Enterprise PL/I for z/OS

IBM

Language Reference

Version 3 Release 3

> **Note!**
>
> Before using this information and the product it supports, be sure to read the general information under "Notices" on page 539.

Contents

Tables

Figures

Chapter 1. About this book

This book is a reference for the programmer using the IBM PL/I compiler. It is not a tutorial, but is designed for the reader who already has a knowledge of the PL/I language and who requires reference information needed to write a program for an IBM PL/I compiler. It contains guidance information and general-use programming interfaces.

Because this book is a reference manual, it is not intended to be read from front to back, and terms can be used before they are defined. Terms are highlighted where they are defined in the book, and definitions are found in the glossary.

WRKSTN⏩

Text set apart by the workstation opening and closing icons designates features which are supported only on PL/I workstation products (AIX, OS/2, and Windows).

◁🖳

Notation conventions used in this book

The following sections describe how information is presented in this book. Examples and user-supplied information are presented in mixed-case characters. The following rules apply to the syntax diagrams used in this book:

Arrow symbols

Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

▶▶── Indicates the beginning of a statement.

──▶ Indicates that the statement syntax is continued on the next line.

▶── Indicates that a statement is continued from the previous line.

──▶◀ Indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the ▶── symbol and end with the ──▶ symbol.

Conventions

- Keywords, their allowable synonyms, and reserved parameters, appear in uppercase for MVS and OS/2® platforms, and lowercase for UNIX® platforms. These items must be entered exactly as shown.
- Variables appear in lowercase italics (for example, *column-name*). They represent user-defined parameters or suboptions.
- When entering commands, separate parameters and keywords by at least one blank if there is no intervening punctuation.
- Enter punctuation marks (slashes, commas, periods, parentheses, quotation marks, equal signs) and numbers exactly as given.

1

- Footnotes are shown by a number in parentheses, for example, (1).
- A ƀ symbol indicates one blank position.

Required items

Required items appear on the horizontal line (the main path).

```
>>--REQUIRED_ITEM---------------------------------------------------><
```

Optional Items

Optional items appear below the main path.

```
>>--REQUIRED_ITEM--------------------------------------------------><
                 |_optional_item_|
```
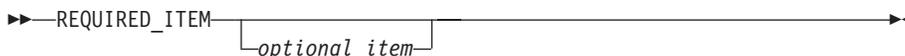
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

```
                  _optional_item_
>>--REQUIRED_ITEM--|            |----------------------------------><
```
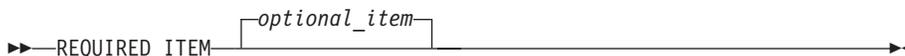
Multiple required or optional items

If you can choose from two or more items, they appear vertically in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

```
>>--REQUIRED_ITEM---required_choice1----------------------------><
                  |_required_choice2_|
```
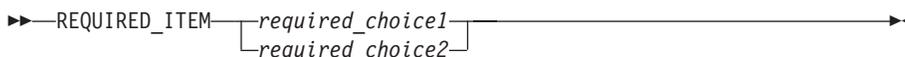
If choosing one of the items is optional, the entire stack appears below the main path.

```
>>--REQUIRED_ITEM----------------------------------------------><
                 |_optional_choice1_|
                 |_optional_choice2_|
```
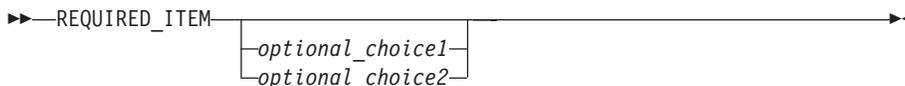
Repeatable items

An arrow returning to the left above the main line indicates that an item can be repeated.

```
                  <--------------
>>--REQUIRED_ITEM---repeatable_item---------------------------><
```
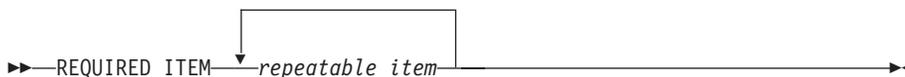
If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                  <----,---------
>>--REQUIRED_ITEM---repeatable_item---------------------------><
```
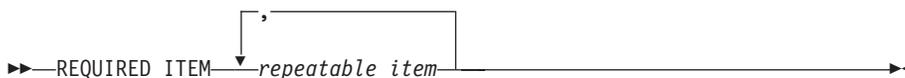
A repeat arrow above a stack indicates that you can specify more than one of the choices in the stack.

Default keywords

IBM-supplied default keywords appear above the main path, and the

remaining choices are shown below the main path. In the parameter list
following the syntax diagram, the default choices are underlined.

```
                          ┌─default_choice─┐
►►──REQUIRED_ITEM─────────┼────────────────┼──────────────────────────────►◄
                          ├─optional_choice─┤
                          └─optional_choice─┘
```
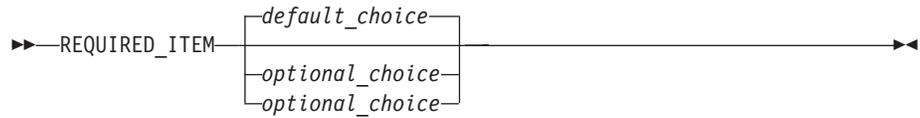
Fragments

Sometimes a diagram must be split into fragments. The fragments are
represented by a letter or fragment name, set off like this: | A |. The
fragment follows the end of the main diagram. The following example shows
the use of a fragment.

```
►►──STATEMENT──item 1──item 2──┤ A ├─────────────────────────────────────►◄
```

A:

```
├──┬─item 3─┬──KEYWORD──┬─────────┬──────────────────────────────────────┤
   └─item 4─┘           ├─item 5─┤
                        └─item 6─┘
```
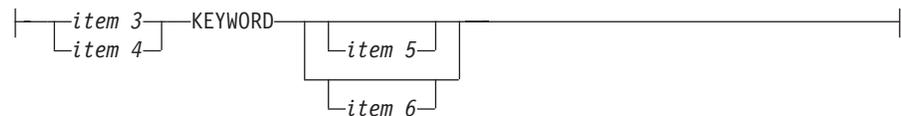
Substitution-block

Sometimes a set of several parameters is represented by a
substitution-block such as **<A>**. For example, in the imaginary /VERB
command you could enter /VERB LINE 1, /VERB EITHER LINE 1, or /VERB
OR LINE 1.

```
►►──/VERB──┬──────┬──LINE──line#─────────────────────────────────────────►◄
           └─<A>─┘
```

where <A> is:

```
►►──┬─EITHER─┬────────────────────────────────────────────────────────────►◄
    └─OR─────┘
```

Parameter endings

Parameters with number values end with the symbol '#', parameters that
are names end with 'name', and parameters that can be generic end with
'*'.

```
►►──/MSVERIFY──┬─MSNAME──msname─┬─────────────────────────────────────────►◄
               └─SYSID──sysid#──┘
```

The MSNAME keyword in the example supports a name value and the
SYSID keyword supports a number value.

Semantics

To describe the PL/I language, the following conventions are used:

- The descriptions are informal. For example, we usually write "x must be a
  variable" instead of the more precise "x must be the name of a variable".
  Similarly, we can sometimes write "x is transmitted" instead of "the value of x is

transmitted". When the syntax indicates "reference", we can later write "the variable" instead of "the referenced variable".

- When we say that two different source constructs are equivalent, we mean that they produce the same result, and not necessarily that the implementation is the same.
- Unless specifically stated in the text following the syntax specification, the unqualified term "expression" or "reference" refers to a scalar expression. For an expression other than a scalar expression, the type of expression is noted. For example, the term "array expression" indicates that neither a scalar expression nor a structure expression is valid.
- When a result or behavior is *undefined*, it is something you "must not" do. Use of an undefined feature is likely to produce different results on different implementations or releases of a PL/I product. The application program is considered to be in error.
- *Default* is used to describe an alternative value, attribute, or option that is assumed by the system when no explicit choice is specified.
- *Implicit* is used to describe the action taken in the absence of an explicit specification by the program.
- The lowercase letter b, when not in a word, indicates a blank character.

Industry standards used

The PL/I compiler is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of December 1987:

- American National Standard Code for Information Interchange (ASCII), X3.4 - 1977
- American National Standard Representation of Pocket Select Characters in Information Interchange, level 1, X3.77 - 1980 (proposed to ISO, March 1, 1979)
- The draft proposed American National Standard Representation of Vertical Carriage Positioning Characters in Information Interchange, level 1, dpANS X3.78 (also proposed to ISO, March 1, 1979)
- Selected features of the American National Standard PL/I General Purpose Subset (ANSI X3.74-1987).

Enhancements in this release

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Improved performance

- The compiler now handles even more conversions by generating inline code which means these conversions will be done much faster than previously. Also, all conversions done by library call are now flagged by the compiler.
- The compiler-generated code now uses, in various situations, less stack storage.
- The compiler now generates much better code for references to the TRANSLATE built-in function.
- The compiler-generated code for SUBSCRIPTRANGE checking is now, for arrays with known bounds, twice as fast as before.
- The ARCH and TUNE options now support 4 as a suboption, thereby allowing exploitation of instructions new to the zSeries machines.
- ARCH(2), FLOAT(AFP) and TUNE(3) are now the default.

Easier migration

- Compiler defaults have been changed for easier migration and compatibility. The changed defaults are:
  - CSECT
  - CMPAT(V2)
  - LIMITS(EXTNAME(7))
  - NORENT
- The compiler now honors the NOMAP, NOMAPIN and NOMAP attributes for PROCs and ENTRYs with OPTIONS(COBOL).
- The compiler now supports PROCs with ENTRY statements that have differing RETURNS attribute in the same manner as did the old host compiler.
- The compiler will now assume OPTIONS(RETCODE) for PROCs and ENTRYs with OPTIONS(COBOL).
- The SIZE condition is no longer promoted to ERROR if unhandled.
- Various changes have been made to reduce compile time and storage requirements.
- The OFFSET option will now produce a statement offset table much like the ones it produced under the older PL/I compilers.
- The FLAG option now has exactly the same meaning as it had under the old compilers, while the new MAXMSG option lets you decide if the compiler should terminate after a specified number of messages of a given severity. For example, with FLAG(I) MAXMSG(E,10), you can now ask to see all I-level messages while terminating the compilation after 10 E-level messages.
- The AGGREGATE listing now includes structures with adjustable extents.
- The STMT option is now supported for some sections of the listing.
- The maximum value allowed for LINESIZE has been changed to 32759 for F-format files and to 32751 for V-format files.

Improved usability

- The defaults for compiler options may now be changed at installation.
- The integrated SQL preprocessor now supports DB2 Unicode.
- The compiler now generates information that allows Debug Tool to support Auto Monitor, whereby immediately before each statement is executed, all the values of all the variables used in the statement are displayed.
- The new NOWRITABLE compiler option lets you specify that even under NORENT and at the expense of optimal performance, the compiler should use no writable static when generating code to handle FILEs and CONTROLLED.
- The new USAGE compiler option gives you full control over the IBM or ANS behavior of the ROUND and UNSPEC built-in function without the other effects of the RULES(IBM|ANS) option.
- The new STDSYS compiler option lets you specify that the compiler should cause the SYSPRINT file to be equated to the C stdout file.
- The new COMPACT compiler option lets you direct the compiler to favour those optimizations which tend to limit the growth of the code.
- The LRECL for SYSPRINT has been changed to 137 to match that of the C/C++ compiler.
- POINTERs are now allowed in PUT LIST and PUT EDIT statements: the 8-byte hex value will be output.
- If specified on a STATIC variable, the ABNORMAL attribute will cause that variable to be retained even if unused.

Enhancements in recent releases

This release also provides all of the functional enhancements offered in Enterprise PL/I V3R1, including the following:

- Support for Multithreading on z/OS
- Support for IEEE floating-point on z/OS
- Support for the ANSWER statement in the macro prepreprocessor
- SAX-style XML parsing via the PLISAXA and PLISAXB built-in subroutines
- Additional built-in functions:
  - CS
  - CDS
  - ISMAIN
  - LOWERCASE
  - UPPERCASE

This release also provides all of the functional enhancements offered in VisualAge PL/I V2R2, including the following:

- Initial UTF-16 support via the WIDECHAR attribute

  There is currently no support yet for

  - WIDECHAR characters in source files
  - W string constants
  - use of WIDECHAR expressions in stream I/O
  - implicit conversion to/from WIDECHAR in record I/O
  - implicit endianness flags in record I/O

    If you create a WIDECHAR file, you should write the endianness flag ('fe_ff'wx) as the first two bytes of the file.
- DESCRIPTORS and VALUE options supported in DEFAULT statements
- PUT DATA enhancements
  - POINTER, OFFSET and other non-computational variables supported
  - Type-3 DO specifications allowed
  - Subscripts allowed
- DEFINE statement enhancements
  - Unspecified structure definitions
  - CAST and RESPEC type functions
- Additional built-in functions:
  - ACOSF
  - ASINF
  - ATANF
  - CHARVAL
  - COSF
  - EXPF
  - ISIGNED
  - IUNSIGNED
  - LOG10F
  - LOGF
  - ONWCHAR
  - ONWSOURCE
  - SINF
  - TANF
  - WCHAR
  - WCHARVAL
  - WHIGH

- – WIDECHAR
- – WLOW
- Preprocessor enhancements
  - – Support for arrays in preprocessor procedures
  - – WHILE, UNTIL and LOOP keywords supported in %DO statements
  - – %ITERATE statement supported
  - – %LEAVE statement supported
  - – %REPLACE statement supported
  - – %SELECT statement supported
  - – Additional built-in functions:
    - - COLLATE
    - - COMMENT
    - - COMPILEDATE
    - - COMPILETIME
    - - COPY
    - - COUNTER
    - - DIMENSION
    - - HBOUND
    - - INDEX
    - - LBOUND
    - - LENGTH
    - - MACCOL
    - - MACLMAR
    - - MACRMAR
    - - MAX
    - - MIN
    - - PARMSET
    - - QUOTE
    - - REPEAT
    - - SUBSTR
    - - SYSPARM
    - - SYSTEM
    - - SYSVERSION
    - - TRANSLATE
    - - VERIFY

About this book

Chapter 2. Program elements

This chapter describes the basic elements that are used to write a PL/I program. The elements include character sets, programmer-defined identifiers, keywords, delimiters, and statements.

PL/I supports a single-byte character set (SBCS) and a double-byte character set (DBCS).

The implementation limits for PL/I's language elements are listed in "Limits," on page 533.

Single-byte character set

A *character set* is an ordered set of unique representations called *characters*; for example, the set of symbols in Morse code, or the letters of the Cyrillic alphabet are character sets. PL/I supports all PC character sets. Character set 0640 is called the *invariant* character set because a character from this set has the same code point in all code pages. A *code point* is a one-byte code representing one of 256 potential characters; a *code page* is an assignment of graphic characters and control function meanings to all of the code points.

PL/I supports all code pages that conform to character set 0640; however, PL/I assumes code page 0850 for all code points, except for those characters which are specified by the programmer using the CURRENCY, NAMES, OR, or NOT compiler options. For more information on these options, refer to the Programming Guide.

Code page 0850 contains the English alphabet, ten decimal digits, special characters, and other national language and control characters. Constants and comments can contain any SBCS character value. PL/I elements (for example, statements, keywords and delimiters) are limited to the characters described in the following sections.

Alphabetic and extralingual characters

The default alphabet for PL/I is the English alphabet plus the extralingual characters.

Alphabetic characters

There are 26 base alphabetic characters that comprise the English alphabet. They are shown in Table 1 with the equivalent ASCII and EBCDIC values in hexadecimal notation.

Table 1. Alphabetic equivalents

Character	EBCDIC Uppercase Hex Value	EBCDIC Lowercase Hex Value	ASCII Uppercase Hex Value	ASCII Lowercase Hex Value
A	C1	81	41	61
B	C2	82	42	62
C	C3	83	43	63
D	C4	84	44	64
E	C5	85	45	65
F	C6	86	46	66
G	C7	87	47	67
H	C8	88	48	68
I	C9	89	49	69
J	D1	91	4A	6A
K	D2	92	4B	6B
L	D3	93	4C	6C
M	D4	94	4D	6D
N	D5	95	4E	6E
O	D6	96	4F	6F
P	D7	97	50	70
Q	D8	98	51	71
R	D9	99	52	72
S	E2	A2	53	73
T	E3	A3	54	74
U	E4	A4	55	75
V	E5	A5	56	76
W	E6	A6	57	77
X	E7	A7	58	78
Y	E8	A8	59	79
Z	E9	A9	5A	7A

Extralingual characters

The default extralingual characters are the *number* sign (#), the *at* sign (@), and the *currency* sign ($). The hexadecimal values for these characters vary across code pages. You can use the NAMES compiler option to define your own extralingual characters. For more information on defining extralingual characters, refer to the Programming Guide.

Alphanumeric characters

An *alphanumeric* character is either an alphabetic or extralingual character, or a digit.

Decimal digits

PL/I recognizes the ten decimal digits, 0 through 9. They are also known simply as digits and are used to write decimal constants and other representations and values. The following table shows the digits and their hexadecimal values.

Table 2. Decimal digit equivalents

Character	EBCDIC Hex Value	ASCII Hex Value
0	F0	30
1	F1	31
2	F2	32
3	F3	33
4	F4	34
5	F5	35
6	F6	36
7	F7	37
8	F8	38
9	F9	39

Binary digits

PL/I recognizes the two binary digits, 0 and 1. They are also known as bits and are used to write binary and bit constants.

Hexadecimal digits

PL/I recognizes the 16 hexadecimal digits, 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively. They are also known as hex digits or just hex and are used to write constants in hexadecimal notation.

Special characters

Table 3 shows the special characters, their PL/I meanings, and their ASCII and EBCDIC values in hexadecimal notation.

Table 3. Special character equivalents

Character	Meaning	Default EBCDIC Hex Value	Default ASCII Hex Value
b	Blank	40	20
=	Equal sign or assignment symbol	7E	3D
+	Plus sign	4E	2B
–	Minus sign	60	2D
*	Asterisk or multiply symbol	5C	2A
/	Slash or divide symbol	61	2F
(	Left parenthesis	4D	28
)	Right parenthesis	5D	29
,	Comma	6B	2C
.	Point or period	4B	2E
'	Single quotation mark	7D	27
"	Double quotation mark	7F	22
%	Percent	6C	25
;	Semicolon	5E	3B
:	Colon	7A	3A

Special characters

Table 3. Special character equivalents (continued)

Character	Meaning	Default EBCDIC Hex Value	Default ASCII Hex Value
¬	Not symbol, exclusive-or symbol [Note 1]	5F	5E
&	And symbol	50	26
|	Or symbol [(Note 1)]	4F	7C
>	Greater than symbol	6E	3E
<	Less than symbol	4C	3C
_	Break character (underscore)	6D	5F
?	Macro trigger character	6F	3F

Note 1:

The or (|) and the not (¬) symbols have variant code points. You can use the compiler options OR and NOT to define alternate symbols to represent these operators. For more information about these options, refer to the Programming Guide.

Composite symbols

You can combine special characters to create composite symbols. The following table describes these symbols and their meanings. Composite symbols cannot contain blanks.

Table 4. Composite symbol description

Composite Symbol	Meaning
||	Concatenation
**	Exponentiation
¬<	Not less than
¬>	Not greater than
¬=	Not equal to; Evaluate, exclusive-or and assign
<=	Less than or equal to
>=	Greater than or equal to
/*	Start of a comment
*/	End of a comment
–>	Locator (pointers and offsets)
=>	Locator (handles)
+=	Evaluate expression, add and assign
–=	Evaluate expression, subtract and assign
*=	Evaluate expression, multiply and assign
/=	Evaluate expression, divide and assign
|=	Evaluate expression, or and assign
&=	Evaluate expression, and, and assign
||=	Evaluate expression, concatenate and assign
**=	Evaluate expression, exponentiate and assign

Case sensitivity

You can use a combination of lowercase and uppercase characters in a PL/I program.

When used in keywords or identifiers, the lowercase characters are treated as the corresponding uppercase characters. This is true even if you entered a lowercase character as a DBCS character.

When used in a comment or in a character, mixed, or a graphic string constant, lowercase characters remain lowercase.

Statement elements for SBCS

This section describes the elements that make up a PL/I program when using SBCS.

A PL/I statement consists of identifiers, delimiters, operators, and constants. Constants are described in Chapter 3, "Data elements," on page 21.

Identifiers

An *identifier* is a series of characters that are not contained in a comment or a constant. Except for P, PIC, and PICTURE, identifiers must be preceded and followed by a delimiter. (P, PIC, and PICTURE identifiers can be followed by a character string.) The first character of an identifier must be an alphabetic or extralingual character. If the identifier names an INTERNAL symbol, it may also use the break (_) character as its first character. Other characters, if any, can be alphabetic, extralingual, digit, or the break (_) character. External user names must not start with IBM, PLI, CEE, _IBM, _PLI, and _CEE.

Identifiers can be PL/I keywords or programmer-defined names. Because PL/I can determine from the context if an identifier is a keyword, you can use any identifier as a programmer-defined name. There are no *reserved* words in PL/I.

PL/I keywords
A *keyword* is an identifier that has a specific meaning in PL/I. Keywords can specify such things as the action to be taken or the attributes of data. For example, READ, DECIMAL, and ENDFILE are keywords. Some keywords can be abbreviated. The keywords and their abbreviations are shown in uppercase letters.

Programmer-defined names
In a PL/I program, *names* are given to variables and program-control data. There are also built-in names, condition names, and generic names. Any identifier can be used as a name. A name can have only one meaning in a program block; the same name cannot be used for both a file and a floating-point variable in the same block.

To improve readability, the break character (_) can be used in a name, such as Gross_Pay.

Examples of names are:

