvarying and n<m, extend V' to the right by concatenating m-n blanks (if Tj is character) or zeroes (if Tj is bit) and assign the extended value to Tj.

3. If n>m, truncate the rightmost n-m characters or bits from V' and assign the truncated value to Tj. If Tj is varying set the current length of Tj to m.

4. If n=m, assign V' to Tj and if Tj is varying set the current length of Tj to m.

If the <target> is a <string pseudo>, the promoted and converted value is assigned to the generation of storage identified by the <reference> as if it were the generation of storage of a nonvarying scalar string variable.

When one or more operands of a <pseudo-variable>, other than a <string pseudo>, is an aggregate, the operands are promoted to the highest common aggregate type. The promoted and converted value V' is assigned to the <pseudo-variable> by assigning the corresponding scalar components in an unspecified order as described in the following paragraphs:

If the <target> is a <substr pseudo>, the values of the <expression>s are converted to fixed-point, binary, real, values of precision (24,0). Let i be the converted value of the first <expression> and let j be the converted value of the second <expression>. If the string variable identified by the <reference> is declared with the <varying attribute> let n be the current length of the string variable value; otherwise, let n be the evaluated string length extent associated with the variable's generation of storage.

If the second <expression> is omitted, let j be n-i+1. If ($0 \leq i-1 \leq j+i-1 \leq n$) is not satisfied, the stringrange condition occurs. If detection of the condition is disabled, the program is in error and the results of continued execution are undefined.

If the inequality is satisfied, the promoted and converted value is assigned to the string variable beginning with the ith character or bit and continuing through the (i+j-1)th character or bit if j^=0; otherwise if j=0, all characters or bits of the string are unmodified. All other characters or bits of the string are unmodified.

If the <target> is an <unspec pseudo>, the generation of storage of the <reference> is treated as a scalar bit-string variable and the promoted and converted value V' is assigned to it.

If the <target> is a <pageno pseudo>, the promoted and converted value V' is assigned to the pagenumber value of the file-state block identified by the file value of the <reference> of the <pageno pseudo>. The file-state block must be open and must have the <print attribute>.

If the <target> is a <real pseudo>, the promoted and converted value V' is assigned to the real part of the complex variable identified by the <reference> of the <real pseudo>.

If the <target> is an <imag pseudo>, the promoted and converted value V' is assigned to the imaginary part of the complex variable identified by the <reference> of the <imag pseudo>.

If the <target> is an <onchar pseudo>, the promoted and converted value V' is assigned to the current generation of "onchar", which must not be the initial generation. Refer to paragraph 10 and paragraph 13.5.

If the <target> is an <onsource pseudo>, the promoted and converted value V' is assigned to the current generation of "onsource", which must not be the initial generation. Refer to Section 10 and paragraph 13.5.

Examples:

```
A,B,C = 0;

string(S) = "new value";

substr(X,i+5,K-5),W = Y||Z;

A = B+C;

A = B+C, by name;

D = B+C, by name;
```

Note that the action of the previous two <statement>s is <u>not</u> necessarily the same as that of the following <statement>:

```
A,D = B+C, by name;
```

12.3 The Begin Statement

Syntax:

<begin statement>::= [<prefix>]begin[<options attribute>];

Semantics:

A <begin statement> defines the beginning of a <begin block>. If executed by the flow of control, it causes a block activation of the <begin block>. Refer to Section 3 for a discussion of block activation and to Section 2 for the role of the <begin statement> in determining the structure of a <begin block>.

Because a <label prefix> on a <begin statement> produces a declaration of a label constant rather than a declaration of an entry constant, a <begin block> cannot be invoked by the execution of a <call statement> or evaluation of a <function reference>. The <options attribute> may only specify the keyword "non_quick" (see 5.4.36, Options).

Example:

```
begin;
```

12.4 The Call Statement

Syntax:

```
<call statement>::= [<prefix>]call<entry reference>
     [<argument list>];

<argument list>::= ([<expression>[,<expression>]...])

<entry reference>::= <reference>
```

Constraints:

Evaluation of the <entry reference> must yield a scalar entry value.

The <entry statement> or <procedure statement> identified by the value of the <entry reference> cannot have a <returns attribute>.

The number of <expression>s in the <argument list> must be equal to the number of <identifier>s in the <parameter list> of the <entry statement> or <procedure statement> identified by the value of the <entry reference>.

Semantics:

A <call statement> is executed by evaluating the <entry reference> and all <expression>s in an undefined order. The entry identified by the value of the <entry reference> is invoked and each <expression> in the <argument list> is associated with the corresponding parameter in the <parameter list> of the invoked entry. Refer to paragraph 6.10 for a discussion of argument passing, and refer to Section 3 for a discussion of block activation.

Examples:

```
call alpha(A,B,C);

call beta();

call gamma;
```

12.5 The Close Statement

Syntax:

```
<close statement>::= [<prefix>]close<file option>
     [,<file option>]...;

<file option>::= file(<reference>)
```

Constraint:

Evaluation of each <reference> must yield a scalar file value.

Semantics:

A <close statement> is executed by evaluating its <file option>s in an unspecified order. For each <file option>, let f denote the file-state block identified by the value of the <reference>. If f is closed, take no further action for this <file option>; otherwise, close f as described in paragraph 11.4.

Example:

```
close file(f), file(q);
```

12.6 The Declare Statement

Syntax:

```
<declare statement>::= [<label prefix>]...{declare|dcl}
     <declaration list>;

<declaration list>::= <declaration component>
     [,<declaration component>]...

<declaration component>::= [<level>][{<declared name>|
     (<declaration list>)}[<attribute set>]

<declared name>::= <identifier>

<attribute set>::= <attribute>...

<level>::= <decimal integer>
```

Semantics:

Execution of a <declare statement> causes control to pass to the <statement> following the <declare statement>.

A <declare statement> establishes a declaration for each <declared name> and is fully described in Section 5.

Examples:

```
declare (a bit, b fixed, c pointer) internal static;

declare 1 S, 2 A, 2 B;
```

12.7 The Default Statement

Syntax:

```
<default statement>::= [<label prefix>]...{default|dft}
     {system|none|<user defaults>};

<user defaults>::= (<predicate>){error|<attribute set>[,<attribute set>]...}

<attribute set>::= <attribute>...

<predicate>::= <predicate one>|
     <predicate>|<predicate one>

<predicate one>::= <predicate two>|
     <predicate one>&<predicate two>

<predicate two>::= <predicate three>|^<predicate two>

<predicate three>::= (<predicate>)|<attribute keyword>|
     <range>

<range>::= range(*)|range(<identifier>)|
     range(<letter>:<letter>)

<attribute keyword>::= <identifier>
```

Semantics:

Execution of a <default statement> causes control to pass to the <statement> following the <default statement>.

A <default statement> supplies <attribute>s to declarations with incomplete <attribute set>s, and is fully described in Section 5.

Example:

```
default (variable & range(c)) character(1);
```

12.8 The Delete Statement

Syntax:

```
<delete statement>::= [<prefix>]delete
     {<file option>[<key option>]|
     [<key option>]<file option>};

<file option>::= file(<reference>)

<key option>::= key(<expression>)
```

Constraints:

Evaluation of the <reference> in the <file option> must yield a scalar file value.

Evaluation of the <expression> in a <key option> must yield a scalar string or arithmetic value.

Semantics:

A <delete statement> is executed by performing the following steps in the indicated order:

1. Evaluate the <file option> and the <key option> in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   Convert the value of the <expression> in the <key option> to a character-string.

2. If f is closed, open it as described in paragraph 11.3. After f is opened, it must have the <update attribute>. If a <key option> is specified, f must have the <keyed attribute>.

3. If a <key option> is specified, set the currentrecord of f to designate the record identified by the converted value of the <key option>. If no such record exists in the data set attached to f, signal the key condition.

   If a <key option> is specified and f has the <sequential attribute>, set nextrecord to designate the record following the new current record. If no next record exists, set nextrecord null.

   If no <key option> is specified, currentrecord must not be null.

4. Delete the record designated by currentrecord and set currentrecord null.

Examples:

```
delete file(f) key(k);

delete file(g);
```

12.9 The Do Statement

Syntax:

```
<do statement>::= [<prefix>]{<noniterative do>|
     <iterative do>}

<noniterative do>::= do;

<iterative do>::= {<do while>|<multiple do>};

<do while>::= do while(<while expression>)

<while expression>::= <expression>

<multiple do>::= do<index>=<control>[,<control>]...

<index>::= <reference>|<pseudo-variable>

<control>::= <single loop>|<repeat control>|
     <fortran control>

<single loop>::= <expression>[while(<while expression>)]

<repeat control>::= <first>repeat<thereafter>
     [while(<while expression>)]

<first>::= <expression>

<thereafter>::= <expression>
```

```
<fortran control>::= <start>{to<limit>[by<increment>]|
    by<increment>[to<limit>]}[while(<while expression>)]

<start>::= <expression>

<limit>::= <expression>

<increment>::= <expression>
```

Constraints:

Evaluation of all <expression>s must yield scalar values.

If an <index> is a <reference>, evaluation of the <reference> must yield a generation of storage of a scalar variable whose data type is other than area.

If an <index> is a <pseudo-variable>, it must satisfy the constraints given in paragraph 12.2 for the <target> of an <assignment statement> and it must be a scalar.

The data types of the values of the <single loop>, <first>, <thereafter> and <start> must be such that they can be assigned to the <index>.

If a <fortran control> contains a <limit>, evaluation of the <index> or the <limit> cannot yield complex arithmetic values.

The <index> of a <multiple do> containing one or more <fortran control>s must have a data type such that the <index> and each <increment> can form an <assignment statement> of the form:

```
<index> = <index> + <increment>;
```

Control cannot transfer from a <statement> outside of an <iterative group> to a <statement> that is a <block component> of an <iterative group>. Refer to Section 2 for the syntax of a <group>.

Note that the scope of a <condition prefix> of a <do statement> is limited to the <do statement> and does not include the <group> headed by the <do statement>.

Semantics:

A <do statement> is executed by selecting the applicable case and performing the steps for that case in the indicated order.

Case (The <do statement> is a <noniterative do>)

1. Transfer control to the <statement> following the <do statement>.
2. Whenever control reaches the <end statement> that ends the <group>, transfer control to the <statement> that follows it.

Case (The <do statement> is a <do while>)

1. Evaluate the <while expression> and convert its value to a bit-string. If all bits are 0, transfer control to the <statement> following the <end statement> that ends the <group>; otherwise, transfer control to the <statement> that follows the <do statement>.
2. Whenever control reaches the <end statement> that ends the <group>, goto step 1 of this case.

Case (The <do statement> is a <multiple do>)

1. Evaluate the left-most <control> by selecting the applicable case and performing the steps for that case in the indicated order. When a step calls for the evaluation of the next <control> and no more <control>s remain, transfer control to the <statement> following the <end statement> that ends the <group>. When a step calls for the evaluation of the next <control> and one or more unevaluated <control>s remain, evaluate the left-most unevaluated <control> by selecting the applicable case and performing the steps for that case in the indicated order.

   Case (The <control> is a <single loop>)

   1. Assign the value of the <expression> to the <index> by evaluating an <assignment statement> of the form:

      <index> = <expression>

   2. If the <control> contains a <while expression>, evaluate the <while expression> and convert its value to a bit-string. If all bits are 0, evaluate the next <control>; otherwise, perform the next step of this case.

   3. Transfer control to the <statement> following the <do statement>.

   4. Whenever control reaches the <end statement> that ends the <group>, evaluate the next <control>.

   Case (The <control> is a <repeat control>)

   1. Let V be the actual text of <first>.

   2. Let Index be a variable whose data type is the data type of <index> and whose generation of storage is the generation identified by <index>.

   3. Assign a value to Index by executing an <assignment statement> of the form:

      Index = V;

   4. If the <repeat control> contains a <while expression>, evaluate the <while expression> and convert its value to a bit-string. If all bits are 0, evaluate the next <control>; otherwise, perform the next step of this case.

   5. Transfer control to the <statement> following the <do statement>.

   6. Whenever control reaches the <end statement> that ends the <group>, let V be the actual text of <thereafter>, and go to step 3 of this case.

   Case (The <control> is a <fortran control>)

   1. Perform the next three steps in an undefined order.

   2. If <limit> is given, evaluate it an let the value be L.

   3. If <increment> is given, evaluate it and let the value be I; otherwise, let I be 1.

   4. Let Index be a variable whose data type is the data type of <index> and whose generation of storage is the generation identified by <index>.

5. Assign the <start> to Index by executing an <assignment statement> of the form:

```
Index = <start>;
```

6. If the <fortran control> does not contain a limit, let S be 1; otherwise, let S be 0 if I≥0 and Index>L, or if I<0 and Index<L; otherwise, let S be 1.

7. If S is 0, evaluate the next <control>; otherwise, perform the next step of this case.

8. If the <fortran control> contains a <while expression>, evaluate the <while expression> and convert its value to a bit string. If all bits are 0, let W be 0; otherwise, let W be 1. If the <fortran control> does not contain a <while expression>, let W be 1.

9. If W is 0, evaluate the next <control>; otherwise, transfer control to the <statement> following the <do statement>.

10. Whenever control reaches the <end statement> that ends the <group>, assign the <index> a new value by executing an <assignment statement> of the form:

```
Index = Index + I;
```

11. Go to step 6 of this case.

Examples:

```
do; ... end;

do i = 1 to 10; ... end;

do X = a,b,c,d; ... end;

do X = 1 to -5 by -1 while(a<b); ... end;

do P = Head repeat(P->Next) while(P^=null); ... end;
```

12.10 The End Statement

Syntax:

```
<end statement>::= [<prefix>]end[<closure label>];

<closure label>::= <identifier>
```

Constraint:

The <closure label> must be a <declared name> that appears in a <label prefix> of a preceding <do statement>, <begin statement>, or <procedure statement>.

Semantics:

The effect of a <closure label> is described in paragraph 2.4.

An <end statement> denotes the end of a <group>, <begin block>, or <procedure> as described in Section 2.

When the flow of control executes an <end statement> that denotes the end of a <procedure>, the effect is as if a <return statement> without a <return value> had been executed.

When the flow of control executes an <end statement> that denotes the end of a <begin block>, the current block activation is terminated and the preceding block activation becomes the current block activation. Control is transferred to the <statement> following the <end statement>.

The effect of executing an <end statement> that denotes the end of a <group> depends on the <do statement> that heads the <group>. Refer to paragraph 12.9.

Examples:

```
P: procedure;
     .
     .
     .
     do while(x<y);
     .
     .
     .
     end;     /*end of group*/
     .
     .
     .
     end P;
```

12.11 The Entry Statement

Syntax:

```
<entry statement>::= <label prefix>...entry
     [([<parameter list>])][<entry option>]...;

<entry option>::= <returns attribute>|<reducible attribute>|
     <irreducible attribute>|<options attribute>

<parameter list>::= <identifier>[,<identifier>]...
```

Constraints:

Each <identifier> in the <parameter list> must identify a level-one variable declared in the immediately containing <block>.

If control passes to the <entry statement> by the execution of a <call statement>, the <entry statement> cannot have a <returns attribute>.

If control passes to the <entry statement> by the evaluation of a <function reference>, the <entry statement> must have a <returns attribute>.

The number of <identifier>s in the <parameter list> must equal the number of <expression>s in the <argument list> of the <call statement> or <function reference> that invoked this entry.

An <entry statement> cannot have both a <reducible attribute> and an <irreducible attribute>.

An <entry statement> containing a <returns attribute> must have exactly one <returns attribute> with a <returns descriptor>.

No <label prefix> can contain a <prefix subscript>.

The <options attribute> may only specify the keyword "variable".

Semantics:

An <entry statement> denotes an entry to a <procedure>. When control is transferred to the entry by the evaluation of a <function reference> or the execution of a <call statement>, a new block activation of the <procedure> occurs and the arguments of the <function reference> or <call statement> are associated with the parameters in the <parameter list>. Refer to paragraph 3.3.1 for a discussion of block activation and refer to paragraph 6.10 for a description of arguments and parameters.