```
 A               Rate_of_pay

 Record          Loop_3
```

Additional requirements for programmer-defined external names are given in "INTERNAL and EXTERNAL attributes" on page 153.

An asterisk (*) can be used as an identifier name in situations where a name is required but you do not otherwise refer to that identifier. For an example, see page 115.

Delimiters and operators

Delimiters and *operators* are used to separate identifiers and constants. Table 5 shows delimiters and Table 6 shows operators.

Table 5. Delimiters

Name	Delimiter	Use
Comment	/* */	The /* and */ enclose commentary (the comment includes the /* and the */ and any characters between them)
Comma	,	Separates elements of a list; precedes the BY NAME option
Period	.	Connects elements of a qualified name; decimal or binary point
Semicolon	;	Terminates a statement
Equal sign	=	Indicates assignment or, in a conditional expression, equality
Colon	:	Connects prefixes to statements; connects lower-bound to upper-bound in a dimension attribute; used in RANGE specification of DEFAULT statement
Blank	b	Separates elements
Parentheses	( )	Enclose lists, expressions, iteration factors, and repetition factors; enclose information associated with various keywords
Locator	–> =>	Denotes locator qualification (pointers and offsets) Denotes locator qualification (handles)
Percent	%	Indicates %statements and %directives
Single quote	'	Encloses constants (indicates the beginning and end of a constant)
Double quote	"	Encloses constants (indicates the beginning and end of a constant)

Note: Omitting certain symbols can cause errors that are difficult to trace. Common errors are unbalanced quotes, unmatched parentheses, unmatched comment delimiters, and missing semicolons.

Table 6. Operators

Operator type	Character(s)	Description
Arithmetic	+ − * / **	Addition or prefix plus Subtraction or prefix minus Multiplication Division Exponentiation
Comparison	= ¬= < ¬< > ¬> <= >=	Equal to Not equal to Less than Not less than Greater than Not greater than Less than or equal to Greater than or equal to
Logical	¬ & |	Not, Exclusive-or And Or
String	||	Concatenation

The characters used for delimiters can be used in other contexts. For example, the period is a delimiter when used in name qualification (for example, `Weather.Temperature`), but is a decimal point in an arithmetic constant (for example, `3.14`).

Blanks

You can surround each operator or delimiter with blanks (b). One or more blanks must separate identifiers and constants that are not separated by some other delimiter. Any number of blanks can appear wherever one blank is allowed.

Blanks cannot occur within identifiers, composite symbols, or constants (except in character, mixed, widechar and graphic string constants).

Some examples are:

`ab+bc`	is equivalent to	`Ab + Bc`
`Table(10)`	is equivalent to	`TABLEb(b10bbb)`
`First,Second`	is equivalent to	`first,bsecond`
`AtoB`	is not equivalent to	`AbtobB`

Other cases that require or allow blanks are noted where those language features are discussed.

Comments

Comments are allowed wherever blanks are allowed as delimiters. A comment is treated as a blank and used as a delimiter. Comments are ignored and do not affect the logic of a program. Use the following syntax when for comments.
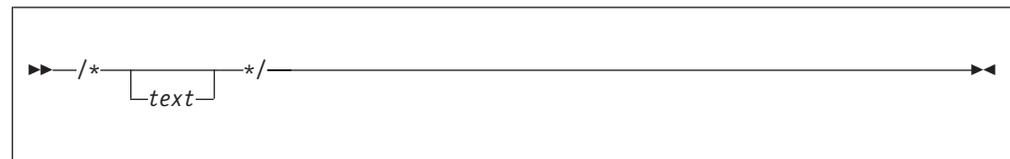
/*   Specifies the beginning of a comment.

text

    Specifies any sequences of characters except the */ composite symbol, which would terminate the comment.

*/   Specifies the end of the comment.

A comment can be entered on one or more lines, for example:

```
A = /* This comment is on one line */ 21;

    /* This comment spans
       two lines         */
```

In the following example, what appears to be a comment is actually a character string constant because it is enclosed in quotes.

```
A = '/* This is a constant, not a comment */' ;
```

Statements

You use identifiers, delimiters, operators, and constants to construct PL/I statements.

Although your source program consists of a series of records or lines, PL/I views the program as a continuous stream of characters. There are few restrictions in the format of PL/I statements, and programs can be written without considering special coding rules or checking to see that each statement begins in a specific column. A statement can begin in the next position after the previous statement, or it can be separated by any number of blanks.

Some statements begin with a % symbol. These statements are either %directives that direct preprocessor and compiler operations (controlling listings, including program source text from a library, and so on) or are PL/I macro facility %statements. A %directive must be on a line by itself.

To improve program readability and maintainability and to avoid unexpected results caused by loss of trailing blanks in source lines:

- Do not split a language element across lines. If a string constant must be written on multiple lines, use the concatenation operator (∥).
- Do not write more than one statement on a line.
- Do not split %directives across lines.

The PL/I statements, macro facility %statements, and the %directives are alphabetically listed in Chapter 9, "Statements and directives," on page 187.

Syntax for a PL/I statement:

```
►►─────┬───────────────────┬──┬──────────────┬──┬────────────┬─────►◄
       └─condition-prefix──┘  └─label-prefix─┘  └─statement──┘
```
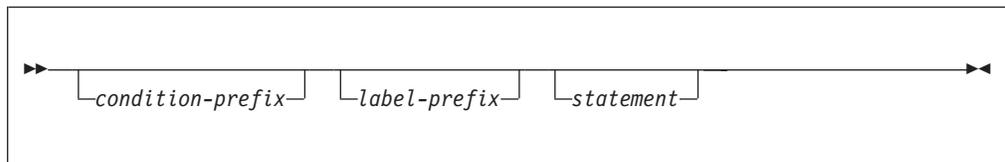
Syntax for a %directive:

```
►►─%─┬────────────┬─────────────────────────────────────────────►◄
     └─statement──┘
```
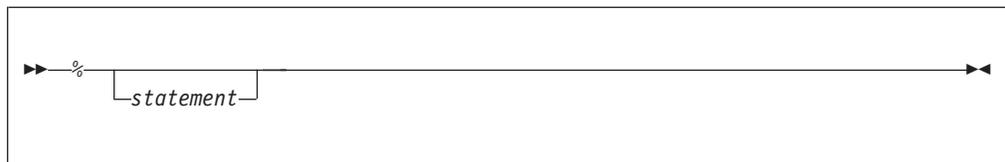
Syntax for a %statement:
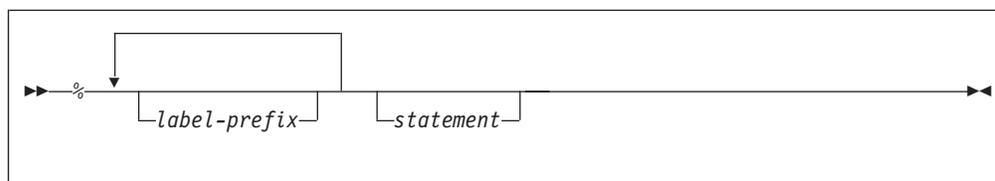
Syntax for a macro statement:
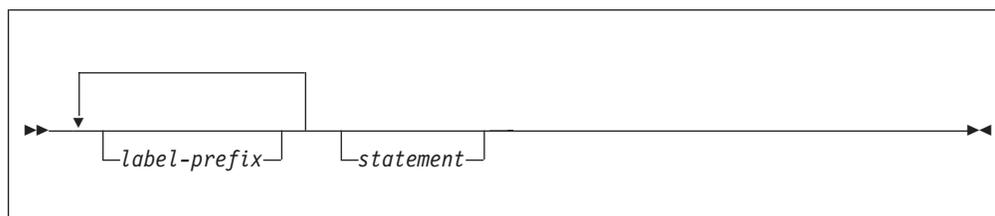
Every statement must be contained within some enclosing group or block. Macro statements must be contained within some enclosing macro group or procedure.

condition-prefix

A *condition prefix* specifies the enabling or disabling of a PL/I condition (see Chapter 16, "Condition handling," on page 331).

label-prefix

A *label prefix* is one or more statement labels. It identifies a statement so that it can be referred to at some other point in the program. Statement labels are either label constants (discussed in "Label data and LABEL attribute" on page 47), entry constants (discussed in "Entry data" on page 111), or format constants (discussed in "Format data and FORMAT attribute" on page 49).

Any statement, except DECLARE, DEFAULT, WHEN, OTHERWISE, and ON statements, can have a label prefix. Use the following syntax for a label prefix.
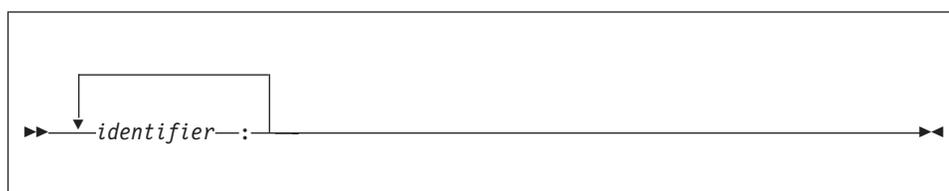
The syntax for individual statements throughout this book generally does not show the condition prefix or the label prefix.

statement

A simple or a compound statement.

Simple statements

The types of simple statements are keyword, assignment, and null.

A *keyword statement* is a statement that begins with a keyword. This keyword indicates the function of the statement. In the following example, READ and DECLARE are keywords.

```
read file(In) into(Input);          /* keyword statement  */
%declare Text char;                 /* keyword %statement */
```

The *assignment statement* contains one or more identifiers on the left side of the assignment symbol (=) and an expression on the right. It does not begin with a keyword:

```
A = B + C;                          /*  assignment statement  */
%Size = 15;                         /* % assignment statement   */
```

The *null statement* consists of only a semicolon and is a nonoperational statement.

```
     ;                              /* null statement        */
Label:;                            /* labeled null statement */
  % ;                              /* % null statement       */
```

Compound statements

Compound statements are all keyword statements. Each begins with a keyword which indicates the purpose of the statement. A compound statement contains one or more simple or compound statements. There are four compound statements: IF, ON, WHEN, and OTHERWISE. A compound statement is terminated by the semicolon that also terminates the final statement of the compound statement.

The following are examples of compound statements:

```
on conversion
  onchar() = '0';

if Text = 'stmt' then
  do;
    select(Type);
      when('if') call If_stmt;
      when('do') call Do_stmt;
      when('') /* do nothing */ ;
      otherwise
        call Other_stmt;
    end;
    call Print;
  end;
end;

%if Type = 'AREA' %then
  %Size = Size + 16;
%else;
```

Groups

Statements can be contained within larger program units called groups. A *group* is either a do-group or a select-group. A do-group is a sequence of statements delimited by a DO statement and a corresponding END statement. A select-group is a sequence of WHEN statements and an optional OTHERWISE statement delimited by a SELECT statement and a corresponding END statement. The delimiting statements are considered to be part of the group.

When a group is used in a compound statement, control either flows into the group or bypasses it, effectively treating the group as if it were a single statement.

The flow of control within a group is discussed for do-groups under "DO statement" on page 195 and for select-groups under "SELECT statement" on page 217.

Every group must be contained within some enclosing group or block. Groups can contain none, one, or more statements, groups, or blocks.

Double-byte character set

Each character in the double-byte character set (DBCS) is stored in 2 bytes. When the GRAPHIC compiler option is in effect, some source language elements can be written using DBCS and SBCS characters. In particular, you can use DBCS characters in the source program in following places:

- inside comments
- as part of statement labels and identifiers
- in G or M literals

However INCLUDE file names and the TITLE option of FETCH statements must be in SBCS.

DBCS identifiers

DBCS identifiers can be composed of single-byte characters in DBCS form, double-byte characters, or a combination of both.

Single-byte identifiers in DBCS form

DBCS identifiers containing only single-byte characters must conform to the normal PL/I naming conventions, including the first-character rule. A DBCS identifier containing single-byte characters expressed as DBCS equivalents is a synonym of the same identifier in SBCS.

Notes:

1. This book uses the symbol "**.**" (bold period) to represent the first byte of a double-byte character that can also be represented as SBCS.
2. This book uses "kk" to represent a double-byte character.

Example:

```
 .I.B.M = 3;     /* is the same as IBM=3; */
```

DBCS identifiers containing double-byte characters

The number of bytes used in a DBCS name cannot exceed the value specified as the maximum length of a name specified in the compiler LIMITS option.

SBCS characters expressed in DBCS form within a DBCS identifier are considered to be SBCS, for example:

```
 AkkB
 Akk.B
 .AkkB           /* are all AkkB  (SBCS-DBCS-SBCS) */
```

Uses for double-byte character identifiers

A DBCS identifier can be used wherever an SBCS identifier is allowed. When DBCS identifiers are used for EXTERNAL names and %INCLUDE file names, you must ensure that the identifiers are acceptable to the operating system, or are made acceptable by one of the following:

- The EXTERNAL attribute with an environment-name is used.
- The TITLE option of the OPEN statement is used.

Statement elements for DBCS

This section provides supplemental information about writing PL/I language elements using DBCS. Definitions of the language elements in this section and general usage rules appear in corresponding sections in "Statement elements for SBCS" on page 13.

DBCS statement elements

The following is a list of the language elements that may be coded using DBCS, an explanation of the rules, and examples of usage.

Identifiers

Use SBCS, DBCS or both.

```
dcl Eof                 /* in SBCS, is the same as    */
dcl .E.o.f              /* this in DBCS.              */

dcl kkkkX                /* these are all the same, where */
dcl kkkk.X              /* kk is a valid              */
dcl kkkkx               /* DBCS character             */
dcl kkkk.x              /*                            */
```

Comments

Use SBCS, DBCS or both.

```
/* comment */           /* all SBCS                      */
/* .T.y.p.e kk */       /* DBCS text */
```

Comment delimiters must be specified in SBCS.

Mixed Character String Constant

Enclose in SBCS or DBCS quotes.

Data can be expressed in SBCS or DBCS or both. The DBCS portion is not converted to SBCS.

'.a.b.c'M	stored as	.a.b.c	6 bytes
'.I.B.M.'.S'M	stored as	.I.B.M.'.S	10 bytes
'.I.B.M''.S'M	stored as	.I.B.M'.S	9 bytes
'IBMkk'M	stored as	IBMkk	5 bytes

Graphic Constant

Enclose in SBCS or DBCS quotes.

```
'.a.b.c'G               /* 6 byte graphic constant  */
'.I.B.M.'  .S'G         /* 10 byte graphic constant .I.B.M.'.S  */
```

G literals can start and end with DBCS quotes, and in that case, the G itself can also be specified in DBCS

```
'.a.b.c'G  .'.a.b.c.'G
.'.a.b.c.'.G
```

DBCS continuation rules

If a string literal or an identifier has a DBCS character that is separated from the right margin by a single SBCS blank, then the blank is ignored and the statement element is continued at the left margin of the next record.

Chapter 3. Data elements

This chapter introduces the kinds of data you can use in PL/I programs and the attributes you use to describe them. The discussion covers:
   A review of data items
   A review of variables and constants
   The types of data that are available and the attributes that define them

For information on how to declare data, refer to Chapter 8, "Data declarations," on page 147.

Data items

A *data item* is either the value of a variable or a constant. (These terms are not exactly the same as in general mathematical usage. They are discussed further in the next section.) Data items can be single items, called *scalars*, or they can be a collection of items called *data aggregates*.

Data aggregates are groups of data items that can be referred to either collectively or individually. The kinds of data aggregates are *arrays*, *structures*, and *unions*. You can use any type of computational or program-control data to form a data aggregate.

Arrays are discussed in "Arrays" on page 167, structures in "Structures" on page 170, unions in "Unions" on page 172, and arrays of structures and unions starting in "Combinations of arrays, structures, and unions" on page 176.

Variables

A *variable* has a value or values that might change during execution of a program. A variable is introduced by a declaration, which declares the name and certain attributes of the variable. However, a variable having the NONASSIGNABLE attribute is assumed not to change during execution. (Refer to "ASSIGNABLE and NONASSIGNABLE attributes" on page 242 for more information.) A *variable reference* is one of the following:
- A declared variable name
- A reference derived from a declared name through one or more of the following:
  - Pointer qualification
  - Structure qualification
  - Subscripting

(See Chapter 4, "Expressions and references," on page 51.)

Constants

A *constant* has a value that cannot change. Constants for computational data are referred to by stating the value of the constant or naming the constant in a DECLARE statement. For more information on declaring named constants, see "Named constants" on page 46.

Constants for program-control data are referred to by name.

Using quotation marks

String constants, hexadecimal constants, and the picture-specification are enclosed in either single or double quotation marks.

The following rules apply to quotation marks within a string:
- If the included quotation marks are the same type as those used to enclose the string, you must enter two quotation marks (that is, '' or "") for each occurrence to be included.
- If the included quotation marks are the type not used to enclose the string, enter only one quotation mark for each instance to be included. The single occurrence is treated as data.

Examples:

```
'Shakespeare''s "Hamlet"' is identical to
"Shakespeare's ""Hamlet"""

PICTURE "99V9" is identical to
PICTURE '99V9'
```

Note: The syntax diagrams in this book show single quotation marks. Double quotation marks can be substituted unless otherwise noted.

Punctuating constants

To improve readability, arithmetic, bit, and hexadecimal constants can use the break character (_).

'1100_1010'B	is the same as	'11001010'B
1100_1010B	is the same as	11001010B
'C_A'B4	is the same as	'ca'b4
'C_A'XN	is the same as	'ca'XN
16_777_216	is the same as	16777216

Data types and attributes

Data used in a PL/I program can be classified as either computational data or program-control data:

Computational data
Represents values that are used in computations to produce a desired result. Arithmetic and string data constitute computational data.

Arithmetic data is either coded arithmetic data or numeric picture data.

Coded arithmetic data items are rational numbers. They have the data attributes of *base* (BINARY or DECIMAL), *scale* (FLOAT or FIXED), *precision* (significant digits and decimal-point placement), and *mode* (REAL or COMPLEX).

Numeric picture data is numeric data that is held in character form and is discussed under "Numeric character data" on page 42.

A *string* is a sequence of contiguous characters, bits, widechars or graphics that are treated as a single data item.

Program-control data
Represents values that are used to control execution of your program. It consists of the following data types—area, entry, label, file, format, pointer, and offset.

For example:

```
Area = (Radius**2) * 3.1416;
```

`Area` and `Radius` are coded arithmetic variables of computational data. The numbers `2` and `3.1416` are coded arithmetic constants of computational data.

If the number `3.1416` is used in more than one place in the program, or if it requires specific data or precision attributes, you should declare it as a named constant. Thus, the above statement can be coded as:

```
dcl Pi FIXED DECIMAL (5,4) VALUE(3.1416);
area = (radius**2) * Pi;
```

Constants for program-control data have a value that is determined by the compiler. In the following example, the name `loop` represents a label constant of program-control data. The value of `loop` is the address of the statement `A=2*B;`.

```
loop:  A=2*B;
       C=B+6;
```

To work with a data item, PL/I needs to know the type of data and how to process it. *Attributes* provide this information. The kinds of attributes are data attributes and nondata attributes.

Data attributes

Describe computational data, program-control data, and program characteristics.

AREA	FILE	OFFSET	STRUCTURE
BINARY	FIXED	ORDINAL	TASK
BIT	FLOAT	PICTURE	TYPE
CHARACTER	FORMAT	POINTER	UNSIGNED
COMPLEX	GRAPHIC	PRECISION	UNION
DECIMAL	HANDLE	REAL	VARYING
DIMENSION	LABEL	RETURNS	VARYINGZ
ENTRY	NONVARYING	SIGNED	WIDECHAR

Nondata attributes
Describe nondata elements (for example, built-in functions) or provide additional description for elements that have other data attributes.

Nondata attributes

ABNORMAL	DEFINED	LIKE	PRINT
ALIGNED	DIRECT	LIST	RECORD
ASSIGNABLE	ENVIRONMENT	LITTLEENDIAN	SEQUENTIAL
AUTOMATIC	EXCLUSIVE	NONASSIGNABLE	STATIC
BASED	EXTERNAL	NONCONNECTED	STREAM
BIGENDIAN	GENERIC	NORMAL	UNALIGNED
BUFFERED	HEXADEC	OPTIONAL	UNBUFFERED
BUILTIN	IEEE	OPTIONS	UPDATE
BYADDR	INITIAL	OUTPUT	VALUE
BYVALUE	INPUT	PARAMETER	VARIABLE
CONDITION	INTERNAL	POSITION	
CONNECTED	KEYED		
CONTROLLED			

For example, the keyword CHARACTER is a data attribute for the string type of computational data. The keyword FILE is a data attribute for the file type of program-control data. The INTERNAL scope attribute specifies that the data item is known only within its declaring block.

The details on using keywords and expressions to specify the attributes are in Chapter 8, "Data declarations," on page 147. Briefly, you specify attributes:

    Explicitly, using a DECLARE statement
    Contextually, letting PL/I determine them
    By using programmer-defined or language-specified defaults

Table 7 on page 26 and Table 8 on page 27 help you correlate PL/I's variety of attributes with its variety of computational and program-control data types. The tables show that the constants and the named constants can only have the indicated data and scope attributes (Table 7 on page 26). Variables can specify additional attributes (Table 8 on page 27).

In the example,

```
Area = (Radius**2)*3.1416;
```

the constant 3.1416 is given the attributes:

    DECIMAL because it is not explicitly a binary constant
    FIXED because it is a fixed-point number
    PRECISION(5,4) (5 significant digits with 4 to the right of the decimal point)
    REAL because it does not have an imaginary part
    INTERNAL and ALIGNED

(See the "Coded arithmetic" row, and "Data Attributes" and "Scope Attributes" columns of Table 7 on page 26.)

The constant 1.0 (a decimal fixed-point constant) is different from the constants 1 (another decimal fixed-point constant), '1'B (a bit constant), '1' (a character constant), 1B (binary fixed-point constant), or 1E0 (a decimal floating-point constant).

In the following example, the variable `Pi` has the programmer-defined data attributes of FIXED and DECIMAL with a PRECISION of five digits, four to the right of the decimal point.

```
declare Pi fixed decimal(5,4) initial(3.1416);
```

Because this DECLARE statement contains no other attributes for `Pi`, PL/I applies the defaults for the remaining attributes:

REAL from the Data Attributes column
ALIGNED from the Alignment Attributes column
INTERNAL from the Scope Attributes column
AUTOMATIC from the Storage Attributes column
SIGNED from the Data Attributes column

(See the coded arithmetic row of Table 8 on page 27.)

Nondata attributes

Table 7. Classification of attributes by constant types

Constant Type	Data Attributes Notes 1 and 2	Scope Attributes Notes 1 and 2
Coded arithmetic	REAL | imaginary FLOAT | FIXED BINARY | DECIMAL PRECISION SIGNED	internal
Named coded arithmetic	REAL | COMPLEX FLOAT | FIXED BINARY | DECIMAL PRECISION VALUE SIGNED | UNSIGNED	internal
String	BIT | CHARACTER | GRAPHIC | WIDECHAR (*length*)	internal
Named string	BIT | CHARACTER | GRAPHIC | WIDECHAR [(*length*)] NONVARYING VALUE	internal
Named locator	POINTER | OFFSET | HANDLE VALUE	internal
Named picture	PICTURE REAL | COMPLEX VALUE	internal
File[Note 3]	FILE ENVIRONMENT STREAM | RECORD INPUT | OUTPUT | UPDATE SEQUENTIAL | DIRECT BUFFERED | UNBUFFERED[Note 4] KEYED PRINT	INTERNAL | EXTERNAL
Entry[Note 5]	ENTRY [RETURNS]	INTERNAL | EXTERNAL
Format[Note 5]	FORMAT	internal
Label[Note 5]	LABEL	internal

Notes:

1. Attributes in this table that appear in uppercase can be explicitly declared. Attributes that are in lowercase are implicitly given to the data type.

2. Defaults for data attributes are underlined. Because the data attributes for literal constants are contextual, defaults are not applicable. Named constants and file constants have selectable attributes, so defaults are shown.

3. File Attributes are described in Chapter 11, "Input and output," on page 257.

4. BUFFERED is the default for SEQUENTIAL files. UNBUFFERED is the default for DIRECT files.

5. Format and label constants, and INTERNAL entry constants cannot be declared in a DECLARE statement.

Table 8. Classification of attributes by variable types

Variable Type	Data Attributes	Alignment Attributes	Scope Attributes	Storage Attributes
Area	AREA(*size*)	ALIGNED	INTERNAL | EXTERNAL (INTERNAL is mandatory for AUTOMATIC BASED DEFINED PARAMETER)	AUTOMATIC | STATIC | BASED | CONTROLLED (AUTOMATIC is the default for INTERNAL; STATIC is the default for EXTERNAL) Defined variable: DEFINED [POSITION] Parameter: PARAMETER [CONNECTED | NONCONNECTED] [CONTROLLED] [INITIAL [CALL]] [VARIABLE] [NORMAL | ABNORMAL] ASSIGNABLE | NONASSIGNABLE
Coded arithmetic Note 1	REAL | COMPLEX FLOAT | FIXED BINARY | DECIMAL PRECISION [SIGNED | UNSIGNED]	ALIGNED | UNALIGNED		
Entry	ENTRY [RETURNS] [LIMITED]			
File	FILE			
Format	FORMAT			
Label	LABEL			
Locator	POINTER | HANDLE | {OFFSET [(area-variable)]}			
Ordinal	ORDINAL			
Picture	PICTURE REAL | COMPLEX	ALIGNED | UNALIGNED		
String	BIT | CHARACTER | GRAPHIC | WIDECHAR [(length)] [VARYING | VARYINGZ | NONVARYING]			
Task	TASK	ALIGNED | UNALIGNED		

Arrays: DIMENSION can be added to the declaration of any variable. Refer to "Arrays" on page 167 for more information.

Structures and unions:
- For a major structure or union: scope, storage (except INITIAL), alignment, STRUCTURE or UNION, and the LIKE attributes can be specified.
- For a member that is a structure or a union: alignment, STRUCTURE or UNION, and the LIKE attributes can be specified.
- Members always have the INTERNAL scope attribute.

Refer to "Structures" on page 170 and "Unions" on page 172 for more information.

Notes:

1. Undeclared names, or names declared without a data type, default to coded arithmetic variables. Default attributes are described in "Defaults for attributes" on page 162. Defaults shown are IBM defaults. ANS defaults are FIXED and BINARY rather than FLOAT and DECIMAL.

2. POSITION can be used only with string overlay defining.

Computational data types and attributes

This section describes the data types classified as computational data and the attributes associated with them.

Coded arithmetic data and attributes

Refer to "Data types and attributes" on page 22 for general information about coded arithmetic data.

Coded arithmetic data and attributes

Syntax for coded arithmetic data is shown in the following diagram:

```
>>─┬─ float sequence ─┬─┬─REAL──────┬─────────────────────────┬──><
   └─ fixed sequence ─┘ │       └─ precision specification ─┘ │
                        └─COMPLEX───────────────────────────────┘
                             └─ precision specification ─┘
```

float sequence:

```
├─FLOAT──────────────────────┬─DECIMAL────────────────────────────┬─┤
       └─ precision specification ─┘  └─ precision specification ─┘
                              └─BINARY──────────────────────────────┘
                                    └─ precision specification ─┘
```

fixed sequence:

```
├─FIXED──────────────────────────────────────────────────────────────>
       └─ precision specification ─┘
```

```
>─┬─DECIMAL──────────────────────────┬──────────────────────────────┤
  │     └─ precision specification ─┘ │
  └─BINARY──────────────────────────┬─SIGNED───┬─┘
        └─ precision specification ─┘ └─UNSIGNED─┘
```

precision specification:

```
├─┬─────────┬─┬─ float precision ─┬─────────────────────────────────┤
  └─PRECISION─┘ └─ fixed precision ─┘
```

float precision:

```
├──(number-of-digits)───────────────────────────────────────────────┤
```

fixed precision:

```
├──(number-of-digits──────────────)─────────────────────────────────┤
              └─,scaling-factor─┘
```

Table 9. Abbreviations for coded arithmetic data attributes

Attribute	Abbreviation
BINARY	BIN
COMPLEX	CPLX
DECIMAL	DEC
PRECISION	PREC

BINARY and DECIMAL attributes

The *base* of a coded arithmetic data item is either decimal or binary. DECIMAL is the default.

FIXED and FLOAT attributes

The *scale* of a coded arithmetic data item is either fixed-point or floating-point.

A fixed-point data item is a rational number in which the position of the decimal or binary point is specified, either by its appearance in a constant or by a scaling factor declared for a variable.

Floating-point data items are rational numbers in the form of a fractional part and an exponent part.

PRECISION attribute

The *precision* of a coded arithmetic data item includes the number of digits and the scaling factor. (The scaling factor is used only for fixed-point items).

number of digits
An integer that specifies how many digits the value can have. For fixed-point items, the integer is the number of significant digits. For floating-point items, the integer is the number of significant digits to be maintained excluding the decimal point (independent of its position).

scaling factor
An optionally-signed integer that specifies the assumed position of the decimal or binary point, relative to the rightmost digit of the number. If no scaling factor is specified, the default is 0.

The precision attribute specification is often represented as (p,q), where *p* represents the number of digits and *q* represents the scaling factor.

A negative scaling factor (-q) specifies an integer, with the point assumed to be located q places to the right of the rightmost actual digit. A positive scaling factor (q) that is larger than the number of digits specifies a fraction, with the point assumed to be located q places to the left of the rightmost actual digit. In either case, intervening zeros are assumed, but they are not stored; only the specified number of digits is actually stored.

If PRECISION is omitted, the precision attribute must follow, with no intervening attribute specifications, the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL or COMPLEX) attributes at the same factoring level.

If included, PRECISION can appear anywhere in the declaration.

Integer value means a fixed-point value with a scaling factor of zero.

REAL and COMPLEX attributes

The *mode* of an arithmetic data item (coded arithmetic or numeric character) is either real or complex.

A real data item is a number that expresses a real value.

A complex data item consists of two parts—a real part and an imaginary part. For a variable representing complex data items, the base, scale, and precision of the two parts are identical.

Arithmetic variables default to REAL.

An imaginary constant is written as a real constant of any type immediately followed by the letter I. Examples are:

```
27I
3.968E10I
11011.01BI
```

Each of these has a real part of zero. A complex value with a nonzero real part is represented by an expression with the following syntax:
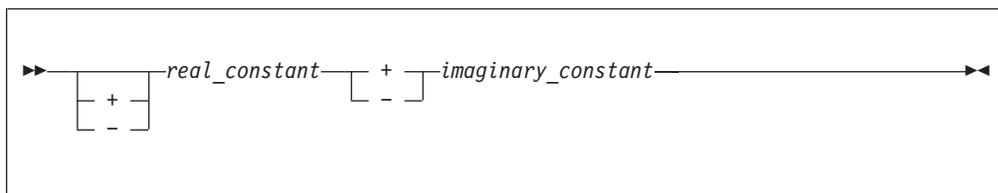
For example, 38+27I.

Given two complex numbers, y and z:

```
y = complex(A,B);
z = complex(C,D);
```

x=y/z is calculated by:

```
real(x) = (A*C + B*D)/(C**2 + D**2);
imag(x) = (B*C - A*D)/(C**2 + D**2);
```

x=y*z is calculated as follows:

```
real(x) = A*C - B*D;
imag(x) = B*C + A*D;
```

Computational conditions can be raised during these calculations.

SIGNED and UNSIGNED attributes

The SIGNED and UNSIGNED attributes can be used only with FIXED BINARY variables and ORDINAL variables. SIGNED indicates that the variable can assume negative values. UNSIGNED indicates that the variable can assume only nonnegative values.

UNSIGNED has the following effects on the semantics of fixed-point operations:

- The result of IAND, IEOR, INOT and IOR is UNSIGNED if all the operands are UNSIGNED.
- The result of ISLL and ISRL is UNSIGNED if the first operand is UNSIGNED.
- The result of REAL or IMAG is UNSIGNED if its operand is UNSIGNED.

If you are using the RULES(ANS) compiler option, UNSIGNED has the following effect on the semantics of fixed- point operations:

- The result of an add, multiply, or divide operation is UNSIGNED if both operands are UNSIGNED.

- The result of MAX or MIN is UNSIGNED if all operands are UNSIGNED
- The result of REM or MOD is UNSIGNED if all operands are UNSIGNED

The SIGNED and UNSIGNED attributes affect storage requirements, as shown in Table 10 and Table 11.

Table 10. FIXED BINARY SIGNED data storage requirements

This precision:	Occupies this amount of storage (bytes):
precision <= 7	1
7 < precision <= 15	2
15 < precision <= 31	4
31 < precision <= 63	8

Table 11. FIXED BINARY UNSIGNED data storage requirements

This precision:	Occupies this amount of storage (bytes):
precision <= 8	1
8 < precision <= 16	2
16 < precision <= 32	4
32 < precision <= 64	8

Binary fixed-point data

The data attributes for declaring binary fixed-point variables are BINARY and FIXED. For example:

```
declare Factor binary fixed (20,2);
```

Factor is declared as a variable that can represent binary fixed-point data of 20 data bits, two of which are to the right of the binary point.

Refer to "SIGNED and UNSIGNED attributes" on page 30 for information on how much storage signed and unsigned fixed-binary data occupy.

The declared number of data bits is in the low-order positions, but the extra high-order bits participate in any operation performed upon the data item. Any arithmetic overflow into such extra high-order bit positions can be detected only if the SIZE condition is enabled.

Binary fixed-point constant

A binary fixed-point constant consists of one or more bits with an optional binary point, followed immediately by the letter B. Binary fixed-point constants have a precision (p,q), where p is the total number of data bits in the constant, and q is the number of bits to the right of the binary point, for example:

Constant	Precision
1011_0B	(5,0)
1111_1B	(5,0)
101B	(3,0)
1011.111B	(7,3)

XN (hex) binary fixed-point constant

The XN constant describes a SIGNED REAL FIXED BINARY constant in hexadecimal notation. If the constant has 8 or fewer digits, it has a precision of 31; otherwise, it has a precision of 63.

XN (hex) binary constant

```
      ┌────────────┐
►►──'──▼─hex-digit─┴──'XN──────────────────────────────────►◄
```

Consider the following examples:

```
'100'XN            /*  same as '00000100'XN with value 256     */
'8000'XN           /*  same as '00008000'XN with value 32,768  */
'FFFF'XN           /*  same as '0000FFFF'XN with value 65,535  */
"ffff_ffff"XN      /*  is the value -1                         */
```

The hexadecimal value for the XN constant is the value specified padded on the left with hex zeros if necessary.

XU (hex) binary fixed-point constant

The XU constant describes an UNSIGNED REAL FIXED BINARY constant in hexadecimal notation. If the constant has 8 or fewer digits, it has a precision of 32; otherwise, it has a precision of 64.