If control reaches an <entry statement> as a result of completing the execution of the preceding <statement>, control is transferred to the <statement> following the <entry statement>.

Because the <label prefix> of an <entry statement> results in the declaration of an entry constant rather than a label constant, a <goto statement> cannot transfer control to an <entry statement>.

Example:

```
E: entry(A,B) returns(bit(1));
```

In this example, E is an entry whose invocation results in a bit-string value. The entry requires two arguments which are associated with the parameters A and B.

12.12 The Format Statement

Syntax:

```
<format statement>::= [<condition prefix>]...
     <label prefix>...format(<format specification list>);

<format specification list>::= <format specification>
     [,<format specification>]...

<format specification>::= [<iteration factor>]<format item>|
     <iteration factor>(<format specification list>)

<iteration factor>::= <decimal integer>|(<expression>)

<format item>::= <data format>|<control format>|
     <remote format>

<data format>::= <real format>|<complex format>|
     <bit-string format>|<character-string format>|
     <picture format>

<real format>::= <fixed-point format>|<floating-point format>

<fixed-point format>::= f(<w>[,<d>[,<k>]])

<floating-point format>::= e(<w>[,<d>[,<s>]])

<complex format>::= c(<format part>[,<format part>])

<format part>::= <picture format>|
     <fixed-point format>|<floating-point format>

<picture format>::= p"<picture>"

<bit-string format>::= <radix factor>[(<w>)]

<radix factor>::= {b|b1|b2|b3|b4}
```

<character-string format>::= a[(<w>)]

<w>::= <expression>

<d>::= <expression>

<k>::= <expression>

<s>::= <expression>

<remote format>::= r(<reference>)

<control format>::= <column format>|<x format>|
<page format>|<skip format>|<line format>

<column format>::= {column|col}(<expression>)

<x format>::= x(<expression>)

<page format>::= page

<skip format>::= skip[(<expression>)]

<line format>::= line(<expression>)

Constraints:

Evaluation of all <expression>s must yield scalar arithmetic or string values.

If the <format statement> is controlling the execution of a <get statement>, each <bit-string format> and each <character-string format> must contain a <w>.

Evaluation of the <reference> in a <remote format> must yield a scalar format value.

Each <format specification list> must contain at least one <data format> or <remote format>.

Semantics:

The <format statement> controls the execution of a <get statement> or <put statement> containing a <get edit> or <put edit>. Each time the <get statement> or <put statement> transmits a value to or from the data stream it passes control to the <format specification list>.

The <format item>s of a <format specification list> are evaluated from left-to-right. When control encounters a <format specification> containing an <iteration factor>, the <iteration factor> is evaluated and converted to a fixed-point, binary, real, integer, n. A <format specification> containing an <iteration factor> is used n times. If n < 0, the program is in error. If n = 0, the <format specification> is ignored. If the <format specification> is a parenthesized <format specification list>, the entire list is used n times.

Each time control passes to a <format specification list> all <format item>s between the last used <format item> and the next <data format> are evaluated, then the next <data format> item is evaluated and used to control the conversion of the value being transmitted to or from the data stream.

If control reaches a <remote format>, the <reference> is evaluated to yield a format value. The <format specification list> contained in the <format statement> identified by the format value is invoked as if it were a <procedure>. When control returns from the <format statement>, the next <format item> is evaluated.

If control reaches the end of the outermost <format specification list> of a <get statement> or <put statement> and one or more values remain to be transmitted to or from the data stream, control passes to the beginning of the <format specification list>.

If control reaches the end of the outermost <format specification list> in a <format statement> and one or more values remain to be transmitted to or from the data stream, control returns to the <remote format> that invoked the <format statement>.

If control reaches a <format statement> as a result of normal execution of the preceding <statement>, control is transferred to the <statement> following the <format statement>.

Because the <label prefix> on a <format statement> is declared as a format constant, it is not possible to transfer control to a <format statement> by the execution of a <goto statement>.

Throughout the following discussion of the semantics of <format item>s, file is understood to be the file-state block identified by the file value of the <reference> of the <get statement> or <put statement> being controlled by the <format statement>.

Linesize, pagesize, columnposition, pagenumber and linenumber designate values in the file. Advancing an input data stream or placing characters in an output data stream affect these values as described in Section 11.

If the <get statement> or <put statement> contains a <string option> instead of a <file option>, data stream is understood to be the string value identified by the <string option>; otherwise, it is understood to be the data set identified by the title of the file.

If the <get statement> or <put statement> contains a <string option> it is an error to attempt to evaluate a <page format>, <skip format>, or <line format>.

A <column format> is evaluated by evaluating its <expression> and converting the value of the <expression> to a fixed-point, binary, real, integer, K; K must be greater than zero. The following cases describe the effects of the <column format> on the data stream:

If K < columnposition and the file has the <input attribute>, the data stream is advanced to the next linemark. It is then advanced K-1 characters and columnposition is set to K. If a linemark is encountered before the Kth character, the stream is positioned to the character following this linemark and columnposition is set to one.

If K < columnposition and the file has the <output attribute>, a linemark and K-1 blanks are placed into the data stream and columnposition is set to K.

If K $\geq$ columnposition and the file has the <input attribute>, K-columnposition characters are ignored and the columnposition is set to K. If a linemark is encountered before the Kth character, the stream is positioned to the character following the linemark and columnposition is set to one.

If K $\geq$ columnposition and K > linesize and the file has the <output attribute>, a linemark is placed into the data stream and columnposition is set to one.

If K $\geq$ columnposition and K $\leq$ linesize and the file has the <output attribute>, K-columnposition blanks are placed into the data stream and columnposition is set to K.

An <x format> is evaluated by evaluating its <expression> and converting the value of the <expression> to a fixed-point, binary, real, integer, K; K must be greater than or equal to zero. If the data stream is being input, the next K characters of the stream are ignored. If the end of the data stream is encountered on the first character, signal the endfile condition. If the end of the data stream is encountered after the first character, signal the error condition. If the data stream is being output, K blanks are placed into the stream. The effect of linemarks in the input or linesize on the output is that described for <data format>s in this section.

A <page format> is evaluated by placing a pagemark into the data stream, setting the linenumber and columnposition to one, and adding one to the pagenumber. The program is in error if the file does not contain the <print attribute>.

A <skip format> is evaluated by evaluating its <expression> and converting the value of the <expression> to a fixed-point, binary, real, integer, K. If the <expression> is omitted, let K be 1. If the file does not have the <print attribute>, K must be greater than zero; otherwise, it must be nonnegative.

If the file contains the <input attribute>, advance the data stream until K linemarks have been encountered. The stream is positioned to the character following the Kth linemark and the columnposition set to one. If the end of the data stream is encountered during the scan, signal the endfile condition.

If the file has the <output attribute> and linenumber>pagesize, place K linemarks into the data stream.

If the file has the <output attribute> and pagesize≥linenumber, place min(K,pagesize+1-linenumber) linemarks into the data stream. If K≥pagesize+1-linenumber, signal the endpage condition. In all cases of output, each linemark sets columnposition to one and adds one to linenumber.

If K=0, an ASCII carriage-return character is placed into the data stream and columnposition is set to one. The effect is to reposition the output stream back to the beginning of the current line such that additional output will overprint, but not replace, data already on the line. Note that overprinting will only occur if the device on which the stream is printed obeys the carriage-return control character.

A <line format> is evaluated by evaluating its <expression> and converting the value of the <expression> to a fixed-point, binary, real, integer, K. If K ≤ 0, the program is in error. If K = linenumber and columnposition = 1, nothing is placed in the data stream and no file control values are changed. If K > pagesize or K ≤ linenumber, pagesize-linenumber+1linemarks are written and the endpage condition is signalled, unless linenumber ≥ pagesize+1. In the latter case, a pagemark is written into the data stream. If K > linenumber and K ≤ pagesize, K-linenumber linemarks are placed into the data stream. The linenumber is set to K and the columnposition is set to one. The program is in error if the file does not contain the <print attribute>.

A <data format> is evaluated by evaluating its <expression>s and converting their values to fixed-point, binary, real, values of precision (17,0). If the data stream is being input, let w be the converted value of <w> or the number of characters described by the <picture format>. The next w characters of the data stream are converted according to the rules for format controlled input conversion given in paragraph 8.2.11. If the end of the data stream is encountered on the first character of the w characters, signal the endfile condition. If the end of the data stream is encountered after the first character, signal the error condition. The converted value is then assigned to the current target of the <get edit>.

If the data stream is being output, the current output value is evaluated and converted according to the rules for format controlled output conversion given in paragraph 8.2.11. The converted value is then placed into the data stream.

If during input, the field of characters to be converted contains one or more linemarks, the linemarks are ignored, except that each linemark causes the columnposition to be set to one.

If during output, the converted value does not fit onto the current line, let w be the number of characters to be output and let n be linesize-columnposition+1. The leftmost n characters are placed into the data stream, followed by a linemark. Let m be the number of characters remaining to be output, (w-n). If m ≤ linesize, the remaining characters are placed into the data stream and columnposition is set to m+1. If m > linesize, let n be linesize. n characters are placed into the data stream followed by a linemark, m is set to m-n and the process is repeated until m ≤ linesize. The remaining characters are output as described above for m ≤ linesize.

Examples:

```
get edit(a,b,c)(r(F));

F: format(e(14,2),F(10),a(5));

put edit(x,y,z)(r(F1));

F1: format(page,a,skip(3),2 e(14,1));
```

12.13 The Free Statement

Syntax:

```
<free statement>::= [<prefix>]free<freeing>[,<freeing>]...;

<freeing>::= <free reference>[<in option>]

<in option>::= in(<reference>)

<free reference>::= <reference>
```

Constraints:

Evaluation of a <reference> in an <in option> must yield a generation of storage of a scalar area variable.

Evaluation of the <free reference> must yield a level-one generation of storage of a based or controlled variable.

The <in option> must be omitted if the <free reference> identifies a controlled variable or if the generation of storage yielded by evaluation of the <free reference> is allocated in "system storage".

Standard PL/I requires that the <in option> be present if the generation of storage is not allocated in "system storage", but Multics PL/I does not require the <in option> in this case.

If the <in option> is present, the generation of storage yielded by evaluation of the <free reference> must be allocated in the area identified by the <in option>.

Semantics:

The <free reference>s and <in option>s of all <freeing>s are evaluated from left-to-right.

Freeing a generation of storage makes any values represented in it undefined. A program that attempts to access a freed generation is in error.

Freeing a generation of a controlled variable makes the previously allocated generation the current generation. Refer to Section 4 for a discussion of storage classes and allocation.

Examples:

```
free x;

free y in(A);
```

12.14 The Get Statement

Syntax:

```
<get statement>::= [<prefix>]get[<file get>|<string get>];

<file get>::= <file get option>...

<file get option>::= <file option>|<copy option>|
     <skip option>|<get list specification>

<file option>::= file(<reference>)

<copy option>::= copy[(<reference>)]

<skip option>::= skip[(<expression>)]

<get list specification>::= <get list>|<get data>|<get edit>

<get list>::= list(<get item>[,<get item>]...)

<get item>::= <target>|(<get item>[,<get item>]...<list do>)

<list do>::= <multiple do>

<target>::= <reference>|<pseudo-variable>

<get data>::= data[(<get data ref>[,<get data ref>]...)]

<get data ref>::= <simple reference>|
     <structure qualified reference>

<get edit>::= edit<get edit pair>...

<get edit pair>::= (<get item>[,<get item>]...)
     (<format specification list>)

<string get>::= <string get option>...

<string get option>::= <string option>|<copy option>|
     <get list specification>

<string option>::= string(<expression>)
```

Constraints:

Evaluation of a <get item> or <get data ref> must yield a generation of storage of an arithmetic or string variable.

Evaluation of the <reference> in the <file option> or <copy option> must yield a scalar file value.

Evaluation of the <string option> and <skip option> must yield scalar arithmetic or string values.

A <file get> must contain either a <skip option> or a <get list specification>, or both.

A <file get> cannot contain more than one <file option>, one <copy option>, one <skip option> and one <get list specification>.

A <string get> must have exactly one <string option> and exactly one <get list specification> and may have one <copy option>.

A <structure qualified reference> in a <get data ref> cannot contain any <subscript>s.

A <get data ref> cannot identify a based variable, unless the variable was declared with a <based attribute> that contained a <locator qualifier>.

A <get data ref> cannot identify a defined variable whose <base reference> contains one or more <isub>s or asterisks.

If the <expression> in the <string option> is a <reference> to a variable, execution of the <get statement> cannot allocate, free or assign a value to the generation of storage identified by the <reference>.

Note that if neither a <file option> nor a <string option> is given in a <get statement>, the compiler supplies a <file option> of the form:

```
file(sysin)
```

Also note that a <copy option> with no <reference> is given "sysprint" as its <reference>. Refer to Section 5 for a discussion of the effect of these compiler supplied options on the establishment of declarations.

Semantics:

If the <get statement> contains a <file option>, it is executed by performing the following steps in the indicated order:

1. Evaluate the <file option> and any <skip option> or <copy option> in an unspecified order. Let f denote the file-state block identified by the value of the <file option>. Let cf denote the file-state block identified by the value of the <copy option>.

   Convert the value of the <skip option> to a fixed-point, binary, real, integer, K. If the <expression> is omitted from the <skip option>, let K be one.

2. If f is closed, open it as described in paragraph 11.3. After f is open, it must have the <stream attribute> and <input attribute>.

3. If cf is closed, open it as described in paragraph 11.3. After cf is open, it must have the <stream attribute> and <output attribute>.

4. If the <skip option> is present, evaluate it as described for a <skip format> in paragraph 12.12.

5. A <get list> is evaluated by performing the following steps in the indicated order:

   5a. Establish the next list item as described in step 8. If the list item is a scalar, consider it to be the next target. If the list item is an aggregate, consider each of its scalar components to be a target taking the elements of arrays in row-major order and the members of structures in left-to-right order. For each scalar target, T, perform steps 5b through 5d.

   5b. Scan the data stream to find the next non<space>. If the end of the data stream is encountered, signal the endfile condition.

   5c. Select the applicable case:

   Case (the current character is a comma)

   If the last operation on this file was the execution of a <get list> and its scan was stopped by a <space>, go to step 5b; otherwise, ignore this target.

Case (the current character is not a comma or a quote)

Scan to find the next <space> or comma, and let S be the string of all characters scanned, except the <space> or comma that stopped the scan. If the end of the data stream is encountered during the scan, stop the scan but do not signal the endfile or error condition.

Assign S to the target, T.

Case (the current character is a quote)

Scan to find the next single quote, and let S1 be the string of all characters scanned, except linemarks. If the end of the data stream is encountered during the scan, signal the error condition.

Scan to find the next <space> or comma, and let S2 be the string of all characters scanned, except the <space> or comma that stopped the scan. If the end of the data stream is encountered during the scan, stop the scan but do not signal the endfile or error condition.

Let S be S1||S2.

Select the applicable case:

Case (S does not satisfy the syntax for <valid field>)

Raise the conversion condition.

Case (S satisfies the syntax for <valid bit-field>)

Let S' be the string S with the trailing <radix factor> removed, the leading and trailing single quote removed, and each contained double quote replaced by a single quote. Let n be the length of S'.

If the <radix factor> is "b", let m be 1; otherwise, let m be the same as the number in the <radix factor>.

If S' is a null character-string, convert it to a null bit-string, R; otherwise, convert it to R, where R is a bit-string of length m*n. For k=1,2,...,n, bits k*m-m+1,...,k*m are obtained from the table in paragraph 2.6.2.1. If the kth character of S' is invalid, the conversion condition occurs.

Assign the bit-string R to the target T.

This page intentionally left blank

Case (S satisfies the syntax for <valid character-field>)

Let S' be the string S with the leading and trailing single quotes removed, and each contained double quote replaced by a single quote.

Assign S' to the target, T.

<valid field>::= <valid bit-field>|
<valid character-field>

<valid bit-field>::= <valid character-field><radix factor>

<radix factor>::= {b|b1|b2|b3|b4}

<valid character-field>::= "<character>..."

<character>::= ""|
Any ASCII character except quote

<space>::= ASCII blank|ASCII tab|linemark

5d. After step 5c is complete, the current character is the character following the <space> or comma, unless the end of the data stream has been reached. In the latter case, the end of the stream will be detected when the next scan is performed. The conversion or stringsize condition can result from each conversion or assignment performed by step 5c. Refer to Section 10.

6. A <get data> is evaluated be performing the following steps in the indicated order.

   6a. Scan to the the next non<space>. If the end of the data stream is encountered during the scan, signal the endfile condition. If the current character is a comma, repeat the step. If the current character is a semicolon, transfer control to the <statement> following the <get statement>. If the current character is an equals, let N be a null string and go to step 6c.

   6b. Scan to find the next equals or semicolon. If the end of the data stream is encountered during the scan, signal the error condition. If the scan was stopped by a semicolon, transfer control to the <statement> following the <get statement>; otherwise, let N be the string of all characters scanned, except linemarks and the equals that stopped the scan.

   6c. Scan to find the next non<space>. If the end of the data stream is encountered during the scan, signal the error condition. Let S3 be the string of all characters scanned, except linemarks.

   Modify the scanning rules described in step 5c so that where they scan to find a comma or <space>, they now scan to find a semicolon, comma, or <space>. Scan using the modified step 5c to obtain a string S. Do not perform the assignments to T described in step 5c.

   6d. If N does not satisfy the syntax for <stream reference>, or if N cannot be resolved such that it identifies a variable whose scope of declaration includes the <get statement>, or if N is improperly subscripted, signal the name condition with (N||"="||S3||S) as the value of the "onfield" built-in function. Refer to paragraph 10.4 and 13.5 N may contain <space>s between any of the <lexeme>s of <stream reference>, just as if <stream reference> were written in the text of an <external procedure>.

   <stream reference>::= [<identifier>[<subs>].]...
   <identifier>[<subs>]

   <subs>::= ([+|-]<decimal integer>
   [,[+|-]<decimal integer>]...)

   6e. Let T be the variable identified by N. Assign S to T using the appropriate assignment rule from step 5c.

   If the last character scanned was a semicolon, transfer control to the <statement> following the <get statement>; otherwise, go to step 6a.

7. A <get edit> is evaluated be performing the following:

   7a. For each <get edit pair> taken from left-to-right, perform steps 7b and 7c. When the last <get edit pair> has been evaluated, transfer control to the <statement> following the <get statement>.

   7b. Establish the next list item as as described in step 8. If the list item is a scalar, consider it to be the next target. If the list item is an aggregate, consider each of its scalar components to be a target taking the elements of arrays in row-major order and the members of structures in left-to-right order.

   7c. For each target, pass control to the to the <format specification list>. The evaluation of the <format specification list> causes the data stream to be scanned and a value assigned to the target. Refer to paragraph 12.12 for a description of the evaluation of a <format specification list>.

8. To evaluate a <get list> or <get edit> to obtain the next list item perform the following steps in the indicated order:

   8a. If there is no current <get item>, let the current <get item> be the leftmost <get item>; otherwise, let the current <get item> be the next <get item>. If no more <get item>s remain, transfer control to the <statement> following the <get statement>.

   8b. If the current <get item> is a <target>, evaluate the <target> as if it were the <target> of an <assignment statement> and make the evaluated <target> the current list item.

   8c. If the current <get item> is a parenthesized list of <get item>s containing a <list do>, consider the entire construct to be a do group of the form:

       <multiple do><get item>[,<get item>]...end;

       Evaluate the do group as if it were a true <group>. Each <get item> is evaluated by performing step 8b or 8c. When control reaches the end of the <group>, evaluate the next <get item>.

Each ASCII tab character encountered during scanning of the data stream sets columnposition to the next higher value in the sequence: 11,21,31,...

Each linemark encountered during scanning of the data stream sets columnposition to one.

If a pagemark or carriage return character appears in an input stream, it is ignored.

During execution of a <get statement> containing a <copy option>, all characters and linemarks in the input data stream, from the first to the last character or linemark scanned, are placed into the copy data stream attached to cf.

The copy data stream is a normal output stream and behaves as such with respect to linemarks, linesize, linenumber, etc.

If the <get statement> contains a <string option>, it is executed by performing the following:

Evaluate the <string option> and the <copy option> in an unspecified order. Convert the value of the <string option> to a character-string and let the character-string value be the data stream. For purposes of evaluating the <get list specification>, consider columnposition to be defined with an initial value of one.

Let cf be the file-state block identified by the value of the <copy option>. If cf is closed, open it as described in paragraph 11.3. After cf is opened it must have the <stream attribute> and the <output attribute>.

Evaluate the <get list specification> as described in steps 5, 6 or 7. If this evaluation would signal the name or endfile conditions, the error condition is signalled instead of the name or endfile condition. The value of "onfield" is not set when the error condition is signalled instead of the name condition.

Example:

```
get file(Input) skip list(A,B,C,(X(i) do i = 1 to 10),D);
```

Execution of this <statement> causes the data stream identified by "Input" to be advanced past the next linemark; three values are obtained and assigned to A, B and C; the <list do> is evaluated and ten values are obtained and assigned to X(1),X(2),...,X(10); finally a value is obtained and assigned to D.

Example:

```
get string(S) edit(A) (p"zzzv99");
```

Execution of this <statement> evaluates S and converts it to a character-string. The converted value is then encoded to an arithmetic value under control of the <picture>, and the encoded value is then assigned to A.

Example:

```
get file(F) data;
```

Execution of this <statement> advances the data stream identified by F and may assign values to any variable whose scope of declaration includes this <statement>. The assignment only occurs if the data stream contains "X = V", where X is a reference to the variable and V is a constant. This <statement> is extremely expensive because it causes a large symbol table to be included into the compiled program. Use of the <statement> is a poor programming practice because one cannot tell by reading the <statement> what effect its execution has on the state of the executing program.

Example:

```
get data(X,Y,Z);
```

Execution of this <statement> advances the data stream identified by "sysin" and may assign values to X,Y, or Z, or any combination of X, Y and Z. This <statement> is not quite as expensive or dangerous as the previous example, but it is to be avoided if "get list(X,Y,Z)" could be used instead.

12.15 The Goto Statement

Syntax:

<goto statement>::= [<prefix>]{goto|go to}<reference>;

Constraint:

Evaluation of the <reference> must yield a scalar label value whose block activation pointer identifies the current block activation record or the activation record of a dynamic predecessor of the current block activation. Refer to paragraph 3.3.1 for a discussion of block activation.

Semantics:

If the label value identifies a <statement> within the current block activation, control transfers to that <statement>.

If the label value identifies a <statement> within a block activation that is a dynamic predecessor of the current block activation, the current block activation and all predecessors up to, but not including the block activation identified by the label value are terminated. The block activation identified by the label value then becomes the current block activation and control transfers to the <statement> identified by the label value.

If the block activation identified by the label value is no longer active, the program is in error and the results of continued execution are undefined.

Example:

```
goto L(2);
```

12.16 The If Statement

Syntax:

```
<if statement>::= [<prefix>]if<expression><then clause>
     [<else clause>]

<then clause>::= then<executable unit>

<else clause>::= else<executable unit>

<executable unit>::= <independent statement>|<group>|
     <begin block>
```

Constraints:

Evaluation of the <expression> must yield a scalar arithmetic or string value.

Note that the scope of the <condition prefix> of an <if statement> does not include its <executable unit>s. Each <executable unit> may have its own <prefix>.

Semantics:

An <if statement> is executed by performing the following: Evaluate the <expression> and convert its value to a bit-string B. If any bit of B is a 1, execute the <then clause>; otherwise, execute the <else clause> if it is present. In all cases, pass control to the <statement> following the <if statement>.

Note that the syntax rules do not show the pairwise relationship between "then" and "else" keywords. An <else clause> is always paired with the preceding <then clause>.

Examples:

```
if x < Y
     then if a=b
          then return;
          else go to L;
```

In this example, the <else clause> belongs to the second <if statement>. If it is to belong to the first <if statement>, it must be written as:

```
if X < Y
     then if a=b
          then return;
          else;
     else go to L;
```

The <null statement> is used as the <executable unit> of the first <else clause> to produce the desired effect.

12.17 The Locate Statement

Syntax:

```
<locate statement>::= [<prefix>]locate<allocation reference>
     <locate option>...;

<allocation reference>::= <identifier>

<locate option>::= <file option>|<set option>|
     <keyfrom option>
```

<file option>::= file(<reference>)

<set option>::= set(<reference>)

<keyfrom option>::= keyfrom(<expression>)

Constraints:

No <locate option> may appear more than once and the <file option> must be present.

Evaluation of the <reference> in the <file option> must yield a scalar file value.

Evaluation of the <reference> in the <set option> must yield a generation of storage of a scalar pointer variable.

Evaluation of the <keyfrom option> must yield a scalar arithmetic or string value.

The <allocation reference> must identify a level-one based variable.

If the <set option> is omitted, the variable identified by the <allocation reference> must have been declared with a <based attribute> that contained a <locator qualifier>. That <locator qualifier> is taken as the <set option> and must satisfy the constraints of the <set option>.

Semantics:

A <locate statement> is executed by performing the following steps in the indicated order:

1. Evaluate the <locate option>s and the <allocation reference> in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   If f is not open, open it as described in paragraph 11.3. After f is opened, it must have the <record attribute> and either the <output attribute> or the <update attribute>. If the <keyfrom option> was specified, f must have the <keyed attribute>, and if f has the <keyed attribute> the <keyfrom option> must be specified.

2. If there is an output buffer associated with f, create a new record in the data set and write the content of the buffer as the value of the new record. If there is an evaluated key associated with the buffer, it is associated with the new record as its key. If any record in the data set already has this key, signal the key condition. If currentrecord is not null, and f has both the <keyed attribute> and the <sequential attribute>, and the key is not greater than the key of the record designated by currentrecord, signal the key condition.

   If f has the <keyed attribute>, create the new record in its proper position within the data set as determined by its key; otherwise, append the new record to the end of the data set. After the record is written, free the buffer and set currentrecord to designate the new record.

3. Allocate a generation of storage for the variable identified by the <allocation reference> by executing an <allocate statement> of the form:

   ```
   allocate x set(p);
   ```

   where x is the variable identified by the <allocation reference> and p is the pointer given in the <set option>. Associate this generation of storage with f as its output buffer. Only the execution of another output operation on f or the closing of f causes the buffer to be written into the data set as a new record.

4. If f has the <keyed attribute>, convert the value of the <keyfrom option> to a character-string and associate it with the output buffer as its key.

Examples:

```
locate X set(p) file(f);
p->X = 7;
locate X set(p) file(f);
p->X = 10;
close file(f);
```

This example writes two records into the data set attached to f. The first contains a 7 and the second contains a 10. The first record is not written until the second <locate statement> is executed and the second record is not written until the <close statement> is executed.

12.18 The Null Statement

Syntax:

<null statement>::= [<prefix>];

Semantics:

Execution of a <null statement> has no effect on the program. It is used primarily as a convenient way of writing an <else clause> or <on unit> that takes no action.

Note that a label value identifying a <null statement> does not compare equal to a label value identifying any other <statement>.

Examples:

```
on endpage(f);

if a = b
     then if c = b
          then go to L1;
          else;
     else go to L2;
```

12.19 The On Statement

Syntax:

<on statement>::= [<prefix>]on<condition list>[snap]<on unit>

<on unit>::= <independent statement>|<begin block>|system;

<condition list>::= <condition name>[,<condition name>]...

Each <condition name> is one of the <condition name>s given in paragraph 10.4.

Constraints:

An <on unit> consisting of an <independent statement> cannot be an <if statement>, <on statement>, <revert statement>, or <return statement>.

An <on unit> consisting of a <begin block> cannot have a <return statement> contained within the <begin block>, unless it is contained within a <procedure> contained within the <begin block>.

An <independent statement> or <begin block> used as an <on unit> cannot have a <label prefix>.

Note that the scope of a <condition prefix> on an <on statement> does not include the <on unit>.

Semantics:

The <on unit> is effectively translated into a <procedure> of the form:

```
P: procedure; <on unit> end;
```

where P is a unique name created by the compiler. The <condition name>s in the <condition list> are evaluated in an unspecified order. For each condition identified by the <condition list>, execution of an <on statement> causes the <procedure> P to be established as the current <on unit> for this condition. If this block activation has previously established an <on unit> for this condition, the previously established <on unit> is reverted.

When the condition identified by the <condition name> is signalled, the most recently established <on unit> for that condition is executed. The signal is effectively a call to the <procedure> P.

If the keyword "snap" is given, a Multics debugging command is called just prior to the invocation of the <on unit>. In an interactive process, the Multics probe command is called. In an absentee process, the Multics trace_stack command is called. Refer to the Multics PL/I Reference Manual.

If the <on unit> consists of the keyword "system", the default <on unit> for the condition is considered to be the <on unit> of this <statement>.

Refer to Section 10 for a full discussion of conditions, signals and <on unit>s.

Example:

```
on endpage(f) put page list("Page Header");
```

12.20 The Open Statement

Syntax:

```
<open statement>::= [<prefix>]open<opening>[,<opening>]...;

<opening>::= <opening option>...

<opening option>::= <file option>|<title option>|
     <linesize option>|<pagesize option>|
     <opening attribute>

<opening attribute>::= <input attribute>|<output attribute>|
     <update attribute>|<record attribute>|
     <stream attribute>|<print attribute>|
     <direct attribute>|<sequential attribute>|
     <keyed attribute>|<environment attribute>

<file option>::= file(<reference>)

<title option>::= title(<expression>)

<linesize option>::= linesize(<expression>)

<pagesize option>::= pagesize(<expression>)
```

Constraints:

Evaluation of the <reference> in each <file option> must yield a scalar file value.

Evaluation of the <expression> in a <pagesize option>, <linesize option>, or <title option> must yield a scalar arithmetic or string value.

Each <opening> can contain only one <file option>, one <title option>, one <pagesize option> and one <linesize option>. It may contain any number of <opening attribute>s.

Each <opening> must contain a <file option>.

Semantics:

An <open statement> is executed by evaluating its <opening>s from left-to-right. For each <opening> perform the following steps in the indicated order:

1. Evaluate the <file option> and any <title option>, <linesize option> and <pagesize option> in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   If a <title option> is specified, convert its value to a character-string.

   If a <pagesize option> or <linesize option> is specified, convert their values to fixed-point, binary, real values of precision (17,0).

2. If f is already open, perform no further action for this <opening>; otherwise, open f as described in paragraph 11.3.

Examples:

```
open file(f) pagesize(20) linesize(80)
     title("vfile_ >udd>database>myfile");

open file(messages) keyed update record;

open file(g) input stream environment(interactive);

open file(f), file(g);
```

12.21 The Procedure Statement

Syntax:

```
<procedure statement>::=[<condition prefix>]...<label prefix>...
     {procedure|proc}[([<parameter list>])][<procedure option>]...;

<procedure option>::= <returns attribute>|recursive|
     <reducible attribute>|<irreducible attribute>|<options attribute>

<parameter list>::= <identifier>[,<identifier>]...
```

Constraints:

Each <identifier> in the <parameter list> must identify a level-one variable declared in the <procedure> headed by this <procedure statement>.

No <label prefix> can contain a <prefix subscript>.

If control passes to the <procedure statement> by the execution of a <call statement>, the <procedure statement> cannot have a <returns attribute>.

If control passes to the <procedure statement> by the evaluation of a <function reference>, the <procedure statement> must have a <returns attribute>.

The number of <identifier>s in the <parameter list> must equal the number of <expression>s in the <argument list> of the <call statement> or <function reference> that invoked this entry.

A <procedure statement> cannot contain both a <reducible attribute> and an <irreducible attribute>.

A <procedure statement> containing a <returns attribute> must have exactly one <returns attribute> with a <returns descriptor>.