```
      ┌────────────┐
►►──'──▼─hex-digit─┴──'XU──────────────────────────────────►◄
```
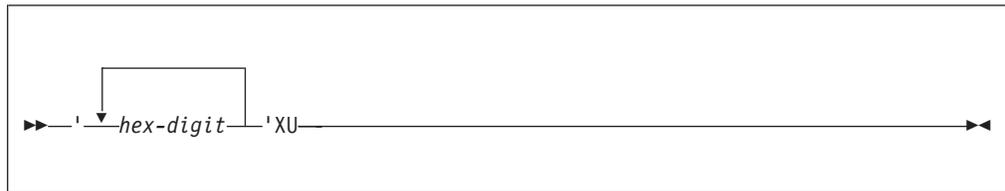
Consider the following examples:

```
'100'XU            /*  same as '00000100'XU with value 256     */
'8000'XU           /*  same as '00008000'XU with value 32,768  */
'FFFF'XU           /*  same as '0000FFFF'XU with value 65,535  */
"ffff_ffff"XU      /*  is the value 2**32-1                    */
```

The hexadecimal value for the XU constant is the value specified padded on the left with hex zeros if necessary.

Decimal fixed-point data

The data attributes for declaring decimal fixed-point variables are DECIMAL and FIXED. For example:

```
declare A fixed decimal (5,4);
```

specifies that A represents decimal fixed-point data of 5 digits, 4 of which are to the right of the decimal point.

These two examples:

```
declare B fixed (7,0) decimal;
declare B fixed decimal(7);
```

both specify that B represents integers of 7 digits.

This example

```
declare C fixed (7,-2) decimal;
```

specifies that C has a scaling factor of -2. This means that C holds 7 digits that range from -9999999*100 to 9999999*100, in increments of 100.

This example

```
declare D decimal fixed real(3,2);
```

specifies that `D` represents fixed-point data of 3 digits, 2 of which are fractional.

Decimal fixed-point data is stored two digits per byte, with a sign indication in the rightmost 4 bits of the rightmost byte. Consequently, a decimal fixed-point data item is always stored as an odd number of digits, even though the declaration of the variable can specify the number of digits, p, as an even number.

When the declaration specifies an even number of digits, the extra digit place is in the high-order position, and it participates in any operation performed upon the data item, such as in a comparison operation. Any arithmetic overflow or assignment into an extra high-order digit place can be detected only if the SIZE condition is enabled.

Decimal fixed-point constant

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point. Decimal fixed-point constants have a precision (p,q), where p is the total number of digits in the constant and q is the number of digits specified to the right of the decimal point. Examples are:

Constant	Precision
3.1416	(5,4)
455.3	(4,1)
732	(3,0)
1_200_300	(7,0)
003	(3,0)
5280	(4,0)
.0012	(4,4)

Binary floating-point data

The data attributes for declaring binary floating-point variables are BINARY and FLOAT. For example:

```
declare S binary float (16);
```

`S` represents binary floating-point data with a precision of 16 binary digits.

The exponent cannot exceed five decimal digits. If the declared precision is less than or equal to (21), short floating-point form is used. If the declared precision is greater than (21) and less than or equal to (53), long floating-point form is used. If the declared precision is greater than (53), extended floating-point form is used.

Binary floating-point constant

A binary floating-point constant is a mantissa followed by an exponent and the letter B. The mantissa is a binary fixed-point constant. The exponent is the letter E, S, D, or Q followed by an optionally-signed decimal integer (meaning 2 to the power of this integer). Constants using E have a precision (p) where p is the number of binary digits of the mantissa. Constants using S, D, and Q always have maximum single, double, and extended precisions, respectively. Examples are:

Constant	Precision
101101E5B	(6)
101.101E2B	(6)
11101E–28B	(5)
11.01E+42B	(4)
1S0b	(21)

Binary floating-point constant

```
1D0b          (53)
1Q0b          (64)(OS/2 and Windows)
1Q0b          (106)(AIX)
1Q0b          (109)(z/OS)
```

Decimal floating-point data

The data attributes for declaring decimal floating-point variables are DECIMAL and FLOAT. Consider this example:

```
declare Light_years decimal float(5);
```

The value for `Light_years` represents decimal floating-point data of 5 decimal digits.

If the declared precision is less than or equal to (6), short floating-point form is used. If the declared precision is greater than (6) and less than or equal to (16), long floating-point form is used. If the declared precision is greater than (16), extended floating-point form is used.

Decimal floating-point constant

A decimal floating-point constant is a mantissa followed by an exponent. The mantissa is a decimal fixed-point constant. The exponent is the letter E, S, D, or Q followed by an optionally-signed decimal integer of four or less digits (meaning 10 to the power of this integer). Constants using E have a precision (p) where *p* is the number of digits of the mantissa. Constants using S, D, and Q always represent single, double, and extended precision respectively. Examples are:

Constant	Precision
15E-23	(2)
15E23	(2)
4E-3	(1)
1.96E+07	(3)
438E0	(3)
3_141_593E-6	(7)
.003_141_593E3	(9)
1s0	(6)
1d0	(16)
1q0	(18)(**OS/2 and Windows**)
1q0	(32)(**AIX**)
1q0	(33)(**z/OS**)

The last five examples represent the same value (although with different precisions).

String data and attributes

Refer to "Data types and attributes" on page 22 for general information about strings.

BIT, CHARACTER, GRAPHIC and WIDECHAR attributes

The BIT attribute specifies a bit variable.

The CHARACTER attribute specifies a character variable. Character strings can also be declared using the PICTURE attribute.

The WIDECHAR attribute specifies a widechar variable which will hold UTF-16 data.

The GRAPHIC attribute specifies a graphic variable.

The syntax for the BIT, CHARACTER, GRAPHIC and WIDECHAR attributes is:

```
►►─┬─BIT───────┬─┬─(length─┬────────────────┬─)─┬─┬─NONVARYING─┬──────►◄
   ├─CHARACTER─┤ │         └─REFER(variable)─┘   │ ├─VARYING────┤
   ├─GRAPHIC───┤ └─(*)───────────────────────────┘ └─VARYINGZ───┘
   └─WIDECHAR──┘
```
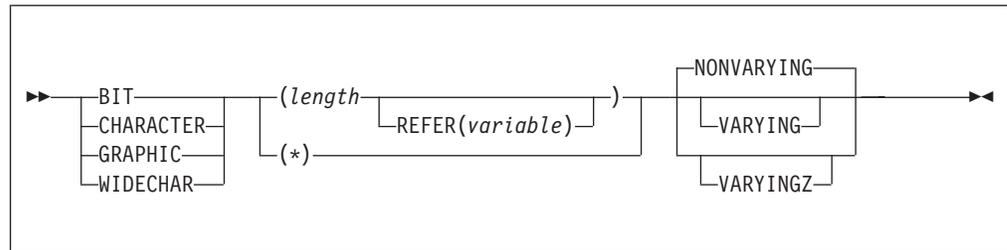
Table 12. Abbreviations for string data attributes

Attribute	Abbreviation
CHARACTER	CHAR
GRAPHIC	G
WIDECHAR	WCHAR
NONVARYING	NONVAR
VARYING	VAR
VARYINGZ	VARZ

length

Specifies the length of a NONVARYING string or the maximum length of a VARYING or VARYINGZ string. The length is in bits, characters, widechars or graphics (DBCS characters), as appropriate.

You can specify the length as an expression or as an asterisk. If the length is not specified, the default is 1. For named constants, length is determined from the length of the value expression.

For a parameter, an expression is valid only if it is CONTROLLED. An asterisk specification for a parameter indicates that the length is taken from the argument that is passed.

If the length specification is an expression, it is evaluated and converted to FIXED BINARY(31,0), which must be positive, when storage is allocated for the variable.

For STATIC data, length must be a restricted expression.

For BASED data, length must be a restricted expression, unless the string is a member of a structure or a union and the REFER option is used.

REFER

See "REFER option (self-defining data)" on page 236 for the description of the REFER option.

The statement below declares `User` as a variable that can represent character data with a length of 15:

```
declare User character (15);
```

The following example shows the declaration of a bit variable:

```
declare Symptoms bit (64);
```

VARYING, VARYINGZ, and NONVARYING attributes

The VARYING and VARYINGZ attributes specify that a variable can have a length varying from 0 to the declared maximum length. NONVARYING specifies that a variable always has a length equal to the declared length.

The storage allocated for VARYING strings is 2 bytes longer than the declared length. The leftmost 2 bytes hold the string's current length.

The storage allocated for a VARYINGZ character string is 1 byte longer than the declared length. The current length of the string is equal to the number of bytes before the first `'00'`x in the storage allocated to it.

The storage allocated for a VARYINGZ GRAPHIC string is 2 bytes longer than the declared length. The current length of the string is equal to half the number of bytes before the first `'0000'`gx in the storage allocated to it.

The storage allocated for a VARYINGZ WIDECHAR string is 2 bytes longer than the declared length. The current length of the string is equal to half the number of bytes before the first `'0000'`wx in the storage allocated to it.

The VARYINGZ attribute is not allowed with BIT strings.

In the following DECLARE statements, both `User` and `Zuser` represent varying-length character data with a maximum length of 15. However, unlike `User`, `Zuser` is null-terminated. The storage allocated is 17 bytes for `User` and 16 bytes for `Zuser`.

```
declare User character (15) varying;
declare Zuser character (15) varyingz;
```

The length for `User` and `Zuser` at any time is the length of the data item assigned to it at that time. You can determine the declared and the current length by using the MAXLENGTH and LENGTH built-in functions, respectively.

The null terminator held in a VARYINGZ string is not used in comparisons or assignments, other than to determine the length of the string. Consequently, although the strings in the following declarations have the same internal hex representation, they do **not** compare as being equal:

```
declare A char(4) nonvarying init( ('abc' ‖ '00'x) );
declare B char(3) varyingz   init( 'abc' );
```

To the contrary, *Z* and *C* in this example do compare as equal:

```
dcl Z char(3) nonvarying init('abc');
dcl C char(3) varyingz init('abc');
```

The VARYING and VARYINGZ strings can be passed and received as parameters with * length. They can be passed without a descriptor if they have the NONASSIGNABLE attribute.

PICTURE attribute

The PICTURE attribute specifies the properties of a character data item by associating a picture character with each position of the data item. A picture character specifies a category of characters that can occupy that position.

```
►►──PICTURE──'──picture-specification──'──────────────────────►◄
```
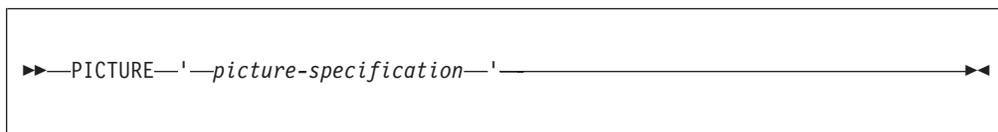
Abbreviation PIC

picture-specification

Describes either a character data item or a numeric character data item. Refer to "Picture characters for character data" on page 316 or "Picture characters for numeric character data" on page 317 for the valid characters.

A numeric picture specification specifies arithmetic attributes of numeric character data in much the same way that they are specified by the appearance of a constant.

Numeric character data has an arithmetic value but is stored in character form. Numeric character data is converted to coded arithmetic before arithmetic operations are performed.

The base of a numeric character data item is decimal. Its scale is either fixed-point or floating-point (the K or E picture character denotes a floating-point scale). The precision of a numeric character data item is the number of significant digits (excluding the exponent in the case of floating-point). Significant digits are specified by the picture characters for digit positions and conditional digit positions. The scaling factor of a numeric character data item is derived from the V or the F picture character or the combination of V and F.

Only decimal data can be represented by picture characters. Complex data can be declared by specifying the COMPLEX attribute along with a single picture specification that describes either a fixed-point or a floating-point data item.

For more information on numeric character data, see "Numeric character data" on page 42.

Character data

Data with the CHARACTER attribute can contain any of the 256 characters supported by the character set. Data with the PICTURE attribute must have characters that match the picture-specification characters. Each character occupies 1 byte of storage.

Character constant

A character constant is a contiguous sequence of characters enclosed in single or double quotation marks.

Quotation marks included in the constant follow the rules listed in "Using quotation marks" on page 22. The length of a character constant is the number of characters between the enclosing quotation marks counting any doubled quotation marks as a single character.

A null character constant is written as two quotation marks with no intervening blank. The syntax for a character constant is:
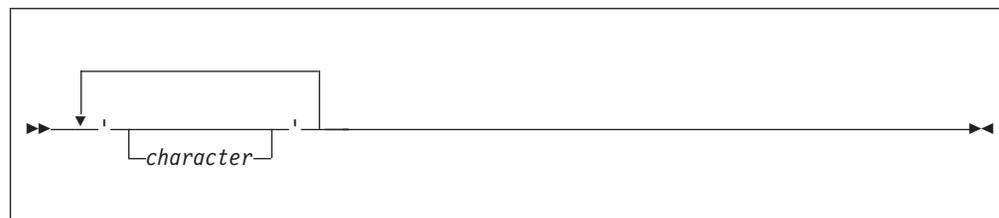
Examples of character constants are:

Constant   **Length**

Character constant

```
'Shakespeare''s "Hamlet"'      22
"Shakespeare's ""Hamlet"""      22
"Page 5"                        6
'/* This is a comment */'      27
''                              0
(2)'Walla '                    12
```

In the last example, the number in parentheses is a *string repetition factor*, which indicates repetition of the characters that follow. This example is equivalent to the constant "Walla Walla ". The string repetition factor must be a constant and enclosed in parentheses.

X (hex) character constant

The X character constant is a contiguous sequence of an even number of hex digits enclosed in single or double quotation marks and followed immediately by the letter X. Each pair of hex digits represents one character.

The length of an X constant is half the number of hex digits specified.

A null X constant is written as two quotation marks followed by the X suffix.
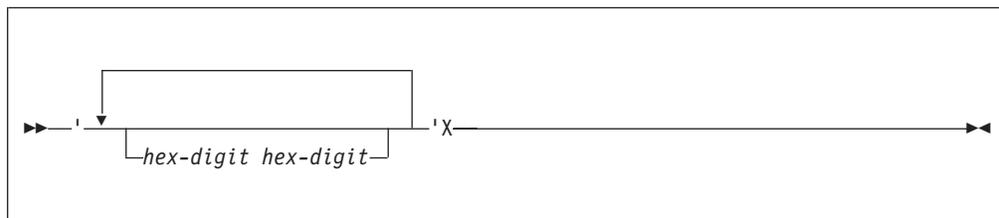
Examples of X constants are:

Constant	Length
"0d0A"x	2
''X	0

Note: The use of X constants can limit the portability of a program.

Bit data

Data with the BIT attribute allows manipulation of storage in terms of bits. Each byte of storage is composed of 8 bits.

Bit constant

A bit constant is a contiguous sequence of binary digits enclosed in single or double quotation marks and followed immediately by the letter B.
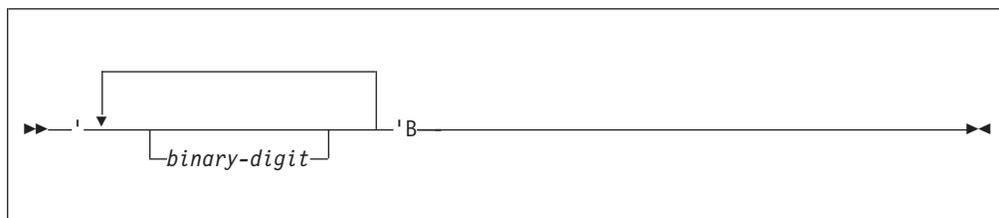
A null bit constant is written as two quotation marks, followed by B.

Examples of bit constants are:

Constant	Length
'1'B	1
"1100_1010_11"B	10
(64)'0'B	64
''B	0
'0'B	1

The number in parentheses in the third example is a string repetition factor which specifies that the following series of bits is repeated the specified number of times. The example shown would result in a string of 64 zero bits.

(See "Source-to-target rules" on page 77 for a discussion on the conversion of bit-to-character data and character-to-bit data.)

B4 (hex) bit constant

The B4 bit constant is a contiguous sequence of hex digits enclosed in single or double quotation marks and followed immediately by B4. Each hex digit represents four bits. BX is a synonym for B4.
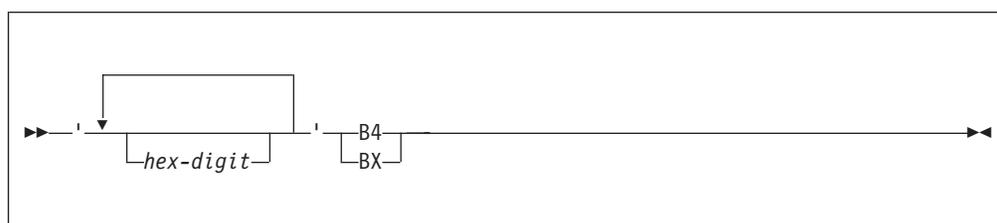
Some examples of B4 constants are:

'CA'B4	is the same as	"1100_1010"B
'80'B4	is the same as	'1000_0000'B
'1'B4	is the same as	'0001'B
(2)'F'B4	is the same as	'1111_1111'B
(2)'F'B4	is the same as	'FF'BX
''B4	is the same as	""B

B3 (octal) bit constant

The B3 bit constant is a contiguous sequence of octal digits enclosed in single or double quotation marks and followed immediately by B3. Each octal digit represents three bits.

Some examples of B3 constants are:

'22'B3	is the same as	"010_010"B
'40'B3	is the same as	'100_000'B
'1'B3	is the same as	'001'B
(2)'7'B3	is the same as	'111_111'B
''B3	is the same as	""B

Graphic data

GRAPHIC data can contain any DBCS character. Each DBCS character occupies 2 bytes of storage.

Graphic constant

A graphic constant is a contiguous sequence of DBCS characters enclosed in single or double quotation marks. Graphic constants take up 2 bytes of storage for each DBCS character in the constant.

G literals can start and end with DBCS quotes. G can also be DBCS.
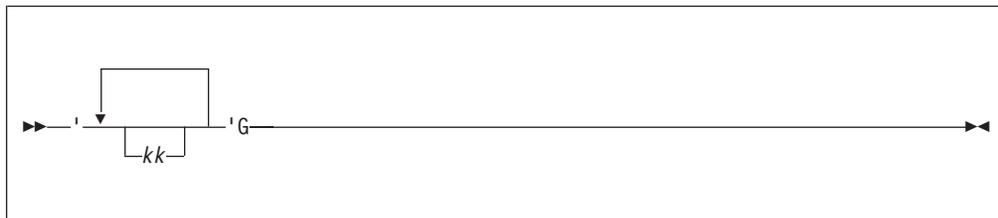
The GRAPHIC compiler option must be in effect for graphic constants to be accepted. If the GRAPHIC ENVIRONMENT option is not specified for STREAM I/O files that include graphic constants, the CONVERSION condition is raised.

GX (hex) graphic constant

The GX graphic constant is a contiguous sequence of hex digits, in multiples of 4, enclosed in single or double quotation marks and followed immediately by GX. Each group of 4 hex digits represents one DBCS character.
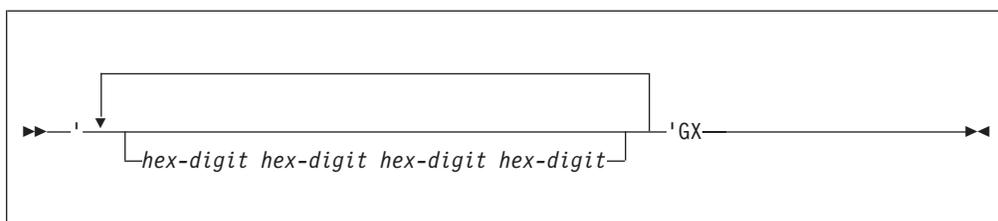
Examples:

```
'81a1'gx        represents one DBCS character
""gX            is the same as ''g
```

Note: The use of GX can limit the portability of a program.

Mixed character data

Mixed character data can contain SBCS and DBCS characters. Mixed data is represented by the CHARACTER data type, and follows the processing rules for CHARACTER data.

The CHARGRAPHIC option of the OPTIONS attribute and the MPSTR built-in function can be used to assist in mixed data handling. For more information on CHARGRAPHIC see "OPTIONS option and attribute" on page 126; for information on MPSTR, see "MPSTR" on page 440.

M (Mixed) character constant

An M constant is a contiguous sequence of DBCS and/or SBCS characters enclosed in quotation marks (single or double), followed immediately by the letter M. Quotations marks included in the constant follow the rules listed in "Using quotation marks" on page 22. The length of an M constant is the number of SBCS characters between the enclosing quotation marks counting any doubled quotation marks as a single character, plus twice the number of DBCS characters in the string.

A null M constant is written as two quotation marks followed by M.

```
>>──'──┬──────────────┬──'M────────────────────────────────────><
       │ ◄──────────┐ │
       └──character──┘
       └──kk──────────┘
```

Examples of mixed character constants are:

Constant	Length
'IBM kkkk'M	8 bytes on PS/2, 10 on S/370
'.I.B.M'M	6 bytes on PS/2, 8 on S/370
''M	0

The GRAPHIC compiler option must be in effect for mixed constants to be accepted. If the GRAPHIC ENVIRONMENT option is not specified for STREAM I/O files having mixed constants, the CONVERSION condition is raised.

Note: Because of the use of shift-codes on some computers, the use of mixed data and M constants can limit program portability.

Widechar data
WIDECHAR data can contain any UTF-16 character. Each widechar occupies 2 bytes of storage.

There is currently no support yet for

• WIDECHAR characters in source files

• W string constants

• use of WIDECHAR expressions in stream I/O

• implicit conversion to/from WIDECHAR in record I/O

• implicit endianness flags in record I/O

  If you create a WIDECHAR file, you should write the endianness flag ('fe_ff'wx) as the first two bytes of the file.

WX (hex) widechar constant
The WX widechar constant is a contiguous sequence of hex digits, in multiples of 4, enclosed in single or double quotation marks and followed immediately by WX. Each group of 4 hex digits represents one UTF-16 character.

```
>>──'──┬──────────────────────────────────────────┬──'WX────────><
       │ ◄──────────────────────────────────────┐ │
       └──hex-digit hex-digit hex-digit hex-digit─┘
```
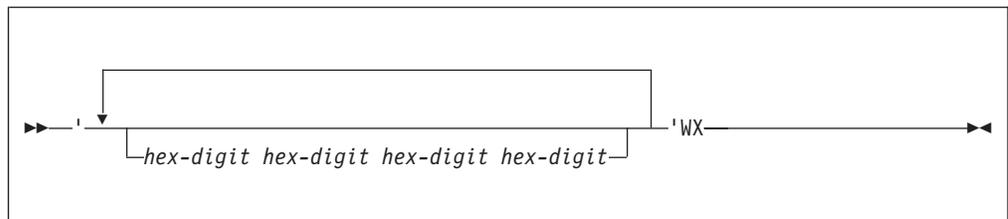
Examples:

'0031'wx	represents one UTF-16 character
""wX	is the same as ''w

> **Note:** WX constants should be specified in bigendian format (even if the program will run in littleendian format). So, for example, the widechar value for the character '1' should always be specified as '0031'wx (and not as '3100'wx).

> **Note:** The use of WX can limit the portability of a program.

Numeric character data

A numeric character data item is the value of a variable that has been declared with the PICTURE attribute with a numeric picture specification. The data item is the character representation of a decimal fixed-point or floating-point value.

Numeric picture specification describes a character string that can be assigned only data that can be converted to an arithmetic value.

For example:

```
declare Price picture '999V99';
```

specifies that any value assigned to `Price` is maintained as a character string of five decimal digits, with an assumed decimal point preceding the rightmost two digits. Data assigned to `Price` is aligned on the assumed point in the same way that point alignment is maintained for fixed-point decimal data.

Numeric character data has arithmetic attributes, but it is not stored in coded arithmetic form. Numeric character data is stored as a character string. Before it can be used in arithmetic computations, it must be converted either to decimal fixed-point or to decimal floating-point format. Such conversions are done automatically, but they require extra processing time.

Although numeric character data is in character form, like character strings, and although it is aligned on the decimal point like coded arithmetic data, it is processed differently from the way either coded arithmetic items or character strings are processed. Editing characters can be specified for insertion into a numeric character data item, and such characters are actually stored within the data item. Consequently, when the item is printed or treated as a character string, the editing characters are included in the assignment operation. However, if a numeric character item is assigned to another numeric character or arithmetic variable, the editing characters are not included in the assignment operation—only the actual digits, signs, and the location of the assumed decimal point are assigned. For example:

```
declare Price picture '$99V.99',
        Cost character (6),
        Amount fixed decimal (6,2);
Price = 12.28;
Cost = '$12.28';
```

In the picture specification for PRICE, the currency symbol ($) and the decimal point (.) are editing characters. They are stored as characters in the data item. However, they are not a part of its arithmetic value. After both assignment statements are executed, the actual internal character representation of `Price` and `Cost` can be considered identical. If they were printed, they would print exactly the same; but they do not always function in the same way. For example:

```
Amount = Price;
Cost   = Price;
Amount = Cost;
Price  = Cost;
```

After the first two assignment statements are executed, the value of `Amount` is 0012.28 and the value of `Cost` is '$12.28'. In the assignment of `Price` to `Amount`, the currency symbol and the decimal point are editing characters, and they are not part of the assignment. The numeric value of `Price` is converted to internal coded arithmetic form. In the assignment of `Price` to `Cost`, however, the assignment is to a character string, and the editing characters of a numeric picture specification always participate in such an assignment. No conversion is necessary because `Price` is stored in character form.

The third and fourth assignment statements would raise the CONVERSION condition. The value of `Cost` cannot be assigned to `Amount` because the currency symbol in the string makes it invalid as an arithmetic constant. The value of `Cost` cannot be assigned to `Price` for the same reason. Only values that are of arithmetic type, or that can be converted to arithmetic type, can be assigned to a variable declared with a numeric picture specification.

Although the decimal point can be an editing character or an actual character in a character string, it does not raise the CONVERSION condition in converting to arithmetic form, since its appearance is valid in an arithmetic constant. The same is true for a valid plus or minus sign, because converting to arithmetic form provides for a sign preceding an arithmetic constant.

Other editing characters, including zero suppression characters, drifting characters, and insertion characters, can be used in numeric picture specifications. For a complete discussion of picture characters, see Chapter 15, "Picture specification characters," on page 315.

Date attribute

Implicit date comparisons and conversions are made by the compiler if the two operands have the DATE attribute. The DATE attribute specifies that a variable (or argument or returned value) holds a date with a specified pattern.

```
►►──DATE─────────────────────────────────────────────►◄
          └─('pattern')─┘
```
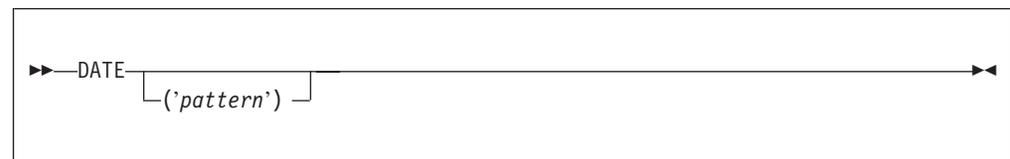
pattern
> One of the supported date patterns. If you do not specify a pattern, YYMMDD is the default.

The DATE attribute is valid only with variables having one of the following sets of attributes:
- CHAR(n) NONVARYING
- PIC'(n)9' REAL
- FIXED DEC(n,0) REAL

The length or precision of *n* must match the length of the *pattern*.

When you specify the RESPECT compile-time option (see the *Programming Guide* for details), the following occur:
- The compiler knows to honor the DATE attribute
- The DATE built-in function returns a value that has the attribute DATE('YYMMDD').

Date attribute

This allows DATE() to be assigned to a variable with the attribute DATE('YYMMDD') without an error message being generated. If DATE() is assigned to a variable not having the DATE attribute, however, an error message is generated.

Implicit DATE comparisons: The DATE attribute causes implicit *commoning* when two variables declared with the DATE attribute are compared. Comparisons where only one variable has the DATE attribute are flagged, and the other comparand is generally treated as if it had the same DATE attribute, although some exceptions apply which are discussed later.

Implicit commoning means that the compiler generates code to convert the dates to a common, comparable representation. This process converts 2-digit years using the *window* you specify in the WINDOW compile-time option.

In the following code fragment, if the DATE attribute is honored, then the comparison in the second display statement is 'windowed'. This means that if the window started at 1900, the comparison would return false. However, if the window started at 1950, the comparison would return true.

```
dcl a    pic'(6)9' date;
dcl b    pic'(6)9' def(a);
dcl c    pic'(6)9' date;
dcl d    pic'(6)9' def(c);

b = '670101';
d = '010101';

display( b || ' < ' || d || ' ?' );
display( a < c );
```

Date comparisons can occur in the following places:
* IF and SELECT statements
* WHILE or UNTIL clauses
* Implicit comparisons caused by a TO clause

Comparing dates with like patterns: The compiler does not generate any special code to compare dates with identical patterns under the following conditions:
* The comparison operator of = or ¬= is used
* The pattern is equal to YYYY, YYYYMM, YYYYDDD, or YYYYMMDD.

Comparing dates with differing patterns: For comparisons involving dates with unlike patterns, the compiler generates code to convert the dates to a common comparable representation. Once the conversion has taken place, the compiler compares the two values.

Comparisons involving the DATE attribute and a literal: If you are making comparisons in which one comparand has the DATE attribute and the other is a literal, the compiler issues a W-level message. Further compiler action depends on the value of the literal as follows:

* If the literal appears to be a valid date, it is treated as if it had the same date pattern and window as the comparand with the DATE attribute.

* If the literal does not appear to be a valid date, the DATE attribute is ignored on the other comparand.

```
dcl start_date char(6) date;
if start_date >= '' then /* no windowing */
...
if start_date >= '851003' then /* windowed */
...
```

Comparisons involving the DATE attribute and a non-literal: In comparisons where one comparand has the DATE attribute and the other is not a date and not a literal, the compiler issues an E-level message. The non-date value is treated as if it had the same date pattern as the other comparand and as if it had the same window.

```
dcl start_date char(6) date;
dcl non_date char (6);

if start_date >= non_date then /* windowed */
...
```

Implicit DATE assignments: The DATE attribute can also cause implicit conversions to occur in assignments of two variables declared with date patterns.

* If the source and target have the same DATE and data attributes, then the assignment proceeds as if neither had the DATE attribute.
* If the source and target have differing DATE attributes, then the compiler generates code to convert the source date before making the assignment.
* In assignments where the source has the DATE attribute but the target does not, the compiler issues an E-level message and ignores the DATE attribute.
* In assignments where the target has the DATE attribute but the source does not (and the source IS NOT a literal), the compiler issues an E-level message and ignores the DATE attribute.
* In assignments where the target has the DATE attribute but the source does not (and the source IS a literal), the compiler issues a W-level message and ignores the DATE attribute.

  ```
  dcl start_date char(6) date;
  start_date = '';
  ...
  ```
* If the source holds a four-digit year and the target holds a two-digit year, the source can hold a year that is not in the target window. In this case, the ERROR condition is raised.

  ```
  dcl x char(6) date;
  dcl y char(8) date('YYYYMMDD');

  y = '20600101';

  x = y; /* raises error if window is <= 1960 */
  ```
* The DATE attribute is ignored in:
  - The debugger
  - Assignments performed in record I/O statements
  - Assignments and conversions performed in stream I/O statements (such as GET DATA).

Even if you do not choose a windowing solution, you might have some code that needs to manipulate both two- and four-digit years. You can use multiple date patterns to help you in these situations:

```
dcl old_date char(6) date('YYMMDD');
dcl new_date char(8) date('YYYYMMDD');

new_date = old_date;
```

Date diagnostics: In PL/I, *effective* assignments occur when

* An expression is passed as an argument to an entry that has described that argument
* An expression is used in a RETURN statement.

Date attribute

The following uses of date variables are flagged:
- Assignments (explicit or effective) which include either
  - A date to a non-date
  - A non-date to a date
- Any arithmetic operation applied to a date
- Use of a date in a BY clause (since this implies an arithmetic operation)
- Use of a date in any mathematical built-in function
- Use of a date in any arithmetic built-in function except BINARY, DECIMAL, FIXED, FLOAT, or PRECISION.
- Use of a date in the built-in functions SUM, PROD, or POLY.

In all of the cases listed previously, code is produced but no windowing occurs. In effect, the DATE attribute is ignored.

Named constants

A named constant is a scalar identifier declared with the VALUE attribute along with other data attributes. All references to the name are logically treated as a reference to the appropriate constant but with the complete set of attributes, whether explicitly declared or defaulted.

Note: The effect of the use of a named constant might not be exactly the same as the use of an unnamed constant. The attributes for a named constant are taken from the declaration which includes explicit and default attributes. The attributes for an unnamed constant are deduced from the shape, form, and size of the constant. For string data, if the length is not specified, or is specified with an asterisk, the length is determined from the length of the restricted expression.

Named constants can be more precise to use in an application program, and they can offer more predictable results. For example, if the named constant `Unit` is defined as FIXED BINARY VALUE(1), it has the attributes FIXED BINARY(15) VALUE(1). If you simply use the digit 1, its attributes are FIXED DECIMAL(1,0). See Figure 1 on page 47 for other differences that can occur.

In addition, named constants allow you to parameterize your application, which makes it easier to debug and maintain.

Named constants can be declared for arithmetic data, string data, and for pointers and offsets. For arithmetic and string data and their attributes, see "String data and attributes" on page 34 and "Coded arithmetic data and attributes" on page 27, respectively. A named constant must be declared before it is used.

VALUE attribute:

```
►►──VALUE(restricted-expression)────────────────────►◄
```

restricted expression
>   The expression must evaluate to a scalar value. For information on restricted expressions see "Restricted expressions" on page 71.

Examples of named constants: Figure 1 shows named constants and the differences in attributes and precisions that can occur between named and unnamed constants.

Named constants can be used wherever a constant is required. They can also be

```
    Dcl A4 value(148) fixed bin,
        C4 value(261) fixed bin,
        Whole value(800) fixed bin;
    Dcl Notes (4) static,
        init(a4,  (Whole/4),   /* 148, 200  */
             c4,  (Whole*2));  /* 261, 1600 */


    /* note that "Head" gets length equal to length of VALUE   */


    Dcl Head char VALUE('Feel the Power of PL/I'); /* char(22) */
    Dcl Headsize fixed bin value(length(Head));      /* 22       */
    Dcl 1 Head1 static,
        2 * char(Headsize) initial(Head),        /* char(22) */
        2 * char(20) init(''),
        2 * char(5) init('Page '),
        2 Page_number pic 'zz9',
        2 * char(0);
    Dcl TwoHeads char(2*Headsize);                 /* char(44) */
    Dcl Page0 picture 'zz9' value(0);
    Dcl MyNullPtr ptr value(ptrvalue('ffff_ffff'xn));


    /* Differences in attributes/results of
       named and unnamed constants                               */


    Dcl Pi float bin value (3.1416);  /* is FLOAT BINARY(21) but ... */
    3.1416                            /* is FIXED DECIMAL(5,4)       */


    Dcl Unit fixed bin value(1);      /* is FIXED BINARY(15) but ... */
    1                                 /* is FIXED DECIMAL(1,0)       */
    1.0                               /* is FIXED DECIMAL(2,1)       */
    1B                                /* is FIXED BINARY(1)          */
    0000_0000_0000_001B               /* is FIXED BINARY(15)         */


    Dcl Title char(20) value('SCIDS'); /* is CHAR(20)     but ... */
    Dcl Title2 char     value('SCIDS');/* is CHAR(5)              */
    'SCIDS'                            /* is CHAR(5)              */
```

Figure 1. Named constants

used in restricted expressions that appear later in the program allowing evaluation of a dependent constant.

Program-control data types and attributes: This section describes program control data and associated attributes. Use program-control data to indicate values that control the execution of your program.

Label data and LABEL attribute: A *label* is a label constant or the value of a label variable.

Label data and attribute

```
►►──LABEL──────────────────────────────────────►◄
```

A label constant is a name written as the label prefix of a statement (other than PROCEDURE, ENTRY, PACKAGE, or FORMAT) so that during execution, program-control can be transferred to that statement through a reference to it. ("Statements" on page 16 discusses the syntax of the label prefix.) In the following line of code, for example, Abcde is a label constant.