The <options attribute> may not specify the keyword "constant". The <options attribute> may specify "support", "separate_static", or "packed_decimal" only if the <procedure statement> heads an <external procedure>. The <options attribute> may specify "main" only if the <procedure statement> heads an <external procedure> and does not contain a <returns attribute>.

Semantics:

A <procedure statement> heads a <procedure> and denotes an entry to the <procedure>. When control is transferred to the entry by the evaluation of a <function reference> or the execution of a <call statement>, a new block activation of the <procedure> occurs and the arguments of the <function reference> or <call statement> are associated with the parameters in the <parameter list>. Refer to paragraph 3.6.2 for a discussion of <procedure> activation and refer to paragraph 6.10 for a description of arguments and parameters.

If control reaches a <procedure statement> as a result of completing the execution of the preceding <statement>, control is transferred to the <statement> following the <end statement> that ends the <procedure>.

Because the <label prefix> of a <procedure statement> results in the declaration of an entry constant rather than a label constant, a <goto statement> can never transfer control to a <procedure statement>.

Standard PL/I requires that recursive <procedure>s contain the keyword "recursive" in their <procedure statement>. Multics PL/I considers all <procedure>s recursive. For compatibility with standard PL/I, it accepts the keyword.

Example:

```
P: procedure(A,B,C) returns(pointer);
```

In this example, P is an entry to the <procedure> headed by the <procedure statement>. Invocation of P results in a pointer value. The entry requires three arguments that are associated with the parameters A, B, and C.

12.22 The Put Statement

Syntax:

```
<put statement>::= [<prefix>]put{<file put>|<string put>};

<file put>::= <file put option>...

<file put option>::= <file option>|<skip option>|
     <line option>|<page option>|<put list specification>

<file option>::= file(<reference>)

<skip option>::= skip[(<expression>)]

<line option>::= line(<expression>)

<page option>::= page
```

```
<put list specification>::= <put list>|<put data>|<put edit>

<put list>::= list(<put item>[,<put item>]...)

<put item>::= <expression>|
     (<put item>[,<put item>]...<list do>)

<list do>::= <multiple do>

<put data>::= data[(<put data item>[,<put data item>]...)]

<put data item>::= <reference>|
     (<put data item>[,<put data item>]...<list do>)

<put edit>::= edit<put edit pair>...

<put edit pair>::= (<put item>[,<put item>]...)
     (<format specification list>)

<string put>::= <string option><put list specification>|
     <put list specification><string option>


<string option>::= string({<reference>|<pseudo-variable>})
```

Constraints:

Evaluation of a <put item> or <put data item> must yield an arithmetic or string value. (As a nonstandard extension, evaluation of a <put item> or <put data item> may yield any type of value except an area value unless the item is contained in a <put edit pair>.)

Evaluation of the <reference> in the <file option> must yield a scalar file value.

Evaluation of the <expression> in a <line option> or <skip option> must yield a scalar arithmetic or string value.

Evaluation of the <string option> must yield a generation of storage of a scalar character-string variable.

Evaluation of a <put data item> or a <put item> must not identify the same generation of storage as the <string option>.

A <put data item> cannot identify a defined variable whose <base reference> contains one or more <isub>s or asterisks.

If the <skip option> is given in a <file put>, the <line option> or <page option> cannot also be given.

Note that if neither a <file option> nor a <string option> is given, the compiler supplies a <file option> of the form:

```
file(sysprint)
```

Refer to Section 5 for a discussion of the effect of this compiler-supplied option on the establishment of declarations.

Semantics:

In the following steps, the phrase "place a linemark into the data stream" includes the action of incrementing linenumber by one and setting columnposition to one as if a <skip format> were being evaluated at that point. The <skip format> is described in paragraph 12.12.

If the <put statement> contains a <file option>, it is executed by performing the following steps in the indicated order:

1. Evaluate the <expression>s or <reference>s immediately contained in the <file option>, <skip option> and <line option> in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   Convert the value of the <skip option> to a fixed-point, binary, integer, k. If the <expression> is omitted from the <skip option>, let k be one.

   Convert the value of the <line option> to a fixed-point, binary, integer, j.

2. If f is closed, open it as described in paragraph 11.3. After f is open, it must have the <stream attribute> and <output attribute>. If the <page option> or the <line option> is given, f must have the <print attribute>.

3. Evaluate any <page option>, <line option>, or <skip option> as if it were a <page format>, <line format>, or <skip format> as described in paragraph 12.12. If both a <page option> and a <line option> are given, the <page option> is evaluated before the <line option>.

4. If f has an <environment attribute> specifying "interactive", place a linemark into the data stream after the <put list specification> has been evaluated.

5. A <put list> is evaluated by performing the following steps in the indicated order:

   5a. Establish the next list item as described in step 8. If the list item is a scalar, consider it to be the next output value. If the list item is an aggregate, consider each of its scalar components to be an output value, taking the elements of arrays in row-major order and the members of structures in left-to-right order. For each scalar output value, perform steps 5b through 5f.

   5b. If the output value is an arithmetic or string value, convert it to a character-string according to the conversion rules given in Section 8. Otherwise, convert the output value to a character-string by using a nonstandard Multics routine. Let S be the converted character-string value.

   If the original output value was a bit-string, enclose S in quote characters and append a "b" to the end of S.

   If the original output value was a character-string, including a pictured character-string, and f does not have the <print attribute>, enclose S in quote characters and replace each contained quote by a pair of quotes; otherwise, do not modify S.

   Let S' be the final converted value to be output and let n be the length of S'.

   5c. If f has the <print attribute> and columnposition is not one or a multiple of ten plus one, place an ASCII tab character into the data stream and set columnposition to the next higher value in the sequence 11,21,31,..., unless this action would cause columnposition to exceed linesize; in that case, place a linemark into the data stream.

   5d. If n>(linesize-columnposition+1) & columnposition^=1, place a linemark into the data stream.

   5e. Place S' into the data stream using linemarks to split S', when necessary, as described for <data format> output in paragraph 12.12.

   5f. Place a single blank into the data stream.

6. A <put data> is evaluated by performing the following steps in the indicated order:

6a. If the <put data> has no <put data item>s, create a list of <put data item>s containing a <reference> to every level-one variable whose scope of declaration includes the <put statement>, but exclude any variables that violate one or more of the constraints given above. The order of the <put data item>s in the list is unspecified.

6b. Establish the next list item as described in step 8. If the list item is a scalar, consider it to be the next output value. If the list item is an aggregate, consider each of its scalar components to be an output value, taking the elements of arrays in row-major order and the members of structures in left-to-right order. For each scalar output value, perform steps 6c. through 6i.

6c. Form the character-string representation of a fully qualified <reference> to the output value with each <subscript> written as a <decimal integer> with an optional minus sign. The <reference> contains a single optional <subscript list> following the rightmost <identifier>, and contains no blanks.

Examples:

```
S.A.B(1,4,-2)

X(5)

W.Q

R
```

6d. If f has the <print attribute> and columnposition is not one or a multiple of ten plus one, place an ASCII tab character into the data stream and set columnposition to the next higher value in the sequence 11,21,31,..., unless this action would cause columnposition to exceed linesize. In that case place a linemark into the data stream.

6e. Let R be the character-string formed in step 6c. Form a character-string S consisting of:

```
R||"="
```

Let V be the character-string representation of the list item, as it would be produced by execution of a <put list> for a file not containing the <print attribute>. Let n be the length of S, and let m be the length of V.

6f. If n>(linesize-columnposition+1) & columnposition^=1, place a linemark into the data stream.

6g. Place S into the data stream using linemarks to split S, when necessary, as described for <data format> output in paragraph 12.12.

6h. If m>(linesize-columnposition+1) & columnposition^=1, place a linemark into the data stream.

6i. Place V into the data stream using linemarks to split V, when necessary, as described for <data format> output in paragraph 12.12.

6j. If this is the last scalar output value to be output by this execution of the <put statement>, place a semicolon into the data stream; otherwise, place a single blank into the data stream.

7. A <put edit> is evaluated by performing the following:

Establish the next list item as described in step 8. If the list item is a scalar, consider it to be the next output value. If the list item is an aggregate, consider each of its scalar components to be an output value,

taking the elements of arrays in row major order and the member o structures in left-to-right order. For each output value pass control t the <format specification list>. The evaluation of a <format specificatio list> causes the output value to be converted and placed into the dat stream. Refer to paragraph 12.12 for a description of the evaluation of <format specification list>.

8. To evaluate a <put list>, <put data> or <put edit> to obtain the next list item, perform the following steps in the indicated order:

   8a. If there is no current <put item>, let the current <put item> be the leftmost <put item> or <put data item>; otherwise, let the current <put item> be the next <put item> or <put data item>. If no <put item>s or <put data item>s remain, transfer control to the <statement> following the <put statement>.

   8b. If the current <put item> is an <expression> or <reference>, evaluate it to obtain its value. Let the obtained value be the current list item.

   8c. If the current <put item> is a parenthesized list of <put item>s or a parenthesized list of <put data item>s, consider the entire construct to be a do group of the form:

       <multiple do><put item>[,<put item>]...end;

       Evaluate the do group as if it were a true <group>. Evaluate each <put item> by performing steps 8b or 8c. When control reaches the end of the <group>, evaluate the next <put item> or <put data item>.

If the <put statement> contains a <string option>, it is executed by performing the following:

Evaluate the <string option> as if it were the <target> of an <assignment statement>. The evaluation must yield a generation of storage of a scalar character-string variable.

Let the generation of storage yielded by evaluation of the <string option> be the data stream, S. For purposes of evaluating the <put list specification>, consider columnposition to be defined with an initial value of one, and consider linesize to be defined with an initial value that is the length extent associated with the generation of storage of S.

Evaluate the <put list specification> as described by steps 6, 7 and 8.

When the last output value has been placed into the data stream by this <put statement>, the current length of S is defined as n, where n is columnposition-1. Let m be the length of the generation of S. If n<m and S is nonvarying, m-n blanks are used to fill the remaining characters of S. If S is varying and $n \leq m$, the current length of S is defined as n. If n>m, the error condition is signalled before the (m+1)th character is placed into the data stream.

Example:

```
put file(f) page list(A,B,C,(X(i) do i=1 to 10),D);
```

Execution of this <statement> causes the data stream identified by f to contain a pagemark; followed by the values of A, B and C; followed by the values of X(1), X(2),...,X(10); followed by the value of D. Each value begins on a columnposition that is a multiple of ten plus one. Linemarks are inserted to split the stream into lines of linesize characters each.

Example:

```
put string(S) edit(A,B)(a(10),p"99v99");
```

Execution of this <statement> causes the values of A and B to be converted to character-strings under control of the <format specification list>. The resulting string of 14 characters is the new value of S.

Example:

```
put file(F) data;
```

Execution of this <statement> causes the value of every level-one variable whose scope of declaration includes the <put statement> to be output into the data stream identified by F. Each value has the form X=V␣ except the last which has the form has X=V; where X is a <reference> to the variable whose value is given by V. This <statement> is extremely expensive because it causes a large symbol table to be included into the compiled program.

Example:

```
put data(A,B(I),C);
```

Execution of this <statement> causes the following to be placed into the data stream identified by "sysprint".

```
A=5␣␣␣␣␣␣␣␣B(10)=2␣␣␣C=7;
```

where 5 is the current value of A, 2 is the current value of B(I), 10 is the current value of I, and 7 is the current value of C.

12.23 The Read Statement

Syntax:

<read statement>::= [<prefix>]read<read option>...;

<read option>::= <file option>|<receiver>|<key spec>

<file option>::= file(<reference>)

<receiver>::= <into option>|<set option>|<ignore option>

<into option>::= into(<reference>)

<set option>::= set(<reference>)

<ignore option>::= ignore(<expression>)

<key spec>::= <key option>|<keyto option>

<key option>::= key(<expression>)

<keyto option>::= keyto(<reference>)

Constraints:

Evaluation of the <reference> in the <file option> must yield a scalar file value.

Evaluation of an <into option> must yield a generation of connected storage.

Evaluation of a <set option> must yield a generation of storage of a scalar pointer variable.

Evaluation of an <ignore option> or <key option> must yield a scalar arithmeti< or string value.

Evaluation of a <keyto option> must yield a generation of storage of a scala character-string variable.

A <read statement> must contain exactly one <file option> and exactly on< <receiver>.

A <read statement> can contain only one <key spec>, but if the <receiver> is a <ignore option>, the <read statement> cannot have a <key spec>.

Semantics:

A <read statement> is executed by performing the following steps in the indicated order:

1. Evaluate all <read option>s in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   Convert the value of the <expression> in the <key option> to a character-string.

   Convert the value of the <expression> in the <ignore option> to a fixed-point, binary, integer, k. The converted value, k, must be greater than zero.

2. If f is closed, open it as described in paragraph 11.3. After f is open, it must have the <input attribute> or the <update attribute>. If f has the <stream attribute>, go to step 10. Otherwise, if a <key option> is given, f must have the <keyed attribute>, and if f has the <direct attribute>, a <key option> must be given.

3. Free any input buffer associated with f. This circumstance occurs when the previous input operation on f was the execution of a <read statement> containing a <set option>.

4. If an <ignore option> is specified, set currentrecord to designate the (k-1)th record following the record designated by nextrecord. Signal the endfile condition if the value of k would position currentrecord off the end of the data set.

   If a <key option> is specified, set the currentrecord of f to designate the record identified by the converted value of the <expression> in the <key option>. If no such record exists in the data set attached to f, signal the key condition.

   If no <key option> or <ignore option> is specified, set currentrecord to the value of nextrecord. If nextrecord is null, signal the endfile condition.

5. If f has the <sequential attribute>, set nextrecord to designate the record following the new current record. If there is no next record, set nextrecord null.

6. If a <keyto option> is specified, assign the key associated with the current record to the variable identified by the <reference> given in the <keyto option>.

7. If an <into option> is specified, assign a copy of the current record to the variable identified by the <into option>. If the file-state block has an <environment attribute> specifying "stringvalue", and the variable, X, referenced by the <into option> is a scalar variable with the <varying attribute>, perform the assignment by executing an assignment statement of the form:

```
X = R;
```

where R is the record treated as stringvalue. If this assignment would raise the <stringsize condition>, raise <record condition> instead. Otherwise perform the assignment by executing an <assignment statement> of the form:

```
unspec(X) = unspec(R);
```

Where X is the variable referenced by the <into option> and R is the record.

If length(unspec(R))^=length(unspec(X)), signal the record condition. On return from the <on unit>, complete the assignment as if the length of R and the length of X were the minimum of the lengths of X and R.

8. If a <set option> is specified, allocate a generation of storage of sufficient size to hold a copy of the current record in "system storage" and associate the generation with f as its input buffer. Assign a copy of the current record to this buffer and assign a pointer value identifying the record in the buffer to the generation of storage identified by the <reference> given in the <set option>.

9. Transfer control to the <statement> following the <read statement>.

10. The <read statement> must not contain a <key spec> and must contain an <into option> which contains a <reference> to a scalar character-string variable.

11. If the data stream is positioned at the end of the data stream, signal the endfile condition.

12. Scan to find the next linemark, and let S be the string of all characters scanned, except the linemark that stopped the scan. If the end of the data stream is encountered during the scan, stop the scan but do not signal the endfile or error condition. Otherwise, position the data stream to the character following the linemark and set columnposition to 1.

13. Let T be the character-string variable referenced by the <into option>. Assign S to T. If this assignment would raise the <stringsize condition>, raise <record condition> instead.

Examples:

```
read file(f) into(X);

read file(g) set(p);

read file(s) ignore(n);

read file(h) key(r) into(x);

read file(f) set(p) keyto(y);
```

This page intentionally left blank.

12.24 The Return Statement

Syntax:

<return statement>::= [<prefix>]return[(<return value>)];

<return value>::= <expression>

Constraint:

If the <return value> is omitted, the current procedure block activation must have been created by the execution of a <call statement>. If the <return value> is present, the current procedure block activation must have been created by the evaluation of a <function reference>.

Semantics:

If the program is executing within a run unit, and the current procedure block activation is the first activation of a <procedure> headed by a <procedure statement> containing an <options attribute> specifying the keyword "main", the effect is as if a <stop statement> had been executed.

If the <return value> is omitted, a <return statement> is executed by terminating the current procedure block activation and transferring control to the <statement> following the <call statement> whose execution created the current procedure block activation. The preceding block activation is the new current block activation.

If the <return value> is present, a <return statement> is executed by evaluating the <return value> and promoting its value to conform to the aggregate type specified in the <returns attribute> of the <entry statement> or <procedure statement> used to enter the current procedure block activation. The promoted value is converted to conform to the data type specified in the <returns attribute>. The promoted and converted value is the value of the <function reference> whose evaluation created the current procedure block activation. The current procedure block activation is terminated and control returns to continue evaluation of the <statement> containing the <function reference>. The preceding block activation is the new current block activation.

Refer to paragraph 3.3.1 for a discussion of block activation and to paragraphs 8 and 9 for discussions of conversion and promotion.

Examples:

```
return;

return(a+b);
```

12.25 The Revert Statement

Syntax:

<revert statement>::= [<prefix>]revert<condition list>;

<condition list>::= <condition name>[,<condition name>]...

Constraint:

Each <condition name> must be one of the <condition name>s given in paragraph 10.4.

Semantics:

A <revert statement> is executed by evaluating the <condition name>s in an unspecified order. For each of the specified <condition name>s, revert the associated <on unit> if it was established by the current block activation. Refer to Section 10 for a full discussion of conditions.

Examples:

```
revert endpage(f);

revert underflow, overflow;
```

12.26 The Rewrite Statement

Syntax:

<rewrite statement>::= [<prefix>]rewrite<rewrite option>...;

<rewrite option>::= <file option>|<key option>|<from option>

<file option>::= file(<reference>)

<key option>::= key(<expression>)

<from option>::= from(<reference>)

Constraints:

A <rewrite statement> must contain exactly one <file option> and cannot contain more than one <key option> or more than one <from option>.

Evaluation of the <reference> in the <file option> must yield a scalar file value.

Evaluation of the <key option> must yield a scalar arithmetic or string value.

Evaluation of the <from option> must yield a generation of connected storage.

Semantics:

A <rewrite statement> is executed by performing the following steps in the indicated order:

1. Evaluate the <rewrite option>s in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   Convert the value of the <expression> in the <key option> to a character-string.

2. If f is closed, open it as described in paragraph 11.3. After f is opened, it must have the <update attribute>. If a <key option> is specified, f must have the <keyed attribute>.

3. If a <key option> is specified, set the currentrecord of f to designate the record identified by the converted value of the <key option>. If no such record exists in the data set attached to f, signal the key condition.

   If a <key option> is specified and f has the <sequential attribute>, set nextrecord to designate the record following the new current record. If there is no next record, set nextrecord null.

   If no <key option> is specified, currentrecord must not be null.

4. If the file-state block has an <environment attribute> specifying "stringvalue", a <from option> is specified, and the variable, X, referenced by the <from option> is a scalar variable with the <varying attribute>, then replace the record designated by the current record with a string equal to the current value of X.

   If a <from option> is specified, and the preceding paragraph does not apply, replace the record designated by currentrecord with a copy of the variable identified by the <reference> in the <from option>. If f does not have the <keyed attribute> and the size of the variable is not equal to the size of the record designated by currentrecord, signal the record condition. If control returns from the <on unit>, transfer control to the <statement> following the <rewrite statement>. In that case, the record designated by currentrecord remains unchanged.

   If no <from option> is specified, there must be an input buffer associated with f. This input buffer will be present only if the last input operation on f was the execution of a <read statement> containing a <set option>. When a <from option> is not specified, replace the record designated by currentrecord with a copy of the record in the input buffer.

Examples:

```
rewrite file(f) from(x);

rewrite file(g) from(x) key(y);
```

12.27 The Signal Statement

Syntax:

<signal statement>::= [<prefix>]signal<condition name>;

Constraint:

The <condition name> must be one of the <condition name>s defined in paragraph 10.4.

Semantics:

If detection of the condition identified by the <condition name> is disabled, transfer control to the <statement> following the <signal statement>; otherwise, perform the following actions in the indicated order:

1. For the conditions listed below, stack the current values of their associated built-in functions and assign the built-in function the default value shown below.

Condition	Builtin Function	Value
conversion	onsource	null string
conversion	onchar	blank
name(f)	onfield	null string
key(f)	onkey	null string
*endfile(f)	onkey	null string
*transmit(f)	onkey	null string
*record(f)	onkey	null string

   * Only set to this value when f has the <keyed attribute>.

2. Signal the condition. A signal for condition, c, causes the most recently established <on unit> for c to be invoked as a <procedure>. Refer to Section 10 for a full discussion of conditions and signals.

Examples:

```
signal condition(x);

signal zerodivide;
```

12.27a The Stop Statement

Syntax:

<stop statement>::= [<prefix>]stop;

Semantics:

If the program is executing within a run unit, execution of the program is terminated by performing the following steps:

1. Raise the <finish condition>.
2. Terminate all block activations.
3. Close all open files.
4. Terminate the run unit.

If the program is not executing within a run unit, the system condition command_abort_ is signalled. The standard system action for command_abort_ is to return control to the Multics command processor in such a way that the remainder of the current command line is executed.

12.28 The Write Statement

Syntax:

<write statement>::= [<prefix>]write<write option>...;

<write option>::= <file option>|<from option>|
<keyfrom option>

<file option>::= file(<reference>)

<from option>::= from(<reference>)

<keyfrom option>::= keyfrom(<expression>)

Constraints:

Evaluation of the <reference> in the <file option> must yield a scalar file value.

Evaluation of the <expression> in the <keyfrom option> must yield a scalar arithmetic or string value.

Evaluation of the <from option> must yield a generation of connected storage.

A <write statement> must contain exactly one <file option> and exactly one <from option>. It cannot contain more than one <keyfrom option>.

Semantics:

A <write statement> is executed by performing the following steps in the indicated order:

1. Evaluate the <write option>s in an unspecified order.

   Let f denote the file-state block identified by the value of the <file option>.

   Convert the value of the <expression> in the <keyfrom option> to a character-string.

2. If f is closed, open it as described in paragraph 11.3. After f is opened, if it has the <stream attribute>, then go to step 9; otherwise, f must have the <record attribute>. It cannot have the <input attribute>. If it has the <update attribute>, it must also have the <keyed attribute>. If the <keyfrom option> is specified, f must have the <keyed attribute>; if f has the <keyed attribute>, the <keyfrom option> must be specified.

3. If an output buffer is associated with f, create a new record in the data set and write the content of the buffer as the value of the new record. If an evaluated key is associated with the buffer, associate it with the record as its key. If any record in the data set already has this key, signal the key condition.

   If f has the <keyed attribute>, create the new record in its proper position within the data set as determined by its key; otherwise, append the new record to the end of the data set.

   After the record is written, free the output buffer. An output buffer exists when the previous output operation on f was the execution of a <locate statement>.

4. If the <keyfrom option> is specified and the data set already contains a record whose associated key is the converted value of the <keyfrom option>, signal the key condition. If currentrecord is not null, and f has both the <keyed attribute> and the <sequential attribute>, and the converted value of the <keyfrom option> is not greater than the key of the record designated by currentrecord, signal the key condition.

5. If f has the <keyed attribute>, create the new record in its proper position within the data set as determined by its key; otherwise, append the new record to the end of the data set. If the variable, X, referenced by the <from option> is a scalar variable with the <varying attribute>, and if the file-state block has an <environment attribute> specifying "stringvalue", the record is a string equal to the current value of X. Otherwise, the record is a copy of the content of the generation identified by the evaluated <from option>.

6. Associate the converted value of the <keyfrom option> with the new record as its key.

7. Set currentrecord to designate the new record and set nextrecord to designate the record following the new record if one exists. If no such record exists, set nextrecord null.

8. Transfer control to the <statement> following the <write statement>.

9. f must have the <output attribute>. The <write statement> must not have a <keyfrom option> and must have a <from option> that references a scalar character-string variable.

10. Let the variable referenced by the <from option> be S. If length(S)<=linesize-columnposition+1, place the value of S into the data stream; otherwise, signal the <record condition>. Upon return from the <on unit>, place substr(S,1,linesize-columnposition+1) into the data stream.

11. Place one linemark into the data stream, set columnposition to one, and add one to linenumber. If linenumber = pagesize+1, then signal the <endpage condition>.

Examples:

```
write file(f) from(x) keyfrom(y);

write file(g) from(x);
```

SECTION 13

BUILT-IN FUNCTIONS

The functions described in this section are an intrinsic part of the language. Functions marked with + are not part of standard PL/I but are supported by Multics PL/I. For descriptive convenience the built-in functions are grouped into six classes.

1. String Built-in Functions.

after	collate	index	reverse	verify
before	+collate9	length	+rtrim	
bit	copy	low	+search	
bool	decat	+ltrim	string	
+byte	high	+maxlength	substr	
character	+high9	+rank	translate	

2. Arithmetic Built-in Functions.

abs	decimal	max	real
add	divide	min	round
binary	fixed	mod	sign
ceil	float	multiply	subtract
complex	floor	precision	trunc
conjg	imag		

3. Mathematical Built-in Functions.

acos	cos	exp	sind
asin	cosd	log	sinh
atan	cosh	log10	sqrt
atand	erf	log2	tan
atanh	erfc	sin	tand
			tanh

4. Array Built-in Functions.

dim	hbound	prod	sum
dot	lbound		

5. Condition Built-in Functions.

onchar	onfield	onkey	onsource
oncode	onfile	onloc	

6. Miscellaneous Built-in Functions.

addr	+convert	+nullo	+stackbaseptr
+addrel	+currentsize	offset	+stackframeptr
allocation	date	pageno	+stacq
+baseno	empty	pointer	time
+baseptr	+environmentptr	+rel	unspec
+clock	lineno	+size	valid
+codeptr	null	+stac	+vclock

To facilitate the description of the built-in functions, each function is described in terms of one or more examples. Built-in functions are referenced with a <function reference> as described in paragraph 6.8. If a function allows a variable number of arguments, the examples show all possible forms of the <function reference>. Unless the description of a specific function states otherwise, all arguments can be <expression>s.

13.1 String Built-in Functions

When a description of a function indicates that its argument is to be converted to a character-string, the conversion occurs as if the argument were an operand of the "||" infix operator. If the argument is to be converted to a bit-string, the conversion occurs as if the argument were an operand of the "|" infix operator. Refer to Sections 7 and 8.

Unless the description of a specific function states otherwise, the function can be invoked with scalar or aggregate arguments. When invoked with one or more aggregate arguments, all arguments are promoted to the highest common aggregate type as if they were operands of an infix operator. Refer to Sections 7 and 9.

13.1.1 After

Example:

```
after(S,C)
```

S and C are converted to S' and C'. If both S and C are bit-strings, S' and C' are bit-strings; otherwise, S' and C' are character-strings.

If S' is a bit-string, the result R is a bit-string; otherwise, R is a character-string.

If C' does not occur as a substring within S', or if S' is a null string, R is a null string.

If C' is a null string, R is S'.

If C' is a substring within S', let i be the position within S' of the rightmost character or bit of the leftmost substring C', and let n be the length of S'.

If i=n, R is a null string.

If i<n, R is substr(S',i+1).

13.1.2 Before

Example:

```
before(S,C)
```

S and C are converted to S' and C'. If both S and C are bit-strings, S' and C' are bit-strings; otherwise, S' and C' are character-strings.

If S' is a bit-string, the result R is a bit-string; otherwise, R is a character-string.

If S' or C' is a null string, R is a null string.

If C' is not a null string and does not occur as a substring within S', R is S'.

If C' is a substring within S', let i be the position within S' of the first character or bit of the leftmost substring C'.

If i=1, R is a null string.

If i>1, R is substr(S',1,i-1).

This page intentionally left blank.

13.1.3 Bit

Example:

```
bit(S) or bit(S,L)
```

L is converted to L', where L' is a fixed-point, binary, real value of precision (24,0). L' must be a nonnegative scalar value.

If L is given, S is converted to a bit-string of length L'.

If L is not given, S is converted to a bit-string S' as described in paragraph 13.1.

The result R is a bit-string whose length is the length of S' and whose value is the value of S'.

13.1.4 Bool

Example:

```
bool(X,Y,W)
```

X, Y and W are converted to bit-string values X', Y' and W'. The length of W' is 4 bits. The shorter of X' or Y' is extended on the right with zero bits until it is the length of the longer string.

The result R is a bit-string.

If both X' and Y' are null strings, R is a null string.

If X' and Y' are not null strings, the length of R is the common length of X' and Y'. The kth bit of R is given in the following table. M1, M2, M3 and M4 are the four bits of W'.

X'k	Y'k	Rk
0	0	M1
0	1	M2
1	0	M3
1	1	M4

13.1.4a Byte

Example:

```
byte(X)
```

byte is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

X is converted to X', where X' is a fixed-point, binary, real, value of precision (9,0). X' must be a nonnegative value.

The result R is a character-string of length 1.

The value of R is substr(collate9(),X'+1,1).

This page intentionally left blank.

13.1.5 Character

Example:

```
character(S) or character(S,L)
```

L is converted to L', where L' is a fixed-point, binary, real, value of precision (24,0). L' must be a nonnegative scalar value.

If L is given, S is converted to a character-string, S', of length L'.

If L is not given, S is converted to a character-string S' as described in paragraph 13.1.

The result R is a character-string whose length is the length of S' and whose value is the value of S'.

The character built-in function has two names: character and char.

Example:

collate() or collate

The result is a character-string of length 128 that consists of the set of characters in the Multics ASCII character set in ascending order. The Multics ASCII character set is defined in the "Multics Programmers' Manual".

13.1.6a Collate9

Example:

collate9() or collate9

The result is a character-string of length 512 that consists of the set of characters in the Multics Extended Character Set in ascending order. The Multics Extended Character Set is defined in the MPM Reference Guide, Order No. AG91.

13.1.7 Copy

Example:

copy(S,N)

N is converted to N', where N' is a fixed-point, binary, real, value of precision (24,0). N' must be a nonnegative scalar value.

If S is a bit-string, it is converted to a bit-string S'; otherwise, it is converted to a character-string S'.

The result R is a string of the same type as S'.

If N' = 0, R is a null string. If N' = 1, the value of R is S'. If N' > 1, the value of R is the value of S' concatenated with itself N'-1 times.

13.1.8 Decat

Example:

decat(S,C,X)

X is converted to a bit-string X' of length 3.

If both S and C are bit-strings, they are converted to bit-strings, S' and C'; otherwise, they are converted to character-strings S' and C'.

The result R is a string of the same type as S'. The value of R is given by the following:

If C' is a null string, the value of R depends on X' as shown in the table below:

X'	R
000	A null string
001	S'
010	A null string
011	S'
100	A null string
101	S'
110	A null string
111	S'

If C' is not a null string, and C' is not a substring of S', the value of R depends on X' as shown in the table below:

X'	R
000	A null string
001	A null string
010	A null string
011	A null string
100	S'
101	S'
110	S'
111	S'

If C' is not a null string, and C' is a substring of S', the value of R depends on X' as shown in the table below:

X'	R'
000	A null string
001	after(S',C')
010	C'
011	C'||after(S',C')
100	before(S',C')
101	before(S',C')||after(S',C')
110	before(S',C')||C'
111	S'

13.1.9 High

Example:

```
high(N)
```

N is converted to N', where N' is a fixed-point, binary, real, value of precision (24,0). N' must be a nonnegative scalar value.

The result R is a string of PAD characters of length N'. A PAD character is the highest character in the Multics ASCII character set as defined in "The Multics Programmers' Manual".

13.1.9a High9

Example:

```
high9(N)
```

N is converted to N', where N' is a fixed-point, binary, real, value of precision (24,0). N' must be a nonnegative scalar value.

The result R is a string of characters of length N', all of whose bits are one-bits. This is the highest character in the Multics Extended Character Set as described in the "Multics Programmers' Manual."

13.1.10 Index

Example:

```
index(S,C)
```

If both S and C are bit-strings they are converted to the bit-strings S' and C'; otherwise, they are converted to the character strings S' and C'.

The result R is a fixed-point, binary, real, value of precision (24,0).

If either S' or C' is a null string or if C' is not contained as a substring within S', R is zero; otherwise, R is the position within S' of the first character or bit of the leftmost substring C'.

13.1.11 Length

Example:

```
length(S)
```

If S is a bit-string, it is converted to a bit-string S'; otherwise, it is converted to a character-string S'.

The result R is a fixed-point, binary, real, value of precision (24,0).

The value of R is the length of S'.

13.1.12 Low

Example:

```
low(N)
```

N is converted to N', where N' is a fixed-point, binary, real, value of precision (24,0). N' must be a nonnegative scalar value.

The result R is a string of NUL characters of length N'. A NUL character is the lowest character in the Multics ASCII character set as defined in the "Multics Programmers' Manual."

13.1.12a Ltrim

Example:

```
ltrim(S,C) or ltrim(S)
```

ltrim is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

S and C are converted to the character-strings S' and C'. If C is omitted, the value of C' is a single blank character. The result R is a character-string.

To determine the value of R, let n be the length of S'.

If n is zero then R is the null character-string. Otherwise, for k=1,2,...,n, the kth character of S', S'k, is tested to see if it occurs in C'. Let m be the first value of k for which the test fails; or if the test succeeds for all values of k, m=n+1.

The length of the result R is l=n-m+1. For k=1,2,...,l, Rk=S'k+m-1.

13.1.12b Maxlength

Example:

```
maxlength(S)
```

maxlength is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

If S is a bit-string, it is converted to a bit-string S'; otherwise, it is converted to a character-string S'.

The result R is a fixed-point binary, real value of precision (24,0).

The value of R is the maximum length of S'.

NOTE: The maxlength built-in function differs from the length built-in function only when S is a varying bit-string or a varying character-string. In all other cases both built-in functions return the same result.

13.1.12c Rank

Example:

```
rank(X)
```

rank is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

X must be a character-string of length 1.

The result R is a fixed-point, binary, real, value of precision (9,0).

The value of R is index(collate9(),X)-1.

13.1.13 Reverse

Example:

```
reverse(S)
```

If S is a bit-string, it is converted to a bit-string S'; otherwise, it is converted to a character-string S'.

The result R is a string whose type and length are the type and length of S'. The kth bit or character in R is the (n-k+1)th bit or character in S', where n is the length of S', and k=1,2,...,n.

13.1.13a Rtrim

Example:

```
rtrim(S,C) or rtrim(S)
```

The rtrim function is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

S and C are converted to the character-strings S' and C'. If C is omitted, the value of C' is a single blank character. The result R is a character-string.

To determine the value of R, let n be the length of S'. For k=n,n-1,...,1, the kth character of S', S'k, is tested to see if it occurs in C'. Let m be the first value of k for which the test fails; or if the test succeeds for all values of k, m=0.

The length of the result R is l=m. For k=1,2,...,m, Rk=S'k.

13.1.14 Search

Example:

```
search(S,C)
```

The search function is a nonstandard built-in function; its use makes programs dependent on Multics PL/I.

S and C are converted to the character-strings S' and C'.

The result R is a fixed-point, binary, real, value of precision (24,0).

If S' is a null string, R is zero; otherwise to determine the value of R, let n be the length of S'. For k=1,2,...,n, the kth character of S' is tested to see if it occurs in C'. R is the first value of k for which the test succeeds; or if no character of S' occurs in C', R is zero.

13.1.15 String

Example:

```
string(S)
```

S must be an arithmetic or string scalar value, or it must be an aggregate of string data suitable for use in string overlay defining as described in paragraph 4.3.3.6.

If S is a scalar, other than a bit-string, it is converted to a character-string S'; otherwise, let S' be S.

The result R is a string whose type and value are the type and value of S'. If S' is an aggregate, the type of R is the type of the components of S', and the value of R is the concatenation of all scalar components of S'.

13.1.16 Substr

Example:

```
substr(S,I,J) or substr(S,I)
```

I and J are converted to I' and J', where I' and J' are fixed-point, binary, real, values of precision(24,0).

If S is a bit-string, it is converted to a bit-string S'; otherwise, it is converted to a character-string S'.

The result R is a string of the same type as S'.

To determine the value of R, let i=I' and, if J is given, let j=J'; otherwise, let j=n-i+1, where n is the length of S'.

If $(0 \leq i-1 \leq j+i-1 \leq n)$ is not satisfied, the stringrange condition occurs. Unless detection of the condition has been enabled the program is in error and the results of continued execution are undefined.

If the inequalities are satisfied, R is a string of length j. The kth character or bit of R is the (i+k-1)th character or bit of S'.

13.1.17 Translate

Example:

```
translate(S,T) or translate(S,T,X)
```

S, T and X are converted to the character-strings S', T' and X'. If X is omitted, X' is the value of collate9(). If T' is shorter than X', it is padded on the right with blanks until, it is the length of X'.

The result R is a character-string of the length of S'.

Let n be the length of S'. For k=1,2,...,n, determine Rk by the following:

Let i be given by index(X',S'k). If i=0, Rk=S'k; otherwise, Rk is the ith character of T'.

13.1.18 Verify

Example:

```
verify(S,C)
```

S and C are converted to the character-strings S' and C'.

The result R is a fixed-point, binary, real, value of precision (24,0).

If S' is a null string, R is zero; otherwise to determine the value of R, let n be the length of S'. For k=1,2,...,n, the kth character of S' is tested to see if it occurs in C'. R is the first value of k for which the test fails; or if every character of S' occurs in C', R is zero.

13.2 Arithmetic Built-in Functions

When the description of a specific function requires that its arguments be converted to the "common" type, base and mode, they are converted as if they were operands of the infix operator "+".

When the description of a specific function requires that a single argument be converted to arithmetic type, the argument is converted as if it were an operand of the prefix operator "+". Refer to Sections 7 and 8 for a discussion of conversion.

Unless the description of a specific function states otherwise, the function can be invoked with scalar or aggregate arguments. When invoked with one or more aggregate arguments, all arguments are promoted to the highest common aggregate type as if they were operands of an infix operator. Refer to Sections 7 and 9.

Each function is described as operating on scalar values and yielding a scalar result. When given aggregate arguments, the function is applied to corresponding scalar components of the promoted aggregate arguments and produces the corresponding scalar component of the aggregate result. The order of evaluation of the scalar components is not defined. The result is an aggregate of the same aggregate type as the promoted aggregate arguments.

13.2.1 Abs

Example:

```
abs(X)
```

X must be an arithmetic or string value.

X is converted to X'. The type, base, mode and precision of X' are given in paragraph 13.2.

The result R has the type and base of X'.

The mode of R is real.

If the type of X' is fixed-point and the mode of X' is complex, the precision of R is:

```
(min(N,p+1),q)
```

where N is 71 if the base of X' is binary and 59 if the base of X' is decimal, and (p,q) is the precision of X'.

Otherwise, the precision of R is the precision of X'.

If X' is complex, the value of R is the positive square root of x**2+y**2, where x and y are the real and imaginary parts of X'.

If X' is real, the value of R is X' if X'>0; otherwise, it is -X'.

13.2.2 Add

Example:

```
add(X,Y,P) or add(X,Y,P,Q)
```

X and Y must be arithmetic or string values, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. X and Y are converted to X' and Y', where the type, base and mode of X' and Y' are the common type, base and mode as defined in paragraph 13.2.

If the common type is fixed-point and Q is not given, Q is assumed to be zero. If the common type is floating-point, Q must not be given. Q must be in the range $-128 \leq Q \leq 127$.

If the common base is decimal, let N be 59. If the common base is binary and the common type is fixed-point, let N be 71. If the common base is binary and the common type is floating-point, let N be 63. P must be less than or equal to N.

The result R has the common type, base and mode.

If R is floating-point, its precision is (P); otherwise, it is (P,Q).

The value of R is X'+Y'.

13.2.3 Binary

Example:

```
binary(X) or binary(X,P) or binary(X,P,Q)
```

X must be an arithmetic or string value, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. If P or P and Q are given, they are the precision of the result. If the result is floating-point, P cannot exceed 63; otherwise, P cannot exceed 71. If the result is floating-point, Q cannot be given. If P, but not Q, is given and the result type is fixed-point, Q is assumed to be zero. If given, Q must be in the range $-128 \leq Q \leq 127$.

The result is formed by converting X to a binary arithmetic value according to the conversion rules given in paragraph 8.2. If P or P and Q are given they are the target precision; otherwise, the target precision is determined by the conversion rules of paragraph 8.2.10.

The binary built-in function has two names: binary and bin.

13.2.4 Ceil

Example:

```
ceil(X)
```

X must be an arithmetic or string value.

X is converted to X'. The type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be real.

The result R has the type, base and mode of X'.

If the type of R is fixed-point, the precision of R is:

```
(min(N,max(p-q+1,1)),0)
```

where N is 71 if X' is binary and N is 59 if X' is decimal, and (p,q) is the precision of X'.

If the type of R is floating-point, the precision of R is the precision of X'.

The value of R is the smallest integer $\geq$ X'.

13.2.5 Complex

Example:

```
complex(X,Y)
```

X and Y must be arithmetic or string values.

X and Y are converted to X' and Y', where the type, base and mode of X' and Y' are the common type, base and mode as defined in paragraph 13.2. The common mode must be real.

The result R is a complex value whose type and base are the common type and base.

If the type of R is fixed-point, the precision of R is:

```
(min(N,max(px-qx,py-qy)+max(qx,qy)),max(qx,qy))
```

where N is 59 if the base of R is decimal and N is 71 if the base of R is binary. (px,qx) is the precision of X' and (py,qy) is the precision of Y'.

If the type of R is floating-point, the precision of R is:

```
min(N,max(px,py))
```

where px is the precision of X' and py is the precision of Y'. If the base of R is binary, N is 63; otherwise N is 59.

The result R is a complex value whose real part is X' and whose imaginary part is Y'.

The complex built-in function has two names: complex and cplx.

13.2.6 Conjg

Example:

```
conjg(X)
```

X must be an arithmetic or string value.

X is converted to X'. The type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be complex.

The result R has the type, base, mode and precision of X'.

The value of R is the conjugate of X'. The conjugate of a complex number is the complex number with the sign of its imaginary part reversed.

13.2.7 Decimal

Example:

```
decimal(X) or decimal(X,P) or decimal(X,P,Q)
```

X must be an arithmetic or string value, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. If P or P and Q are given, they are the precision of the result. P cannot exceed 59. If the result is floating-point, Q cannot be given. If P, but not Q, is given and the result type is fixed-point, Q is assumed to be zero. If given, Q must be in the range $-128 \leq Q \leq 127$.

The result is formed by converting X to a decimal arithmetic value according to the conversion rules given in paragraph 8.2. If P or P and Q are given, they are the target precision; otherwise, the target precision is determined by the conversion rules of paragraph 8.2.

The decimal built-in function has two names: decimal and dec.

13.2.8 Divide

Example:

```
divide(X,Y,P) or divide(X,Y,P,Q)
```

X and Y must be arithmetic or string values, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. X and Y are converted to X' and Y', where the type, base and mode of X' and Y' are the common type, base and mode as defined in paragraph 13.2.

If the common type is fixed-point and Q is not given, Q is assumed to be zero. If the common type is floating-point, Q must not be given. Q must be in the range $-128 \leq Q \leq 127$.

If the common base is decimal, let N be 59. If the common base is binary and the common type is fixed-point, let N be 71. If the common base is binary and the common type is floating-point, let N be 63. P must be less than or equal to N.

The result R has the common type, base and mode. The precision of R is (P,Q) or (P).

If Y' = 0, the zerodivide condition occurs. Unless detection of the condition has been enabled, the program is in error and the results of continued execution are undefined.

R is the value of X'/Y'.

13.2.9 Fixed

Example:

```
fixed(X) or fixed(X,P) or fixed(X,P,Q)
```

X must be an arithmetic or string value, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. If P or P and Q are given, they are the precision of the result. If the result is decimal, P cannot exceed 59 and if the result is binary, P cannot exceed 71. If P is given and Q is not, the precision of the result is (P,0). Q must be in the range $-128 \leq Q \leq 127$.

The result is formed by converting X to a fixed-point arithmetic value according to the conversion rules given in paragraph 8.2. If P or P and Q are given, they are the target precision; otherwise, the target precision is determined by the conversion rules of paragraph 8.2.

13.2.10 Float

Example:

```
float(X) or float(X,P)
```

X must be an arithmetic or string value and P must be a <decimal integer>.

No conversion or promotion is performed for P. If the base of the result is binary, P must not exceed 63; otherwise, P must not exceed 59.

The result is formed by converting X to a floating-point arithmetic value according to the conversion rules given in paragraph 8.2. If P is given, it is the target precision; otherwise, the target precision is determined by the rules of paragraph 8.2.

13.2.11 Floor

Example:

```
floor(X)
```

X must be an arithmetic or string value.

X is converted to X'. The type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be real.

The result R has the type, base and mode of X'.

If the type of R is fixed-point, the precision of R is:

```
(min(N,max(p-q+1,1)),0)
```

where N is 71 if X' is binary and N is 59 if X' is decimal, and (p,q) is the precision of X'.

If the type of R is floating-point, the precision of R is the precision of X'.

The value of R is the largest integer that is $\leq$ X'.

13.2.12 Imag

Example:

```
imag(X)
```

X must be an arithmetic or string value.

X is converted to X'. The type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be complex.

The result R has the type, base and precision of X' but its mode is real.

The value of R is the imaginary part of X'.

13.2.13 Max

Example:

```
max(X1,X2,...,Xn)
```

Each Xj must be an arithmetic or string value and n must be greater than 1.

Each Xj is converted to X'j, where X'j has the common type, base and mode of the given arguments. The common mode must be real.

The result R has the common type, base and mode.

If the common type is fixed-point, the precision of R is:

```
(min(N,max(p1-q1,...,pn-qn)+
max(q1,...,qn)), max(q1,...,qn))
```

where N is 71 if the common base is binary and N is 59 if the common base is decimal. (pj,qj) is the precision of X'j.

If the common type is floating-point, the precision of R is:

```
max(p1,...,pn)
```

where pj is the precision of X'j.

The value of R is the maximum value of X'1, X'2, ..., X'n.

13.2.14 Min

Example:

```
min(X1,X2,...,Xn)
```

Each Xj must be an arithmetic or string value and n must be greater than 1.

Each Xj is converted to X'j, where X'j has the common type, base and mode of the given arguments. The common mode must be real.

The result R has the common type, base and mode.

If the common type is fixed-point, the precision of R is:

```
(min(N,max(p1-q1,...,pn-qn)+
max(q1,...,qn)), max(q1,...,qn))
```

where N is 71 if the common base is binary and N is 59 if the common base is decimal. (pj,qj) is the precision of X'j.

If the common type is floating-point, the precision of R is:

```
max(p1,...,pn)
```

where pj is the precision of X'j.

The value of R is the minimum value of X'1, X'2, ..., X'n.

13.2.15 Mod

Example:

```
mod(X,Y)
```

X and Y must be arithmetic or string values.

X and Y are converted to X' and Y', where the type, base and mode of X' and Y' are the common type, base and mode as described in paragraph 13.2. The common mode must be real.

The result R has the common type, base and mode.

If the common type is fixed-point, the precision of R is:

```
(min(N,py-qy+max(qx,qy)),max(qx,qy))
```

where N is 71 if the common base is binary and N is 59 if the common base is decimal. (px,qx) is the precision of X' and (py,qy) is the precision of Y'.

If the common type is floating-point, the precision of R is:

```
max(px,py)
```

where px is the precision of X' and py is the precision of Y'.

If Y' = 0, R is X'; otherwise, R is X'-Y'*floor(X'/Y').

13.2.16 Multiply

Example:

```
multiply(X,Y,P) or multiply(X,Y,P,Q)
```

X and Y must be arithmetic or string values, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. X and Y are converted to X' and Y', where the type, base and mode of X' and Y' are the common type, base and mode as defined in paragraph 13.2.

If the common type is fixed-point and Q is not given, Q is assumed to be zero. If the common type is floating-point, Q must not be given. Q must be in the range $-128 \leq Q \leq 127$.

If the the common base is decimal, let N be 59. If the the common base is binary and the common type is fixed-point, let N be 71. If the common base is binary and the common type is floating-point, let N be 63. P must be less than or equal to N.

The result R has the common type, base and mode. The precision of R is (P,Q) or (P).

The value of R is X'*Y'.

13.2.17 Precision

Example:

```
precision(X,P) or precision(X,P,Q)
```

X must be an arithmetic or string value, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q.

X is converted to X', where the type, base, mode and precision of X' are determined as follows:

If X is arithmetic the type, base and mode of X' are the type, base and mode of X.

If X is a character-string, the type of X' is fixed-point, the base is decimal and the mode is real.

If X is a bit-string, the type of X' is fixed-point, the base is binary and the mode is real.

The precision of X' is (P,Q) if X' is fixed point; and is P if X' is floating point.

If the type of X' is fixed-point and Q is not given, Q is assumed to be zero. Q must be in the range $-128 \leq Q \leq 127$. If the type of X' is floating-point, Q cannot be given.

If the base of X' is decimal, P cannot greater than 59. If the base of X' is binary and the type is fixed-point, P cannot be greater than 71; while if the type of X' is floating-point, P cannot be greater than 63.

The result R has the type, base, mode and precision of X'.

The value of R is the value of X'.

The precision built-in function has two names: precision and prec.

13.2.18 Real

Example:

```
real(X)
```

X must be an arithmetic or string value.

X is converted to X', where the type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be complex.

The result R has the type, base and precision of X' but its mode is real.

The value of R is the real part of X'.

13.2.19 Round

Example:

```
round(X,K)
```

X must be an arithmetic or string value, and K must be an optionally signed <decimal integer>.

No conversion or promotion is performed for K. X is converted to X', where the type, base, mode and precision of X' are given in paragraph 13.2.

The result R has the type, base and mode of X'.

If X' is fixed-point, the precision of R is:

```
(max(1,min(p-q+1+K,N)),K)
```

where N is 71 if the base of X' is binary and N is 59 if the base of X' is decimal, and (p,q) is the precision of X'.

If X' is floating-point, K must be greater than zero, and the precision of R is:

```
(min(K,N))
```

where N is 59 if X' is decimal and N is 63 if X' is binary.

The value of R is:

```
sign(X')*floor(abs(X')*(b**n)+0.5)/(b**n)
```

where b is 2 if the base of X' is binary, and b is 10 if the base of X' is decimal. If the type of X' is fixed-point, n=K; otherwise, n=K-e, where e is the exponent of X'.

If the mode of X' is complex, R is the value of X' with its real and imaginary parts rounded as described above for real numbers.

13.2.20 Sign

Example:

```
sign(X)
```

X must be an arithmetic or string value.

X is converted to X', where the type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be real.

The result R is a fixed-point, binary, real value of precision (17,0).

If X' < 0, the value of R is -1.

If X' = 0, the value of R is 0.

If X' > 0, the value of R is +1.

13.2.21 Subtract

Example:

```
subtract(X,Y,P) or subtract(X,Y,P,Q)
```

X and Y must be arithmetic or string values, and P and Q must be <decimal integer>s. Q may be signed.

No conversion or promotion is performed for P or Q. X and Y are converted to X' and Y', where the type, base, and mode of X' and Y' are the common type, base and mode as defined in paragraph 13.2.

If the common type is fixed-point and Q is not given, Q is assumed to be zero. If the common type is floating-point then Q must not be given. Q must be in the range $-128 \leq Q \leq 127$.

If the common base is decimal, let N be 59. If the common base is binary and the common type is fixed-point, let N be 71. If the common base is binary and the common type is floating-point, let N be 63. P must be less than or equal to N.

The result R has the common type, base and mode. The precision of R is (P,Q) or (P).

The value of R is X'-Y'.

13.2.22 Trunc

Example:

```
trunc(X)
```

X must be an arithmetic or string value.

X is converted to X', where the type, base, mode and precision of X' are given in paragraph 13.2. The mode of X' must be real.

The type, base and mode of the result R are the type, base and mode of X'.

If the type of X' is fixed-point, the precision of R is:

```
(min(N,max(p-q+1,1)),0)
```

where N is 71 if X' is binary and N is 59 if X' is decimal, and (p,q) is the precision of X'.

If the type of X' is floating-point, the precision of R is the precision of X'.

If $X' < 0$, the value of R is ceil(X').

If $X' \geq 0$, the value of R is floor(X').

13.3 The Mathematical Built-in Functions

All arguments to mathematical built-in functions must be arithmetic or string values. They are converted to floating-point values as described below:

Let X be the argument and let X' be the converted argument.

If X is a bit-string, X' is a real binary floating-point value of precision 63. If X is a character-string, X' is a real decimal floating-point value of precision 59.

If X is floating-point, X' has the base, mode and precision of X.

If X is fixed-point, X' has the base and mode of X, but its precision is:

min(N,p)

where p is the precision of X, and N is 59 if X' is decimal or N is 63 if X' is binary.

The result R is a floating-point value that has the base, mode and precision of X'.

If the built-in function has two arguments, X and Y, convert them as above to X' and Y'. In addition, if one of the converted arguments is decimal and the other binary, convert the decimal argument to binary. Let p be the precision of X' and r be the precision of Y'; then the result R has the common base and mode of X' and Y' and the precision max(p,r).

The mathematical built-in functions of Multics PL/I are designed to produce accurate results for binary arguments. If the converted argument X' is decimal, it is converted to binary and its precision is set to 63 binary digits. The function is evaluated and the result is converted back to decimal. If the precision of X' was greater than 21, the accuracy of the result will be approximately 20 decimal digits.

Unless the description of a specific function states otherwise, the function can be invoked with scalar or aggregate arguments. When invoked with one or more aggregate arguments, all arguments are promoted to the highest common aggregate type as if they were operands of an infix operator. Refer to Sections 7 and 9.

Each function is described as operating on scalar values and yielding a scalar result. When given aggregate arguments, the function is applied to corresponding scalar components of the promoted aggregate arguments and produces the corresponding scalar component of the aggregate result. The order of evaluation of the scalar components is not defined. The result is an aggregate of the same aggregate type as the promoted aggregate argument.

The table below lists all of the mathematical built-in functions and gives the error conditions and value returned by each function. If one or more of the listed error conditions occurs during the evaluation of one of the functions, the error condition is signalled. Note that other computational conditions may also be signalled as described in Section 10.

When reading the table, let a complex argument X be defined as Y1+i*Y2.

TABLE OF MATHEMATICAL BUILT-IN FUNCTIONS

Function Reference	Argument Mode	Value Returned	Error Conditions
acos(x)	real	arccos in radians 0≤acos(x)≤pi	x>1 x<-1
asin(x)	real	arcsin in radians -(pi/2)≤asin(x)≤(pi/2)	x>1 x<-1
atan(x)	real	arctan(x) in radians -(pi/2)<atan(x)<(pi/2)	
	complex	-i*atanh(i*x)	x=±1i
atan(y,x)	both real	for x>0 arctan(y/x) for y>0 & x=0 pi/2 for y>0 & x<0 pi+arctan(y/x) for y<0 & x=0 -pi/2 for y<0 & x<0 -pi+arctan(y/x)	x=y=0
atand(x)	real	arctan(x) in degrees -90<atand(x)<90	-
atand(y,x)	both real	(180/pi)*atan(y,x)	x=y=0
atanh(x)	real	arctanh(x)	|x|≥1
	complex	(log((1+x)/(1-x)))/2	x=±1
cos(x) x in radians	real	cosine(x)	-
	complex	cosine(y1)*cosineh(y2) -i*sine(y1)*sineh(y2)	-
cosd(x) x in degrees	real	cosine(x)	-
cosh(x)	real	cosineh(x)	-
	complex	cosineh(y1)*cosine(y2) +i*sineh(y1)*sine(y2)	-
erf(x)	real	$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$	-
erfc(x)	real	1 - erf(x)	-
exp(x)	real	e**x	-
	complex	e**x	-
log(x)	real	ln(x)	x≤0
	complex	ln(x) = w where w = u+i*v and -pi<v≤pi	x=0
log10(x)	real	log base 10 of x	x≤0
log2(x)	real	log base 2 of x	x≤0
sin(x) x in radians	real	sine(x)	-
	complex	sine(y1)*cosineh(y2) +i*cosine(y1)*sineh(y2)	-
sind(x) x in degrees	real	sine(x)	-

This page intentionally left blank.

sinh(x)	real	sineh(x)	-
	complex	sineh(y1)*cosine(y2) +i*cosineh(y1)*sine(y2)	-
sqrt(x)	real	$\sqrt{x}$	x<0
	complex	$\sqrt{x} = w$ where w = u+i*v and either u>0, or u=0 and v≥0	-
tan(x) x in radians	real	tangent(x)	-
	complex	tangent(x)	-
tand(x) x in degrees	real	tangent(x)	-
tanh(x)	real	hyperbolic tangent of x	-
	complex	hyperbolic tangent of x	-

13.4 The Array Built-in Functions

13.4.1 Dimension

Example:

```
dimension(X,N)
```

X must be an array value and N must be a scalar arithmetic or string value.

N is converted to N', where N' is a fixed-point, binary, real, value of precision (17,0).

The program is in error if X has less than N' dimensions, or if N' is less than one.

The result R is a fixed-point, binary, real, value of precision (24,0) whose value is the number of elements in the N'th dimension of X.

The dimension built-in function has two names: dimension and dim.

13.4.2 Dot

Example:

```
dot(X,Y,P) or dot(X,Y,P,Q)
```

X and Y must be one dimensional arrays of arithmetic or string values, and P and Q must be <decimal integer>s. Q may be signed.

X and Y are converted to X' and Y', where X' and Y' are the common type, base and mode as defined in paragraph 13.2. The precision of X' and Y' is (P) or (P,Q).

If the common type is fixed-point and Q is not given, Q is assumed to be zero. Q must be in the range $-128 \leq Q \leq 127$. If the common base is decimal, let N be 59. If the common base is binary and the common type is fixed-point, let N be 71. If the common base is binary and the common type is floating-point, let N be 63. P must be less than or equal to N.

The result R is a scalar arithmetic value whose type, base, mode and precision are the type, base, mode and precision of X'.

The value of R is:

$$\sum_{i=m}^{n} X'[i]*Y'[r-m+i]$$

where [m:n] are the bounds of X' and [r:s] are the bounds of Y'. The program is in error if $n-m+1 \neq s-r+1$.

13.4.3 Hbound

Example:

```
hbound(X,N)
```

X must be an array value and N must be a scalar arithmetic or string value.

N is converted to N', where N' is a fixed-point, binary, real, value of precision (24,0).

The program is in error if X has less than N' dimensions, or if N' is less than one.

The result R is a fixed-point, binary, real, value of precision (24,0) whose value is the upper bound of the N'th dimension of X.

13.4.4 Lbound

Example:

```
lbound(X,N)
```

X must be an array value and N must be a scalar arithmetic or string value.

N is converted to N', where N' is a fixed-point, binary, real, value of precision (24,0).

The program is in error if X has less than N' dimensions, or if N' is less than one.

The result R is a fixed-point, binary, real, value of precision (24,0) whose value is the lower bound of the N'th dimension of X.

13.4.5 Prod

Example:

```
prod(X)
```

X must be an array of arithmetic or string values.

X is converted to X' as if it was an operand of the prefix operator "+". If X' is a fixed-point value with precision (p,0) it is converted to a fixed-point value, Y, of the same base and mode, but with precision (N,0), where N is 71 if the base is binary and N is 59 if the base is decimal.

If X' is not a fixed-point value of precision (p,0) it is converted to a floating-point value Y, that has the base and mode of X'. The precision of Y is min(N,p), where N is 59 if X' is decimal or N is 63 if X' if binary.

The result R is an arithmetic scalar whose type, base, mode, and precision are those of Y.

The value of R is:

```
X(1)' * X(2)' * ... * X(n)'
```

13.4.6 Sum

Example:

```
sum(X)
```

X must be an array of arithmetic or string values.

X is converted to X' as if it was an operand of the prefix operator "+".

The result R is an arithmetic scalar value whose type, base and mode are the type, base and mode of X'.

If X' is fixed-point of precision (p,q), the precision of R is (N,q), where N is 71 if the base of R is binary and N is 59 if the base of R is decimal.

If X' is floating-point, the precision of R is the precision of X'.

The value of R is:

X(1)' + X(2)' + ... + X(n)'

13.5 Condition Built-in Functions

The condition built-in functions access values that are set by the signalling of certain conditions. They are best understood if they are considered external controlled variables that are allocated and assigned values by the signalling of a condition.

When one of the conditions that sets the value of a condition built-in function is signalled, the old value of the function is stacked or pushed down until control returns to the point where the signal was made. Control is considered to have returned if the <on unit> entered by the signal returns to the block activation making the signal, or to any of its dynamic predecessors.

The effect of this mechanism is to properly stack the values of these built-in functions. For example, if the conversion condition occurs in an <on unit> entered by a signal of the conversion condition, the values of "onchar" and "onsource" are stacked and the condition is signalled again. On return from the second activation of the <on unit>, the old values of "onchar" and "onsource" are restored and the execution of the first activation of the <on unit> is resumed.

Since the initial value of each of these functions is a null-string, except for "onchar" which is a blank, and "oncode" which is zero, these are the values returned by the functions when they are invoked by a block activation that is not an <on unit> or a dynamic descendent of an <on unit> whose signal set the value.

13.5.1 Onchar

Example:

onchar() or onchar

The value returned by this function is a single character set by the occurrence of the conversion condition as described in paragraph 10.4.2, or is a blank.

13.5.2 Oncode

Example:

oncode() or oncode

The value returned by this function is a fixed-point, binary, real number of precision (17,0). The value indicates the reason why the condition was signalled. Because the run-time subroutines that support the execution of PL/I programs are subject to modification and improvement, the list of error code values is subject to change and is not published in this document. If a program

is expected to run on other implementations of PL/I or on future versions of Multics PL/I, the program logic must not depend on the value returned by this built-in function.

13.5.3 Onfield

Example:

```
onfield() or onfield
```

The value returned by this function is a character-string set by the occurrence of the name condition as described in paragraph 10.4.6, or is a null string.

13.5.4 Onfile

Example:

```
onfile() or onfile
```

The value returned by this function is the filename for which the conversion, name, endfile, transmit, record, key, endpage, or undefinedfile condition was signalled, as described in Section 10, or is a null string.

13.5.5 Onkey

Example:

```
onkey() or onkey
```

The value returned by this function is the character-string key of the record for which the endfile, transmit, record or key condition was signalled, as described in Section 10, or is a null string.

13.5.6 Onloc

Example:

```
onloc() or onloc
```

The value returned by this function is a character-string that identifies the entry point used to enter the most recent <procedure> block activation that is a dynamic predecessor of the most recent <on unit> activation. If no <on unit> activation exists, the function returns a null string.

13.5.7 Onsource

Example:

```
onsource() or onsource
```

The value returned by this function is the value set by the occurrence of the conversion condition as described in paragraph 10.4.2, or is a null string.

13.6 Miscellaneous Built-in Functions

13.6.1 Addr

Example:

```
addr(X)
```

X must be a <reference> to a variable whose storage is connected, as described in paragraph 4.3.1.3.

If X is a <simple reference> that identifies an unallocated, level-one, controlled variable, the result is a null pointer; otherwise, the result is a scalar pointer that identifies the generation of storage referenced by X. In the latter case, the evaluation of X must yield a generation of storage.

13.6.2 Addrel

Example:

```
addrel(X,I)
```

Addrel is a nonstandard built-in function and its use makes programs dependent on the data representation of Multics PL/I.

X must be a scalar pointer value and I is converted to I'. If I is a bit-string, I' is a scalar bit-string of length 18; otherwise I' is a scalar fixed-point, binary, real value of precision (18,0).

The result R is a scalar pointer value whose ring number and segment number are the ring and segment numbers of X and whose word offset is given by the sum of the word offset of X and the value of I. The bit offset of R is zero.

13.6.3 Allocation

Example:

```
allocation(X)
```

X must be a <reference> to a level-one controlled variable.

The result R is a scalar, binary, fixed-point, real number of precision (17,0).

The value of R is the number of generations of X currently allocated. If no generations are allocated, the value of R is zero.

The allocation built-in function has two names: allocation and allocn.

13.6.4 Baseno

Example:

```
baseno(X)
```

Baseno is a nonstandard built-in function and its use makes programs dependent on the representation of pointer values in Multics PL/I.

X must be a scalar pointer value.

The result R is a bit-string of length 18 whose value is the bit-string representation of the segment number part of X.

13.6.5 Baseptr

Example:

```
baseptr(I)
```

Baseptr is a nonstandard built-in function and its use makes programs dependent on the representation of pointer values in Multics PL/I.

I is converted to I'. If I is a bit-string, I' is a scalar bit-string of length 18; otherwise I' is a scalar, fixed-point, binary, real value of precision (18,0).

The result R is a scalar pointer value whose ring number is the current ring, whose segment number is I, and whose offsets are zero.

13.6.5a Clock

Example:

```
clock or clock()
```

Clock is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

The result R is a fixed-point, binary, real value of precision (71,0).

The value of R is the number of microseconds since 0000 hours January 1, 1901, Greenwich mean time.

13.6.5b Codeptr

Example:

```
codeptr(X)
```

Codeptr is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

X must be an entry, label, or format value. The result R is a pointer value. If X is an entry value, then R is a pointer to the entry point identified by X. If X is a label value, then R is a pointer to the <statement> identified by X. If X is a format value, then R is a pointer to the <format statement> identified by X.

13.6.6 Convert

Example:

```
convert(X,Y)
```

Convert is a nonstandard built-in function.

X must be a <reference> to a scalar variable and Y must be a scalar value.

Y is converted to Y', where the data type of Y' is the data type of X, and the value of Y' is the value of Y converted according to the rules for conversion given in Section 8.

The result R has the data type and value of Y'.

13.6.6a Currentsize

Example:

```
currentsize(X)
```

Currentsize is a nonstandard built-in function and its use makes programs dependent on the internal representation of data in Multics PL/I.

X must be an unsubscripted <reference> to a level-one variable.

The result is a fixed-point, binary, real number of precision (19,0) whose value is the number of 36-bit words occupied by the generation of storage obtained by evaluating the reference X. Note that when X is a reference to a based variable with <refer option>s, this function returns a value that depends on the <reference> contained in the <refer option>, not on the <expression> in the <extent expression>.

13.6.7 Date

Example:

```
date() or date
```

The result R is a character-string of length 6.

The value of R is:

```
YYMMDD
```

where YY is the year, MM is the month, and DD is the day.

13.6.8 Empty

Example:

```
empty() or empty
```

The result R is an empty area value.

13.6.8a Environmentptr

Example:

```
environmentptr(X)
```

Environmentptr is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

X must be an entry, label, or format value. The result R is the activation record pointer of X.

13.6.9 Lineno

Example:

```
lineno(X)
```

X must be a scalar file value.

If X does not identify an open file-state block with the <print attribute>, the program is in error.

The result R is a scalar, fixed-point, binary, real number of precision (35,0).

The value of R is the linenumber of the file-state block identified by X.

13.6.10 Null

Example:

```
null() or null
```

The result is a null pointer value.

13.6.11 Nullo

Example:

```
nullo() or nullo
```

Nullo is a nonstandard built-in function.

The result is a null offset value.

13.6.12 Offset

Example:

```
offset(X,Y)
```

X and Y must be scalar values. X must be a pointer value and Y must be an area value.

Unless X identifies a generation of storage within Y, the program is in error.

The result R is an offset value that identifies the generation of storage identified by X.

13.6.13 Pageno

Example:

```
pageno(X)
```

X must be a scalar file value.

If X does not identify an open file-state block with the <print attribute> the program is in error.

The result R is a scalar, fixed-point, binary, real number of precision (35,0).

The value of R is the pagenumber of the file-state block identified by X.

13.6.14 Pointer

Example:

```
pointer(X,Y)
```

The pointer built-in function is a generic function with two entirely different meanings that depend on the data types of X and Y.

13.6.14.1 The Standard Definition of Pointer

X must be a scalar offset value and Y must be a scalar area value.

Unless X identifies a generation of storage within Y, the program is in error.

The result R is a pointer value that identifies the generation of storage identified by X.

13.6.14.2 The Nonstandard Definition of Pointer

The use of the nonstandard definition of the pointer function makes programs dependent on the representation of pointer values in Multics PL/I.

X must be a scalar pointer value and Y is converted to Y'. If Y is a bit-string, Y' is a scalar bit-string of length 18; otherwise, Y' is a scalar, fixed-point, binary, real value of precision (18,0).

The result R is a pointer value whose ring number and segment number are the ring and segment numbers of X and whose word offset is given by Y. The bit offset is zero.

The program is in error if X and Y do not satisfy the argument constraints of one of the two definitions of the function.

The pointer built-in function has two names: pointer and ptr.

13.6.15 Rel

Example:

```
rel(X)
```

Rel is a nonstandard built-in function and its use makes programs depend on the representation of pointer values in Multics PL/I.

X must be a scalar pointer value.

The result R is a bit-string of length 18 whose value is the word offset portion of X.

13.6.16 Size

Example:

```
size(X)
```

Size is a nonstandard built-in function and its use makes programs depend on the internal representation of data in Multics PL/I.

X must be a <simple reference> to a level-one variable.

The result is a fixed-point, binary, real number of precision (24,0) whose value is the number of 36 bit words necessary to allocate a generation of storage for X. Note that when X is a based variable with <refer option>s, this function returns a value that depends on the <expression> contained in the <extent expression>, not on the <reference> contained in the <refer option>.

13.6.17 Stac

Example:

```
stac(X,Y)
```

Stac is a nonstandard built-in function and its use makes programs depend on the Multics hardware. Coordination of Multics processes should be done by calls to Multics locking primitives as described in the "Multics Programmers' Manual".

X must be a scalar pointer value and Y must be a scalar bit-string of length 36.

If the 36 bit word addressed by the pointer is zero, the value of Y is assigned to that word; otherwise, no assignment is made.

The result R is a bit-string of length 1.

If the assignment of Y to the location identified by X was made, the value of R is "1"b; otherwise, it is "0"b.

The testing of X and the assignment of Y to X is an indivisible operation of the Multics hardware.

13.6.17a Stacq

Example:

```
stacq(L,A,Q)
```

Stacq is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

L must be a <reference> to an aligned scalar bit-string variable of length 36. A and Q must be bit-strings of length less than or equal to 36. The result R is a bit-string of length 1.

If L equals Q, the value of A is assigned to L, and the value of R is "1"b; otherwise, no assignment is made and the value of R is "0"b.

The testing for equality between L and Q and the conditional assignment of A to L is an indivisible operation of the Multics hardware; refer to the description of the stacq instruction in the Multics Processor Manual, Order No. AL39.

13.6.17b Stackbaseptr

Example:

```
stackbaseptr() or stackbaseptr
```

Stackbaseptr is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

Stackbaseptr returns a pointer to the base of the current <block>'s stack segment.

13.6.17c Stackframeptr

Example:

```
stackframeptr() or stackframeptr
```

Stackframeptr is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

Stackframeptr returns a pointer to the stack frame containing the activation record of the current <block>.

13.6.18 Time

Example:

```
time() or time
```

The value returned by the function is a character-string of length 12 whose value is:

```
HHMMSSFFFFFF
```

where HH is the hour, 00 to 23; MM is the minute, 00 to 59; SS is the second, 00 to 59; and FFFFFF is the microsecond, 000000 to 999999.

13.6.19 Unspec

Example:

```
unspec(X)
```

X must be a <reference> to a variable.

The result R is a bit-string whose length and value depend on the data type, aggregate type, and value of X. The value of R is the internal representation of X.

13.6.20 Valid

Example:

```
valid(X)
```

X must be a <reference> to a scalar pictured value.

The result R is a bit-string of length 1. Its value is "1"b if the character-string value of X can be edited into the <picture> declared for X; otherwise, the value of R is "0"b.

13.6.20a Vclock

Example:

```
vclock or vclock()
```

Vclock is a nonstandard built-in function and its use makes programs dependent on Multics PL/I.

The result R is a fixed-point, binary, real value of precision (71,0).

The value of R is the number of microseconds of virtual CPU time used by the calling process.

APPENDIX A

Differences Between Multics PL/I and Standard PL/I

This appendix lists all known deviations of the Multics PL/I language from the American National Standard Programming Language PL/I, ANSI X3.53-1976, as of March, 1981.

The features that are marked with a + are not detected by the -check_ansi control argument of the pl1 command.

Features of the Standard Not in Multics PL/I:

1. The tab option and tab format item.
2. The "t", "i", and "r" picture characters.
3. The every and some built-in functions.

Features Restricted in Multics PL/I:

1. Only one <prefix subscript> is permitted in a <label prefix>.
2. The <condition name>s defined by the language are reserved such that a user-defined condition cannot have the same name as a language defined condition.
3. A <condition name> cannot have internal scope.
4. The <extent>s of static variables must be <decimal integer>s, and the <expression>s in the <initial attribute> of a static variable are restricted to optionally signed <literal constant>s, pairs of real and imaginary signed <literal constant>s, or the null and empty built-in functions.
5. The <label prefix> of a <procedure statement>, <entry statement>, or <format statement> cannot contain a <prefix subscript>.
6. The string built-in function requires that its argument be a scalar, or an aggregate of packed bit-string or packed character-string data.
7. The alignment attributes of two structures must match if the two structures are to share storage.
8. All <condition prefix>s of a statement must precede any <label prefix>s of the statement.
9. An area variable cannot be used as the <index> of a <do statement>.

10. Defined variables whose <defined attribute> contains <isub>s or asterisks cannot be input or output by a <get statement> or <put statement> that specifies data-directed transmission.

11. File constants cannot have the <dimension attribute>.

12. If the <expression> of an <assignment statement> is a <reference> that identifies a scalar string variable, then no <target> of the <assignment statement> can identify a generation of storage that overlaps the generation of storage of the string variable, unless it is exactly the same generation or unless the generation of the <target> does not start to the right of the generation of the string variable.

13. An unconnected array cannot be passed to an array parameter declared with constant extents; asterisk extents must be used.

14. When an array is defined onto another array by simple defining, the <base reference> must contain an asterisk for each dimension of the defined array.

15. The pointer value yielded by "addr" of a parameter is valid only so long as the block activation to which the corresponding argument was passed is still active.

16. The standard allows an array of scalars to be promoted to an array of structures, but Multics PL/I does not allow this promotion.

17. A simple or isub defined variable must have extents that equal the corresponding extents of the base variable on which it is defined. The standard allows such extents to be less than or equal to the extents of the base variable.

18. In structure promotion of the form s=r or s+r, Multics PL/I requires that the aggregate type of each member of s match the aggregate type of the corresponding member of r. The standard performs aggregate promotion for each member that does not match.

19. The dot built-in function requires that the precision of its result be given in the function reference.

20. Both the <ignore option> and a <key spec> cannot be given in the same <read statement>.

21. If a completed <attribute set> contains a <position attribute>, that <position attribute> must contain a <position>. The standard has a system default of 1.

22. The min and max built-in functions must have at least two arguments; the standard allows them to have one argument.

23. If an item has the <parameter attribute> or is part of a <descriptor>, the <extent expression> must be an unsigned <decimal integer>.

Features Implemented at Variance with the Standard:

+ 1. The <bound>s of an evaluated array expression are always normalized such that each lower <bound> is one and each upper <bound> is the number of elements in the dimension.

+ 2. A mismatch between the alignment attributes of a structure and a structure parameter descriptor causes the argument to be passed by-value, rather than by-reference. The standard ignores the alignment attributes of structures.

+ 3. The stringsize condition is disabled by default in Multics PL/I, but enabled by default in standard PL/I.

Extensions:

1. An <identifier> can contain the special character "$", and in the case of external names, this character has additional semantics.

2. Varying strings can be used in simple or isub defining.

3. The base variable identified by a <defined attribute> can be a based variable.

+ 4. Most restrictions on the <refer option> are removed.

5. Several new built-in functions are implemented.

6. The <local attribute> is allowed in all <descriptor>s.

+ 7. If the current position of a file is well defined, a <key option> is not needed in a <delete statement> or <rewrite statement> operating on a direct file.

+ 8. New records can be written into a keyed sequential update file. (The locate statement may be used for this purpose.)

+ 9. Partially qualified references are allowed in stream input scanned by data-directed input.

+ 10. An <in option> is not required in a <free statement> when freeing a generation of storage allocated in an area.

+ 11. The "recursive" keyword is never required in a <procedure statement>.

12. The <unspec pseudo> and unspec built-in function allow aggregate arguments.

+ 13. Assignments and infix operations can be performed on two arrays of unequal <bound>s if the number of dimensions is equal and the number of elements in each dimension of one array is equal to the number of elements in the corresponding dimension of the other array.

+ 14. A replication factor in a <picture> can be zero, indicating that the <picture char> to which it applies is to be deleted from the <normal picture> produced by translation of the <picture>.

15. A name declared with the <environment attribute> will acquire the <file attribute> by application of the language default rules. A name declared with the <options attribute> will acquire the <entry attribute> by application of the language default rules unless the parenthesized keyword "constant" is specified. The standard gives no defaults for these cases.

+ 16. Multics PL/I allows a <column format> to be used by a <get statement> or <put statement> containing a <string option>.

+ 17. When an array is passed as an argument to an array parameter which has different <bound>s but equal extents, the standard says that the program is in error. Multics PL/I assigns the argument to an array temporary whose <bound>s are equal to the <bound>s of the array parameter.

+ 18. A <picture scale factor> is allowed for floating-point <picture>s.

19. The <reducible attribute> and <irreducible attribute>s are allowed.

+ 20. No delimiter is required between the keywords "picture" or "pic" and the quoted <picture> in a <picture attribute>. No delimiter is required between the letter "p" and the quoted <picture> in a <picture format>.

21. Any data type except area is allowed in put list and put data.

+ 22. ASCII tab characters in an input data stream get special treatment.

23. The <options attribute> with the parenthesized keyword "constant" specified may be used with any computational variable containing the <static attribute> and the <internal attribute>.

24. <default statement>s may appear in any block.

+ 25. A <returns attribute> of the form returns () is permitted. (Of course <returns descriptor>s must then be supplied during default processing.)

26. The pointer built-in function may take a pointer as its first argument and any computational expression as its second argument.

27. The fixed and float built-in functions may take as few as one argument. (The standard requires two arguments.)

28. The <read statement> and the <write statement> may be used with stream data sets to read and write a line, respectively.

29. The <unsigned attribute> and <signed attribute>s are allowed.

+ 30. A <programmer-defined condition name> may be an <identifier>.

+ 31. ASCII newline characters, horizontal tab characters, vertical tab characters, and newpage characters are delimiters.

+ 32. The <member attribute>, <structure attribute>, and <parameter attributes> are allowed in the <attribute set> of a <default statement>.

This index contains every <notation variable> defined by the syntax rules. It also contains every underlined term defined in prose, as well as a few general terms not defined in prose. For each <notation variable> the only sections listed in the index are those in which the <notation variable> is defined. Note that definitions are sometimes repeated to aid understanding and reduce the number of cross-references.

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS PL/I LANGUAGE SPECIFICATION (INCLUDES ADDENDA A,B,C,D AND E)

ORDER NO. AG94-02

DATED JULY 1976

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho, Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

42039, 1C185, Printed in U.S.A.

AG94-02