```
  Abcde: Miles = Speed*Hours;
```

The labelled statement can be executed either by normal sequential execution of instructions or by using the GO TO statement to transfer control to it from some other point in the program.

A label variable can have another label variable or a label constant assigned to it. When such an assignment is made, the environment of the source label is assigned to the target. If you declare a static array of labels to have initial values, the array is treated as nonassignable.

A label variable used in a GO TO statement must have as its value a label constant that is used in a block that is active at the time the GO TO is executed. Consider the following example:

```
declare Lbl_x label;
Lbl_a:    statement;

:
Lbl_b:    statement;

:
Lbl_x = Lbl_a;

:
go to Lbl_x;
```

Lbl_a and Lbl_b are label constants, and Lbl_x is a label variable. By assigning Lbl_a to Lbl_x, the statement GO TO Lbl_x transfers control to the Lbl_a statement. Elsewhere, the program can contain a statement assigning Lbl_b to Lbl_x. Then, any reference to Lbl_x would be the same as a reference to Lbl_b. This value of Lbl_x is retained until another value is assigned to it.

If a label variable has an invalid value, detection of such an error is not guaranteed. In the following example, transfer is made to a particular element of the array Z based on the value of I.

```
go to Z(I);

:
Z(1):  if X = Y then return;

:
Z(2):  A = A + B + C * D;

:
Z(3):  A = A + 10;
```

If `Z(2)` is omitted, `GO TO Z(I)` when `I=2` raises the ERROR condition. `GO TO Z(I)` when `I < LBOUND(Z)` or `I > HBOUND(Z)` causes unpredictable results if the SUBSCRIPTRANGE condition is disabled.

Format data and FORMAT attribute: A format data item is a format constant or a format variable. A format constant is a name written as the label prefix of a FORMAT statement.

The FORMAT attribute specifies that the name being declared is a format variable.

```
►►─FORMAT──────────────────────────────────────────────────►◄
```

A name declared with the FORMAT attribute can have another format variable or a format constant assigned to it. When such an assignment is made, the environment of the source label is assigned to the target.

To maintain compatability between other PL/I compilers, format variables may be declared as label variables.

Consider the following example:

```
  Prntexe: format
             ( column(20),A(15), column(40),A(15), column(60),A(15) );
  Prntstf: format
             ( column(20),A(10), column(35),A(10), column(50),A(10) );
```

`Prntexe` and `Prntstf` are the format constants.

A second example indicates that **4** and **5** have the same effect as **2** , and **6** and **7** have the same effect as **3** .

```
1   dcl Print format;
2   put edit (X,Y,Z) (R(Prntexe) );
3   put edit (X,Y,Z) (R(Prntstf) );
4   Print = Prntexe;
5   put edit (X,Y,Z) (R(Print) );
6   Print = Prntstf;
7   put edit (X,Y,Z) (R(Print) );
```

VARIABLE attribute: The VARIABLE attribute establishes the name as a variable and is allowed only with ENTRY, FILE, and LABEL attributes.

```
►►─VARIABLE────────────────────────────────────────────────►◄
```
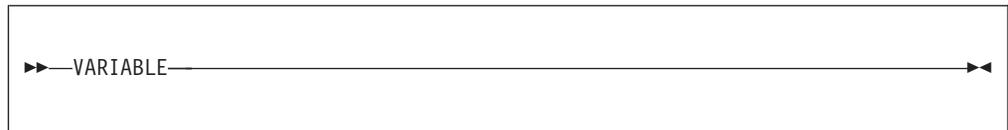
The VARIABLE attribute is implied if the name is a member of a structure or union, or if any of the following attributes are specified:
    Storage class attribute
    DIMENSION
    PARAMETER
    Alignment attribute
    INITIAL

VARIABLE

In the following declaration, `Account1` and `Account2` are file variables and `File1` and `File2` are file constants.

```
declare Account1 file variable,
        Account2 file automatic,
        File1 file,
        File2 file;
```

`File1` and `File2` can subsequently be assigned to `Account1` or to `Account2`.

Chapter 4. Expressions and references

This chapter discusses the various types of expressions and references.

An *expression* is a representation of a value. An expression can be one of the
following:

- A single constant, variable, or function reference
- Any combination of constants, variables, or function references, including
  operators and parentheses used in the combination.

An expression that contains operators is an *operational expression*. The constants
and variables of an operational expression are called *operands*. See "Operational
expressions" on page 55 for more information.

The following diagram shows the syntax for expressions and references.
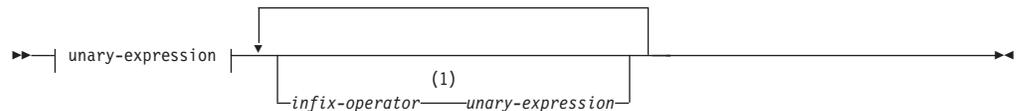
unary-expression:
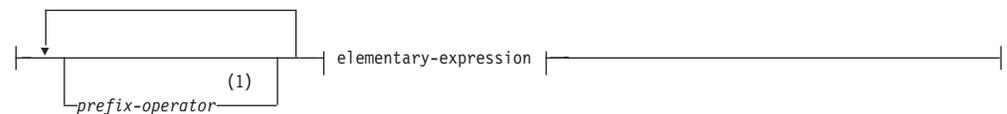
elementary-expression:

51

Expressions and references

reference:
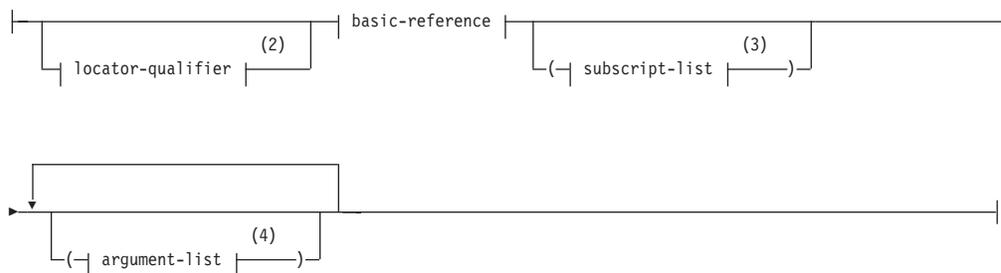
```
├──┬─────────────────────────┬── basic-reference ──┬──────────────────────────┬──────────────►
   │                    (2)  │                     │                     (3)  │
   └─ locator-qualifier ─────┘                     └─ (─┤ subscript-list ├─) ─┘


   ┌───────────────────────────────┐
   ▼                               │
►──┴──┬───────────────────────┬────┴────────────────────────────────────────────────────────┤
      │                  (4)  │
      └─ (─┤ argument-list ├─) ─┘
```

locator-qualifier:

```
       (2)
├──── reference ──┬─ -> ─┬──────────────────────────────────────────────────────────────────┤
                  ├─ => ─┤
                  └─ .  ─┘
```

basic-reference:

```
                                               (6)
├──┬────────────────────────────────┬── identifier ──────────────────────────────────────────┤
   │                           (5)  │
   └─┤ qualified-reference ├────────┘
```

subscript-list:

```
        (3)   ┌──── , ─────┐
              ▼            │
├─────────────┬─ expression ─┬──────────────────────────────────────────────────────────────┤
              └─ * ──────────┘
```

argument-list:

```
        (4)   ┌──── , ─────┐
              ▼            │
├─────────────┬─ expression ─┬──────────────────────────────────────────────────────────────┤
              └─ * ──────────┘
```

qualified-reference:

```
   (5)
├──── basic-reference ──┬──────────────────────────┬─ . ──────────────────────────────────────┤
                        └─ (─┤ subscript-list ├─) ──┘
```
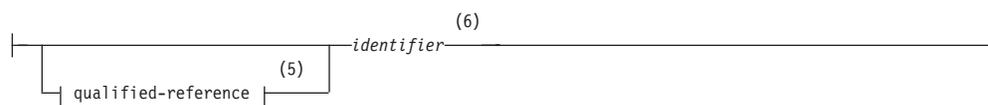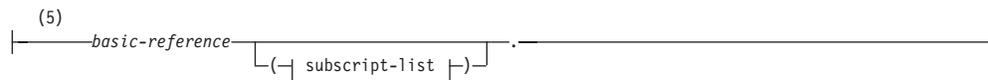
Notes:

1   Operators are shown in Table 6 on page 14.

2   Locator-qualifier is described under "Locator qualification" on page 231 and "Typed structure qualification" on page 141.

3   Subscripts are described under "Arrays" on page 167.

4   Arguments are described in "Passing arguments to procedures" on page 108.

5   Qualified-reference is described under "Structure/union qualification" on page 173.

6   Identifiers are described under "Identifiers" on page 13.

Any expression can be classified as an *element expression* (also called a scalar expression), an *array expression*, or a *structure expression*. Element variables and array variables can appear in the same expression.

An element expression
> represents a single value. This definition includes an elementary name within a structure or a union or a subscripted name that specifies a single element of an array.

An array expression
> represents an array of values. This definition includes a member of a structure or union that has the dimension attribute.

A structure expression
> represents a structured set of values.

Given the following example:
```
dcl A(10,10) bin fixed(31),
    B(10,10) bin fixed(31),
    1 Rate,
      2 Primary dec fixed(4,2),
      2 Secondary dec fixed(4,2),
    1 Cost(2),
      2 Primary dec fixed(4,2),
      2 Secondary dec fixed(4,2),
    C bin fixed(15),
    D bin fixed(15);
dcl Pi bin float value(3.1416);
```

These are element expressions:
```
Pi
27
C
C * D
A(3,2) + B(4,8)
Rate.Primary - Cost.Primary(1)
A(4,4) * C
Rate.Secondary / 4
A(4,6) * Cost.Secondary(2)
sum(A)
addr(Rate)
```

These are array expressions:
```
A
A + B
A * C - D
B / 10B
```

The syntax of many PL/I statements allows expressions, provided the result of the expression conforms with the syntax rules. Unless specifically stated in the text following the syntax specification, the unqualified term *expression* or *reference* refers to a scalar expression. For expressions other than a scalar expression, the type of expression is noted. For example, the term *array expression* indicates that a scalar expression is not valid.

An example of a structure expression is:
```
Rate = Rate*2
```

Order of evaluation

PL/I statements often contain more than one expression or reference. Except as described for specific instances (for example, the assignment statement), evaluation can be in any order, or (conceptually) at the same time.

For example:

```
dcl (X,Y,Z) entry returns(float), (F,G,H) float;
F = X( Y(G,H), Z(G,H) );
```

The functions `Y` and `Z` can change the value of the arguments passed to them. Hence, the value returned by `X` might be different depending on which function is invoked first. You should not presume that the first parameter is evaluated first. In some situations, it is more optimal to evaluate the last first.

Assuming that the INC function increments the value of the argument passed to it and returns the updated value, the example that follows could put out `B(1,2)` or `B(2,1)` depending on which subscript is evaluated first. You should not presume which subscript is evaluated first.

```
dcl B(2,2);
I = 0;
put list ( B( INC(I), INC(I) ) );
```

Targets

The results of an expression evaluation or of a conversion are assigned to a *target*. Targets can be variables, pseudovariables, or intermediate results.

Variables

In the case of an assignment, such as the statement:

```
A = B;
```

the target is the variable on the left of the assignment symbol (in this case A). Assignment to variables can also occur in stream I/O, DO, DISPLAY, and record I/O statements.

Pseudovariables

A pseudovariable represents a target field, for example:

```
declare A character(10),
        B character(30);
substr(A,6,5) = substr(B,20,5);
```

In this assignment statement, the SUBSTR built-in function extracts a substring of length 5 from the string `B`, beginning with the twentieth character. The SUBSTR pseudovariable indicates the location, within string `A`, that is the target. Thus, the last 5 characters of `A` are replaced by characters 20 through 24 of `B`. The first 5 characters of `A` remain unchanged.

Pseudovariables are discussed in Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365.

Intermediate results

When an expression is evaluated, the target attributes usually are partly derived from the source, partly from the operation being performed, and partly from the

attributes of a second operand. Some defaults may be used, and some implementation restrictions (for example, maximum precision) and conventions exist. An intermediate result can undergo conversion if a further operation is performed. After an expression is evaluated, the result can be further converted for assignment to a variable or pseudovariable. These conversions follow the same rules as the conversion of programmer-defined data, for example:

```
declare A character(8),
        B fixed decimal(3,2),
        C fixed binary(10);
  A = B + C;
```

During the evaluation of the expression `B + C` and during the assignment of that result, there are four different results:

1. The intermediate result to which the converted binary equivalent of `B` is assigned

2. The intermediate result to which the binary result of the addition is assigned

3. The intermediate result to which the converted decimal fixed-point equivalent of the binary result is assigned

4. `A`, the final destination of the result, to which the converted character equivalent of the decimal fixed-point representation of the value is assigned

The attributes of the first result are determined from the attributes of the source `B`, from the operator, and from the attributes of the other operand. If one operand of an arithmetic infix operator is binary, the other is converted to binary before evaluation.

The attributes of the second result are determined from the attributes of the source (`C` and the converted representation of `B`).

The attributes of the third result are determined in part from the source (the second result) and in part from the attributes of the eventual target `A`. The only attribute determined from the eventual target is DECIMAL (a binary arithmetic representation must be converted to decimal representation before it can be converted to a character value).

The attributes of `A` are known from the DECLARE statement.

Operational expressions

An operational expression consists of one or more single operations. A single operation is either a *prefix operation* (an operator preceding a single operand) or an *infix operation* (an operator between two operands). The two operands of any infix operation normally should be the same data type when the operation is performed.

The operands of an operation in a PL/I expression are converted, if necessary, to the same data type before the operation is performed. Detailed rules for conversion can be found in Chapter 5, "Data conversion," on page 73.

There are few restrictions on the use of different data types in an expression. However, these mixtures imply conversions. If conversions take place at run time, the program takes longer to run. Also, conversion can result in loss of precision. When using expressions that mix data types, you should understand the relevant conversion rules.

There are five classes of operations—pointer, arithmetic, bit, comparison, and concatenation.

Pointer Operations

The following pointer operations can be used:

- Add an expression to or subtract an expression from a pointer expression. The expression type must be computational. If necessary, the nonpointer operand is converted to FIXED BINARY(31,0), for example:

```
Ptr1 = Ptr1 - 16;
Ptr2 = Ptr1 + (I*J);
```

  You can also use the built-in function, POINTERADD, to perform these operations. You must use POINTERADD if the result is used as a locator reference, for example:

```
(Ptr1 + 16) -> Based_ptr          is invalid

pointeradd(Ptr1,16) -> Based_ptr    is valid
```

- Subtract two pointers to obtain the logical difference. The result is a FIXED BINARY(31,0) value.

```
Bin31 = Ptr2 - Ptr1;
```

- Compare pointer expressions using infix operators.

```
if Ptr2 > Ptr1 then
    Bin31 = Ptr2 - Ptr1;
```

- Use pointer expressions in arithmetic contexts using the built-in function, BINARYVALUE.

```
Bin31 = Bin31 + binaryvalue(Ptr1);
```

- Use computational expressions in pointer contexts using the built-in function, POINTERVALUE.

```
dcl 1 Cvtprt pointer based(pointervalue(16));
dcl 1 Cvt based(Cvtptr),
      2 Cvt ...;
```

  If necessary, the expressions are converted to FIXED BINARY(31,0).

A PL/I block can use pointer arithmetic to access any element within a structure or an array variable. However, the block must be passed the containing structure or array variable, or have the referenced aggregate within its name scope.

Arithmetic operations

An arithmetic operation is specified by combining operands with one of these operators:

```
 +   -   *   /   **
```

The plus sign and the minus sign can appear as prefix operators or as infix operators. All other arithmetic operators can appear only as infix operators. (Arithmetic operations can also be specified by the ADD, SUBTRACT, DIVIDE, and MULTIPLY built-in functions.)

Prefix operators can precede and be associated with any of the operands of an infix operation. For example, in the expression A*-B, the minus sign indicates that the value of A is multiplied by -1 times the value of B.

More than one prefix operator can precede and be associated with a single variable. More than one positive prefix operator has no cumulative effect, but two negative prefix operators have the same effect as a single positive prefix operator.

Data conversion in arithmetic operations

The two operands of an arithmetic operation can differ in type, base, mode, precision, and scale. When they differ, conversion takes place as described below. (For coded arithmetic operands, you can also determine conversions using Table 13 on page 58. Each operand is converted to the type, base, and mode of the result. It is not necessarily converted to the result's precision and scale.)

Note: Scaled FIXED BINARY operands are converted to scaled FIXED DECIMAL before any operations on them are performed.

Type: Character operands are converted to FIXED DECIMAL(N,0). Bit operands are converted to FIXED BINARY(M,0). (Refer to "Limits," on page 533 for the maximums.) Numeric character operands are converted to DECIMAL with scale and precision determined by the picture-specification.

Graphic and widechar variables and strings are allowed in all computational contexts. If conversion is necessary, the rules followed are the same as for character.

The result of an arithmetic operation is always in coded arithmetic form. Type conversion is the only conversion that can take place in an arithmetic prefix operation.

Base: If the bases of the two operands differ, the decimal operand is converted to binary.

Mode: If the modes of the two operands differ, the real operand is converted to complex mode by acquiring an imaginary part of zero with the same base, scale, and precision as the real part. The exception to this is in the case of exponentiation when the second operand (the exponent of the operation) is fixed-point real with a scaling factor of zero. In such a case, conversion is not necessary.

Precision: If only precisions and/or scaling factors vary, type conversion is not necessary.

Scale: If the scales of the two operands differ, the fixed-point operand is converted to floating-point scale. The exception to this is in the case of exponentiation when the first operand is of floating-point scale and the second operand (the exponent of the operation) is fixed-point with a scaling factor of zero, that is, an integer or a variable that has been declared with precision (p,0). In such a case, conversion is not necessary, but the result is floating-point.

If both operands of an exponentiation operation are fixed-point, conversions can occur in one of the following ways:
- Both operands are converted to floating-point if the exponent has a precision other than (p,0).
- The first operand is converted to floating-point unless the exponent is an unsigned integer.
- The first operand is converted to floating-point if precisions indicate that the result of the fixed-point exponentiation would exceed the maximum number of digits allowed.

Results of arithmetic operations

After any necessary conversion of the operands in an expression has been carried out, the arithmetic operation is performed and a result is obtained. This result can

Results of arithmetic operations

be the value of the expression, or it can be an intermediate result upon which further operations are to be performed, or a condition can be raised.

Table 13 and Table 14 on page 59 show the attributes and precisions that result from various arithmetic operations.

Table 18 on page 63 shows the attributes of the result for the special cases of exponentiation noted in the right-hand columns of Table 13 and Table 14 on page 59.

Under the compiler option RULES(ANS), if one operand is scaled FIXED DECIMAL and the other is FIXED BINARY, the FIXED BINARY value is converted to FIXED DECIMAL. Table 15 on page 60 shows the attributes and precisions that result for this case under compiler option RULES(ANS). For more information on the RULES compiler option, see the Programming Guide.

Table 13. Results of arithmetic operations for one or more FLOAT operands

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multipli-cation Precision	Division Precision	Attributes of the Result for Exponentiation
FLOAT DECIMAL (p1)	FLOAT DECIMAL (p2)	FLOAT DECIMAL (p)	$p = \text{MAX}(p1,p2)$			FLOAT DECIMAL (p) (unless special case C applies) $p = \text{MAX}(p1,p2)$
FLOAT DECIMAL (p1)	FLOAT DECIMAL (p2,q2)					
FIXED DECIMAL (p1,q1)	FLOAT DECIMAL (p2)					
FLOAT BINARY (p1)	FLOAT BINARY (p2)	FLOAT BINARY (p)				FLOAT BINARY (p) (unless special case C applies) $p = \text{MAX}(p1,p2)$
FLOAT BINARY (p1)	FIXED BINARY (p2,q2)					
FIXED BINARY (p1,q1)	FLOAT BINARY (p2,q2)					
FIXED DECIMAL (p1,q1)	FLOAT BINARY (p2)	FLOAT BINARY (p)	$p = \text{MAX}(\text{CEIL}(p1*3.32),p2)$			FLOAT BINARY (p) (unless special case A or C applies) $p = \text{MAX}(\text{CEIL}(p1*3.32),p2)$
FLOAT DECIMAL (p1)	FLOAT BINARY (p1,q2)					
FLOAT DECIMAL (p1)	FLOAT BINARY (p2)					

Table 13. Results of arithmetic operations for one or more FLOAT operands (continued)

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multiplication Precision	Division Precision	Attributes of the Result for Exponentiation
FIXED BINARY (p1,q1)	FLOAT DECIMAL (p2)	FLOAT BINARY (p)	p = MAX(p1,CEIL(p2*3.32))			FLOAT BINARY (p) (unless special case B or C applies) p = MAX(p1,CEIL(p2*3.32))
FLOAT BINARY (p1)	FIXED DECIMAL (p2,q2)					
FLOAT BINARY (p1)	FLOAT DECIMAL (p2)					

Notes:

1.   Special cases of exponentiation are described in Table 18 on page 63.
2.   For a table of CEIL(N*3.32) values, see Table 22 on page 76.

Table 14. Results of arithmetic operations between two unscaled FIXED operands under RULES(ANS)

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multiplication Precision	Division Precision	Attributes of the Result for Exponentiation
FIXED DECIMAL (p1,0)	FIXED DECIMAL (p2,0)	FIXED DECIMAL (p,q)	p = 1 +MAX(p1,p2) q = 0	p = 1 +p1+p2 q = 0	p = N q = N–p1	FLOAT DECIMAL (p) (unless special case A applies) p = MAX(p1,p2)
FIXED BINARY (p1,0)	FIXED BINARY (p2,0)	FIXED BINARY (p,0)	p = 1 +MAX(p1–q1, p2–q2) +q q = 0	p = 1+p1 +p2 q = 0	p = M q = 0	FLOAT BINARY (p) (unless special case B applies) p = MAX(p1,p2)
FIXED DECIMAL (p1,0)	FIXED BINARY (p2,0)	FIXED BINARY (p,0)	p = 1 +MAX(r,p2) q = 0	p = 1 +r+p2 q = 0	p = M q = 0	FLOAT BINARY (p) (unless special case A or C applies) p = MAX(CEIL (p1*3.32),p2)
FIXED BINARY (p1,0)	FIXED DECIMAL (p2,0)	FIXED BINARY (p,0)	p = 1 +MAX(p1,t) q = 0	p = 1 +p1+t q = 0	p = M q = 0	FLOAT BINARY (p) (unless special case A or C applies) p = MAX(CEIL (p1*3.32),p2)

M is the maximum precision for FIXED BINARY.
N is the maximum precision for FIXED DECIMAL.
r = 1 + CEIL(p1*3.32)
s = CEIL(ABS(q1*3.32)) * SIGN(q1)

t = 1 + CEIL(p2*3.32)
u = CEIL(ABS(q2*3.32)) * SIGN(q2)
v = CEIL(p2/3.32)
w = CEIL(p1/3.32)

Results of arithmetic operations

Notes:

The scaling factor must be in the range −128 through +127.

1. Special cases of exponentiation are described in Table 18 on page 63.
2. For a table of CEIL(N*3.32) values, see Table 22 on page 76.
3. Under RULES(ANS) a divide with unscaled FIXED operands can produce a scaled result only if both operands are FIXED DECIMAL.

Table 15. Results of arithmetic operations between two scaled FIXED operands under RULES(ANS)

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multiplication Precision	Division Precision	Attributes of the Result for Exponentiation
FIXED DECIMAL (p1,q1)	FIXED DECIMAL (p2,q2)	FIXED DECIMAL (p,q)	$p = 1 + MAX(p1{-}q1, p2{-}q2) + q$ $q = MAX(q1,q2)$	$p = 1 + p1 + p2$ $q = q1 + q2$	$p = N$ $q = N{-}p1{+}q1{-}q2$	FLOAT DECIMAL (p) (unless special case A applies) $p = MAX(p1,p2)$
FIXED DECIMAL (p1,q1)	FIXED BINARY (p2,0)	FIXED DECIMAL (p,q)	$p = 1 + MAX(p1{-}q1,v) + q$ $q = q1$	$p = 1 + p2 + v$ $q = q1$	$p = N$ $q = N{-}q1$	FLOAT BINARY (p) (unless special case A or C applies) $p = MAX(CEIL(p1{*}3.32),p2)$
FIXED BINARY (p1,0)	FIXED DECIMAL (p2,q2)	FIXED DECIMAL (p,q)	$p = 1 + MAX(p2{-}q2,w) + q$ $q = q2$	$p = 1 + p2 + w$ $q = q1$	$p = N$ $q = N{-}q2$	FLOAT BINARY (p) (unless special case A or C applies) $p = MAX(CEIL(p1{*}3.32),p2)$

M is the maximum precision for FIXED BINARY.
N is the maximum precision for FIXED DECIMAL.
$r = 1 + CEIL(p1{*}3.32)$
$s = CEIL(ABS(q1{*}3.32)) * SIGN(q1)$

$t = 1 + CEIL(p2{*}3.32)$
$u = CEIL(ABS(q2{*}3.32)) * SIGN(q2)$
$v = CEIL(p2/3.32)$
$w = CEIL(p1/3.32)$

Notes:

The scaling factor must be in the range −128 through +127.

1. Special cases of exponentiation are described in Table 18 on page 63.
2. For a table of CEIL(N*3.32) values, see Table 22 on page 76.
3. Under RULES(ANS), scaled FIXED BINARY is not allowed.

Table 16. Results of arithmetic operations between two FIXED operands under RULES(IBM)

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multiplication Precision	Division Precision	Attributes of the Result for Exponentiation
FIXED DECIMAL (p1,q1)	FIXED DECIMAL (p2,q2)	FIXED DECIMAL (p,q)	p = 1 +MAX(p1–q1, p2–q2) +q q = MAX(q1,q2)	p = 1 +p1+p2 q = q1+q2	p = N q = N–p1+q1–q2	FLOAT DECIMAL (p) (unless special case A applies) p = MAX(p1,p2)
FIXED BINARY (p1,q1)	FIXED BINARY (p2,q2)	FIXED BINARY (p,q)	p = 1 +MAX(p1–q1, p2–q2) +q q = MAX(q1,q2)	p = 1 +p1+p2 q = q1+q2	p = M q = M–p1 +q1–q2	FLOAT BINARY (p) (unless special case B applies) p = MAX(p1,p2)
FIXED DECIMAL (p1,q1)	FIXED BINARY (p2,q2)	FIXED BINARY (p,q)	p = 1 +MAX(r–s, p2–q2)+q q = MAX(s,q2)	p = 1+r +p2 q = s+q2	p = M q = M–r +s–q2	FLOAT BINARY (p) (unless special case A or C applies) p =MAX(CEIL((p1*3.32),p2)
FIXED BINARY (p1,q1)	FIXED DECIMAL (p2,q2)	FIXED BINARY (p,q)	p = 1 +MAX(p1– q1,t–u) +q q = MAX(s,q1,u)	p = 1 +p1+t q = q1+u	p = M q = M–p1 +q1–u	FLOAT BINARY (p) (unless special case A or C applies) p = MAX(p1, CEIL(p2*3.32))

M is the maximum precision for FIXED BINARY.
N is the maximum precision for FIXED DECIMAL.
r = 1 + CEIL(p1*3.32)
s = CEIL(ABS(q1*3.32)) * SIGN(q1)

t = 1 + CEIL(p2*3.32)
u = CEIL(ABS(q2*3.32)) * SIGN(q2)
v = CEIL(p2/3.32)
w = CEIL(p1/3.32)

Notes:

The scaling factor must be in the range –128 through +127.

1. Special cases of exponentiation are described in Table 18 on page 63.
2. For a table of CEIL(N*3.32) values, see Table 22 on page 76.

Consider the expression:

```
A * B + C
```

The operation `A * B` is performed first, to give an intermediate result. Then the value of the expression is obtained by performing the operation (intermediate result) + `C`.

PL/I gives the intermediate result attributes the same way it gives attributes to any variable. The attributes of the result are derived from the attributes of the two operands (or the single operand in the case of a prefix operation) and the operator involved. The way the attributes of the result are derived is further explained under "Targets" on page 54.

The ADD, SUBTRACT, MULTIPLY, and DIVIDE built-in functions allow you to override the implementation precision rules for addition, subtraction, multiplication, and division operations.

Results of arithmetic operations

FIXED division: FIXED division can result in overflows or truncation. For example, the result of evaluating the expression:

```
25+1/3
```

is undefined and the FIXEDOVERFLOW condition is raised because FIXED division results in a value of maximum implementation defined precision.

For the following expression, however:

```
25+01/3
```

The result is 25.3333333333333 (when the maximum precision is 15) because constants have the precision with which they are written. The results of the two evaluations are reached as shown in Table 17:

Table 17. Comparison of FIXED division and constant expressions

Item	Precision	Result
1	(1,0)	1
3	(1,0)	3
1/3	(15,14)	0.33333333333333
25	(2,0)	25
25+1/3	(15,14)	undefined (truncation on left; FIXEDOVERFLOW is raised when the maximum precision is 15)
01	(2,0)	01
3	(1,0)	3
01/3	(15,13)	00.3333333333333
25	(2,0)	25
25+01/3	(15,13)	25.3333333333333

The PRECISION built-in function can also be used. For example:

```
25+prec(1/3,15,13)
```

Note: Named constants are recommended for situations that require exact precisions.

Using exponentiation

The following table describes how exponentiation is handled in PL/I.

Table 18. Special cases for exponentiation

Case	First Operand	Second Operand	Attributes of Result
A	FIXED DECIMAL (p1,q1)	Integer with value n	FIXED DECIMAL (p,q) (provided $p <= N$) where $p = (p1 + 1)*n-1$ and $q = q1*n$
B	FIXED BINARY (p1,q1)	Integer with value n	FIXED BINARY (p,q) (provided $p <= M$) where $p = (p1 + 1)*n-1$ and $q = p1*n$
C	FLOAT (p1)	FIXED (p2,0)	FLOAT (p1) with base of first operand

Special cases of x**y in real/complex modes:

Real mode:
>**Complex mode:**

If x=0 and y>0,
>**result is 0**. If x=0, and real part of y>0 and imaginary part of y=0, result is 0.

If x=0 and y<=0,
>**ERROR condition is raised**. If x=0 and real part of y<=0 or imaginary part of y ¬=0, ERROR condition is raised.

If x<0 and y not FIXED (p,0),
>**ERROR condition is raised**. If x¬=0 and real and imaginary parts of y=0, result is 1.

Bit operations

A bit operation is specified by combining operands with one of the following logical operators:

>¬　　&　　|

The *not/exclusive-or* symbol (¬), can be used as a prefix or infix operator. The *and* (&) symbol and the *or* (|) symbol, can be used as infix operators only. (The operators have the same function as in Boolean algebra.)

Operands of a bit operation are converted, if necessary, to bit strings before the operation is performed. If the operands of an infix operation do not have the same length, the shorter is padded on the right with '0'B.

The result of a bit operation is a bit string equal in length to the length of the operands.

Bit operations are performed on a bit-by-bit basis. Table 19 on page 64 illustrates the result for each bit position for each of the operators. Table 20 on page 64 shows some examples of bit operations.

Bit operations

Table 19. Bit operations

A	B	¬A	¬B	A&B	A|B	A¬B
1	1	0	0	1	1	0
1	0	0	1	0	1	1
0	1	1	0	0	1	1
0	0	1	1	0	0	0

Table 20. Bit operation examples

For these operands and values	This operation	Yields this result
A = '010111'B B = '111111'B C = '110'B D = 5	¬ A	'101000'B
	¬ C	'001'B
	C & B	'110000'B
	A | B	'111111'B
	A ¬ B	''101000''B
	A ¬ C	'100111'B
	C | B	'111111'B
	A | (¬C)	'011111'B
	¬((¬C)|(¬B))	'110111'B
	SUBSTR(A,1,1)|(D=5)	'1'B

BOOL built-in function

In addition to the *not*, *exclusive-or*, *and*, and *or* operations using the operators ¬, &, and |, Boolean operations can be performed using the BOOL built-in function discussed in "BOOL" on page 394.

Comparison operations

A comparison operation is specified by combining operands with one of the following infix operators:

<	¬<	<=	=	¬=	>=	>	¬>

The result of a comparison operation is always a bit string of length 1. The value is '1'B if the relationship is true, or '0'B if the relationship is false.

Comparisons are defined as follows:

Algebraic　is the comparison of signed arithmetic values in coded arithmetic form. If operands differ in base, scale, precision, or mode, they are converted in a manner analogous to arithmetic operation conversions. Numeric character data is converted to coded arithmetic before comparison. Only the operators = and ¬= are valid for comparison of operands that are complex numbers.

Character　is a left-to-right, character-by-character comparison of characters according to the binary value of the bytes.

Bit　is a left-to-right, bit-by-bit comparison of binary digits.

Graphic   is a left-to-right, symbol-by-symbol comparison of DBCS characters. The comparison is based on the binary values of the DBCS characters.

Widechar   is a left-to-right, widechar-by-widechar comparison of characters according to the binary value of the byte-pairs.

Ordinal data is a comparison of ordinals of the same type using relational operators.

Pointer and offset data
is a comparison of pointer and offset values containing any relational operators. However, the only conversion that can take place is offset to pointer.

Program-control data
is a comparison of the internal coded forms of the operands. Only the comparison operators = and ¬= are allowed; area variables cannot be compared. No type conversion can take place; all type differences between operands for program-control data comparisons are in error.

Comparisons are equal for the following operands:

Entry   In a comparison operation, it is not an error to specify an entry variable whose value is an entry point of an inactive block. Entry names on the same PROCEDURE or ENTRY statement do not compare equal.

Format   Format labels on the same FORMAT statement compare equal.

File   If the operands represent file values, all of whose parts are equal.

Label   Labels on the same statement compare equal. In a comparison operation, it is not an error to specify a label variable whose value is a label constant used in a block that is no longer active.

The label on a compound statement does not compare equal with that on any label contained in the body of the compound statement.

If the operands of a computational data comparison have data types that are appropriate to different types of comparison, the operand of the lower precedence is converted to conform to the comparison type of the other. The precedence of comparison types is (1) algebraic (highest), (2) widechar, (3) graphic, (4) character, (5) bit. For example, if a bit string is compared with a fixed decimal value, the bit string is converted to fixed binary for algebraic comparison with the decimal value. The decimal value is also converted to fixed binary.

In the comparison of strings of unequal lengths, the shorter string is padded on the right. This padding consists of:
- Blanks in a character comparison
- '0'B in a bit comparison
- A graphic (DBCS) blank in a graphic comparison.
- A widechar blank ('0020'wx) in a widechar comparison.

The following example shows a comparison operation in an IF statement:

Comparison operations

```
if A = B
    then action-if-true;
    else action-if-false;
```

The evaluation of the expression A = B yields either '1'B, for true, or '0'B, for false.

In the following assignment statement:
```
  X = A <= B;
```

the value '1'B is assigned to X if A is less than B; otherwise, the value '0'B is assigned.

In the following assignment statement:
```
  X = A = B;
```

the first equal symbol is the assignment symbol; the second equal symbol is the comparison operator. The value '1'B is assigned to X if A is equal to B; otherwise, the value '0'B is assigned.

An example of comparisons in an arithmetic expression is:
```
  (X<0)*A + (0<=X & X<=100)*B + (100<X)*C
```

The value of the expression is A, B, or C and is determined by the value of X.

Concatenation operations

A concatenation operation is specified by combining operands with the concatenation infix operator:

∥

Concatenation signifies that the operands are to be joined in such a way that the last character, bit, graphic or widechar of the operand to the left immediately precedes the first character, bit, graphic or widechar of the operand to the right, with nothing intervening.

The concatenation operator can cause conversion to a string type because concatenation can be performed only upon strings—either character, bit, graphic or widechar. The results differ according to the setting of the RULES compiler option:

Results under RULES(IBM)
When you specify RULES(IBM), the concatenation operator behaves as folows:
- If either operand is widechar, the result is widechar.
- Else, if either operand is graphic, the result is graphic.
- Else, if either operand is bit or binary, the result is bit.
- Otherwise the result is character.

For example:
```
  dcl B bin(4)  initial(4),
      C bit(1)  initial('1'b);
  put skip list (B ∥ C);

  /*  Produces '01001'  not 'bbb41'  */
```

Results under RULES(ANS): When you specify RULES(ANS), the concatenation operator behaves as follows:
- If either operand is widechar, the result is widechar.
- Else, if either operand is graphic, the result is graphic.

- Else, if both operands are bit, the result is bit.
- Otherwise the result is character.

Consider this example:

```
dcl B bin(4)  initial(4),
    C bit(1)  initial('1'b);
put skip list (B ∥ C);

/*  Produces 'bbb41', not '01001'  */
```

The result of a concatenation operation is a string whose length is equal to the sum of the lengths of the two operands, and whose type (that is, character, bit, graphic or widechar) is the same as that of the two operands.

If an operand requires conversion for concatenation, the result depends upon the length of the string to which the operand is converted.

For these operands and values	This operation	Yields this result
A = '010111'B B = '101'B C = 'xy,Z' D = 'aa/BB'	A ∥ B	'010111_101'B
	A ∥ A ∥ B	'010111_010111_101'B
	C ∥ D	'xy,Zaa/BB'
	D ∥ C	'aa/BBxy,Z'
	B ∥ D	'101aa/BB'

In the last example, the bit string '101'B is converted to the character string '101' before the concatenation is performed. The result is a character string.

Combinations of operations

Different types of operations can be combined within the same operational expression. Any combination can be used.

For example:

```
declare Result bit(3),
   A fixed decimal(1),
   B fixed binary (3),
   C character(2), D bit(4);
Result = A + B < C & D;
```

Each operation within the expression is evaluated according to the rules for that kind of operation, with necessary data conversions taking place before the operation is performed, as follows:

- The decimal value of A is converted to binary base.
- The binary addition is performed, adding A and B.
- The binary result is compared with the converted binary value of C.
- The bit result of the comparison is extended to the length of the bit variable D, and the & operation is performed.
- The result of the & operation, a bit string of length 4, is assigned to Result without conversion, but with truncation on the right.

The expression in this example is evaluated operation-by-operation, from left to right. The order of evaluation, however, depends upon the priority of the operators appearing in the expression.

Priority of operators

The priority of the operators in the evaluation of expressions is shown in Table 21.

Table 21. Priority of operations and guide to conversions

Priority	Operator	Type of Operation	Remarks
1	**	Arithmetic	Result is in coded arithmetic form
	prefix +, −	Arithmetic	No conversion is required if operand is in coded arithmetic form
			Operand is converted to FIXED DECIMAL if it is a CHARACTER string or numeric character (PICTURE) representation of a fixed-point decimal number
			Operand is converted to FLOAT DECIMAL if it is a numeric character (PICTURE) representation of a floating-point decimal number
			Operand is converted to FIXED BINARY if it is a BIT string
	prefix ¬	Bit string	All non-BIT data converted to BIT
2	*, /	Arithmetic	Result is in coded arithmetic form
3	infix +, −	Arithmetic	Result is in coded arithmetic form
4	‖	Concatenation	Refer to "Results under RULES(ANS)" on page 66 and "Results under RULES(IBM)" on page 66
5	<, ¬<, <=, =, ¬=, >=, >, ¬>	Comparison	Result is always either '1'B or '0'B
6	&	Bit string	All non-BIT data converted to BIT
7	|	Bit string	All non-BIT data converted to BIT
	infix ¬	Bit string	All non-BIT data converted to BIT

Notes:

1. The operators are listed in order of priority, group 1 having the highest priority and group 7 the lowest. All operators in the same priority group have the same priority. For example, the exponentiation operator ** has the same priority as the prefix + and prefix − operators and the *not* operator ¬.

2. For priority group 1, if two or more operators appear in an expression, the order of priority is right to left within the expression; that is, the rightmost exponentiation or prefix operator has the highest priority, the next rightmost the next highest, and so on. For all other priority groups, if two or more operators in the same priority group appear in an expression, their order or priority is their order left to right within the expression.

The order of evaluation of the expression

```
A + B < C & D
```

is the same as if the elements of the expression were parenthesized as

```
(((A + B) < C) & D)
```

The order of evaluation (and, consequently, the result) of an expression can be changed through the use of parentheses. Expressions enclosed in parentheses are evaluated first, to a single value, before they are considered in relation to surrounding operators.

The above expression, for example, might be changed as follows:

```
(A + B) < (C & D)
```

The value of `A` converts to fixed-point binary, and the addition is performed, yielding a fixed-point binary result (result_1). The value of `C` converts to a bit string (if valid for such conversion) and the *and* operation is performed. At this point, the expression is reduced to:

```
Result_1 < Result_2
```

`Result_2` is converted to binary, and the algebraic comparison is performed, yielding a bit string of length 1 for the entire expression.

The priority of operators is defined only within operands (or sub-operands). Consider the following example:

```
A + (B < C) & (D || E ** F)
```

In this case, PL/I specifies only that the exponentiation occurs before the concatenation. It does not specify the order of the evaluation of (`D||E ** F`) in relation to the evaluation of the other operand (`A + (B < C)`).

Any operational expression (except a prefix expression) must eventually be reduced to a single infix operation. The operands and operator of that operation determine the attributes of the result of the entire expression. In the following example, the `&` operator is the operator of the final infix operation.

```
A + B < C & D
```

The result of the evaluation is a bit string of length 4.

In the next example, because of the use of parentheses, the operator of the final infix operation is the comparison operator:

```
(A + B) < (C & D)
```

The evaluation yields a bit string of length 1.

Array expressions

Array expressions are allowed as:
- the source in an assignment or in multiple assignments
- the argument to the ALL, ANY, POLY, PROD or SUM built-in functions
- an argument to a user procedure and function, as long as the associated parameter is not a string of unknown length
- an item in the data-lists of PUT LIST and PUT EDIT statements

Evaluation of an array expression yields an array result. All operations performed on arrays are performed element-by-element, in row-major order. Therefore, all arrays referred to in an array expression must have the same number of dimensions, and each dimension must be of identical bounds.

Array expressions can include operators (both prefix and infix), element variables, and constants. The rules for combining operations and for data conversion of operands are the same as for element operations.

Prefix operators and arrays

The operation of a prefix operator on an array produces an array of identical bounds. Each element of this array is the result of the operation performed on each element of the original array. For example:

```
If A is the array      5   3  -9
                       1   2   7
                       6   3  -4

then -A is the array  -5  -3   9
                      -1  -2  -7
                      -6  -3   4
```

Infix operators and arrays

Infix operations that include an array variable as one operand can have an element or another array as the other operand.

Array-and-element operations

The result of an expression with an element, an array, and an infix operator is an array with bounds identical to the original array. Each element of the resulting array is the result of the operation between each corresponding element of the original array and the single element. For example:

```
If A is the array        5  10   8
                        12  11   3

then A*3 is the array   15  30  24
                        36  33   9

and 9 > A is the array of   1   0   1
bit strings of length 1     0   0   1
```

The element of an array-element operation can be an element of the same array. Consider the following assignment statement:

```
A = A * A(1,2);
```

Again, using the above values for A, the newly assigned value of A would be:

```
  50    100   800
1200   1100   300
```

That is, the value of A(1,2) is fetched again.

Array-and-array operations

If the two operands of an infix operator are arrays, the arrays must have the same number of dimensions, and corresponding dimensions must have identical lower bounds and identical upper bounds. The result is an array with bounds identical to those of the original arrays; the operation is performed upon the corresponding elements of the two original arrays. For example:

```
If A is the array          2   4   3
                           6   1   7
                           4   8   2

and if B is the array      1   5   7
                           8   3   4
                           6   3   1

then A+B is the array      3   9  10
                          14   4  11
                          10  11   3

and A*B is the array       2  20  21
                          48   3  28
                          24  24   2

and A>B is the array of    1   0   0
bit strings of length 1    0   0   1
                           0   1   1
```

Structure expressions

Structure expressions, unlike structure references, are allowed only in assignments and as arguments to procedures or functions, as long as the associated parameter has constant extents.

All structure variables appearing in a structure expression must have identical structuring, which means:

- The structures must have the same minor structuring and the same number of contained elements and arrays.
- The positioning of the elements and arrays within the structure (and within the minor structures, if any) must be the same.
- Arrays in corresponding positions must have identical bounds.

Restricted expressions

Where PL/I requires a (possibly signed) constant, a *restricted expression* can be used. A restricted expression is an expression whose value is calculated at compile time and used as a constant. For example, you can use expressions to define constants required for:
- Extents in static, parameter, and based declarations
- Extents in entry descriptions
- Values and iteration factors to be used in static initialization

A restricted expression is identical to a normal expression but requires that each operand be:

- A constant or a named constant. A named constant must be declared before it is used.

- A built-in function applied to a restricted expression(s), where the built-in function is from the following categories:

  - String-handling

  - Arithmetic (except RANDOM)

  - Mathematical

  - Floating-point inquiry

  - Floating-point manipulation

  - Integer manipulation

  - Precision-handling

  - Array-handling functions DIMENSION, LBOUND, and HBOUND

  - Storage-control functions BINARYVALUE, LENGTH, NULL, OFFSETVALUE, POINTERVALUE, SIZE, STORAGE, and SYSNULL

- Type functions BIND, CAST, FIRST, LAST, RESPEC and SIZE

Examples

```
dcl Max_names fixed bin value (1000),
    Name_size fixed bin value (30),
    Addr_size fixed bin value (20),
    Addr_lines fixed bin value (4);
dcl 1 Name_addr(Max_names),
      2 Name char(Name_size),
      2 * union,
        3 Address char(Addr_lines*Addr_size), /* address    */
        3 addr(Addr_lines) char(Addr_size),
      2 * char(0);
```

Restricted expressions

```
dcl One_Name_addr char(size(Name_addr(1)));   /* 1 name/addr*/
dcl Two_Name_addr char(length(One_Name_addr)
                       *2);                    /* 2 name/addrs  */
dcl Name_or_addr char(max(Name_size,Addr_size)) based;

dcl Ar(10) pointer;
dcl Ex     entry( dim(lbound(Ar):hbound(Ar)) pointer);
dcl Identical_to_Ar( lbound(Ar):hbound(Ar) ) pointer;
```

If you change the value of any of the named constants in the example, all of the
dependent declarations are automatically reevaluated.

Chapter 5. Data conversion

This chapter discusses data conversions for computational data. PL/I converts data when a data item with a set of attributes is assigned to another data item with a different set of attributes. In this chapter, *source* refers to the data item to be converted, and *target* refers to the attributes to which the source is converted. Topics discussed for these data conversions include:

Built-in functions
String lengths
Arithmetic precision
Mode
Source-to-target rules

Examples of data conversion are included at the end of the chapter.

Data conversion for locator data is discussed in "Locator conversion" on page 230.

Conversion of the value of a computational data item can change its internal representation, precision or mode (for arithmetic values), or length (for string values). The tables that follow summarize the circumstances that can cause conversion to other attributes.

Case	Target Attributes
Assignment	Attributes of variable on left of assignment symbol
Operand in an expression	Determined by rules for evaluation of expressions
Stream input (GET statement)	Attributes of receiving field
Stream output (PUT statement)	As determined by format list if stream is edit-directed, otherwise character-string
Argument to PROCEDURE or ENTRY	Attributes of corresponding parameter
Argument to built-in function or pseudovariable	Depends on the function or pseudovariable
INITIAL attribute	Other attributes of variable being initialized
RETURN statement expression	Attributes specified in PROCEDURE statement
DO statement, BY, TO, or REPEAT option	Attributes of control variable

Data conversion

The following can cause conversion to character values:

Statement	Option
DISPLAY	
Record I/O	KEYFROMKEY
OPEN	TITLE

The following can cause conversion to a BINARY value:

Statement	Option/Attribute/Reference
DECLARE, ALLOCATE, DEFAULT	length, size, dimension, bound, repetition factor
DELAY	milliseconds
FORMAT (and format items in GET and PUT)	iteration factorw, d, s, p
OPEN	LINESIZE, PAGESIZE
I/O	SKIP, LINE, IGNORE
Most statements	subscript

All attributes for source and target data items (except string length) must be specified at compile time. Conversion can raise one of the following conditions: CONVERSION, OVERFLOW, SIZE, or STRINGSIZE. (Refer to Chapter 17, "Conditions," on page 339.)

Constants can be converted at compile time as well as at run time. In all cases, the conversions are as described here.

More than one conversion might be required for a particular operation. The implementation does not necessarily go through more than one. To understand the conversion rules, it is convenient to consider them separately, for example:

```
dcl A fixed dec(3,2) init(1.23);
dcl B fixed bin(15,5);
B = A;
```

In this example, the decimal representation of 1.23 is first converted to binary (11,7), as 1.0011101B. Then precision conversion is performed, resulting in a binary (15,5) value of 1.00111B.

Additional examples of conversion are provided at the end of this chapter.

Built-in functions for computational data conversion

Conversions can take place during expression evaluation, I/O GET and PUT operations, and assignment operations, and between arguments and parameters. Conversions can also be initiated with the following built-in functions:

BINARY	FIXED	REAL
BIT	FLOAT	SIGNED
CHAR	GRAPHIC	UNSIGNED
COMPLEX	IMAG	WIDECHAR
DECIMAL	PRECISION	

Each is discussed in Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365.

Each function returns a value with the attribute specified by the function name, performing any required conversions.

With the exception of the conversions performed by the COMPLEX, GRAPHIC, and IMAG built-in functions, assignment to a PL/I variable having the required attributes can achieve the conversions performed by these built-in functions. However, you might find it easier and clearer to use a built-in function than to create a variable solely to carry out a conversion.

Converting string lengths

The source string is assigned to the target string from left to right. If the source string is longer than the target, excess characters, bits, graphics or widechars on the right are ignored, and the STRINGSIZE condition is raised. For fixed-length targets, if the target is longer than the source, the target is padded on the right. If STRINGSIZE is disabled, and the length of the source and/or the target is determined at run time, and the target is too short to contain the source, unpredictable results can occur.

Note: If you use SUBSTR with variables as the parameters, and the variables specify a string not contained in the target, unpredictable results can occur if the STRINGRANGE condition is not enabled.

Character strings are padded with blanks, bit strings with '0'B, graphic strings with DBCS blanks, and widechar strings with widechar blanks.

```
declare Subject char(10);
Subject = 'Transformations';
```

'Transformations' has 15 characters, therefore, when PL/I assigns the string to `Subject`, it truncates five characters from the right end of the string. This is equivalent to executing the following:

```
Subject = 'Transforma';
```

The first two of the following statements assign equivalent values to `Subject` and the last two assign equivalent values to `Code`:

```
Subject = 'Physics';
Subject = 'Physics   ';
declare Code bit(10);
Code = '110011'B;
Code = '1100110000'B;
```

The following statements do *not* assign equivalent values to `Subject`:

```
Subject = '110011'B;
Subject = '1100110000'B;
```

When the first statement is executed, the bit constant on the right is first converted to a character string and is then extended on the right with blank characters rather than zero characters. This statement is equivalent to:

```
Subject = '110011bbbb';
```

The second of the two statements requires only a conversion from bit to character type and is equivalent to:

```
Subject = '1100110000';
```

A string value is not extended with blank characters or zero bits when it is assigned to a string variable that has the VARYING attribute. Instead, the length of the target

Converting string lengths

string variable is set to the length of the assigned string. However, truncation will occur if the length of the assigned string exceeds the maximum length declared for the varying-length string variable.

Converting arithmetic precision

When an arithmetic value has the same data attributes (except for precision) as the target, precision conversion is required.

For fixed-point data items, decimal or binary point alignment is maintained during precision conversion. Therefore, padding or truncation can occur on the left or right. If nonzero bits or digits on the left are lost, the SIZE condition is raised.

For floating-point data items, truncation on the right, or padding on the right with zeros, can occur.

Converting mode

If a complex value is converted to a real value, the imaginary part is ignored. If a real value is converted to a complex value, the imaginary part is zero.

Converting other data attributes

Source-to-target rules are given, following this section, for converting data items with the following data attributes:
- Coded arithmetic:
  FIXED BINARY
  FIXED DECIMAL
  FLOAT BINARY
  FLOAT DECIMAL
- Arithmetic character PICTURE
- CHARACTER
- BIT
- GRAPHIC
- WIDECHAR

Changes in value can occur in converting between decimal representations and binary representation. In converting between binary and decimal, the factor 3.32 is used as follows:
- n decimal digits convert to CEIL (n*3.32) binary digits.
- n binary digits convert to CEIL (n/3.32) decimal digits.

A table of CEIL values is provided in Table 22 to calculate these conversions.

*Table 22. CEIL (n*3.32) and CEIL (n/3.32) values*

n	CEIL (n*3.32)	n	CEIL (n/3.32)
1	4	1-3	1
2	7	4-6	2
3	10	7-9	3
4	14	10-13	4
5	17	14-16	5
6	20	17-19	6
7	24	20-23	7
8	27	24-26	8

Table 22. CEIL (n*3.32) and CEIL (n/3.32) values (continued)

n	CEIL (n*3.32)	n	CEIL (n/3.32)
9	30	27-29	9
10	34	30-33	10
11	37	34-36	11
12	40	37-39	12
13	44	40-43	13
14	47	44-46	14
15	50	47-49	15
16	53[1]	50-53	16
17	57	54-56	17
18	60	57-59	18
19	64	60-63	19
20	67	64-66	20
21	70	67-69	21
22	74	70-73	22
23	77	74-76	23
24	80	77-79	24
25	83	80-83	25
26	87	84-86	26
27	90	87-89	27
28	93	90-92	28
29	97	93-96	29
30	100	97-99	30
31	103	100-102	31
32	107	103-106	32
33	110	107-109	33
		110-112	34
		113-116	35

Note 1: While ceil(16*3.32) = 54, the value 53 is used. If it were not, a float decimal(16), when converted to binary, would have to be converted from long floating-point to extended floating-point (because float binary(54) is represented as extended floating-point).

For fixed-point integer values, conversion does not change the value. For fixed-point fractional values, the factor 3.32 provides only enough digits or bits so that the converted value differs from the original value by less than 1 digit or bit in the rightmost place.

For example, the decimal constant .1, with attributes FIXED DECIMAL (1,1), converts to the binary value .0001B, converting 1/10 to 1/16. The decimal constant .10, with attributes FIXED DECIMAL (2,2), converts to the binary value .0001100B, converting 10/100 to 12/128.

Source-to-target rules

Target: Coded Arithmetic

Source:

FIXED BINARY, FIXED DECIMAL,

FLOAT BINARY, and FLOAT DECIMAL
These are all coded arithmetic data. Rules for conversion between them are given under each data type taken as a target.

Arithmetic character PICTURE
Data first converts to decimal with scale and precision determined by the corresponding PICTURE specification. The decimal value then converts to the base, scale, mode, and precision of the target. See the specific target types of coded arithmetic data using FIXED DECIMAL or FLOAT DECIMAL as the source.

CHARACTER
The source string must represent a valid arithmetic constant or complex expression; otherwise, the CONVERSION condition is raised. The constant can be preceded by a sign and can be surrounded by blanks. The constant cannot contain blanks between the sign and the constant, or between the end of the real part and the sign preceding the imaginary part of a complex expression.

The constant has base, scale, mode, and precision attributes. It converts to the attributes of the target when they are independent of the source attributes, as in the case of assignment. See the specific target types of coded arithmetic data using the attributes of the constant as the source.

If an intermediate result is necessary, as in evaluation of an operational expression, the attributes of the intermediate result are the same as if a decimal fixed-point value of precision had appeared in place of the string. (This allows the compiler to generate code to handle all cases, regardless of the attributes of the contained constant.) Consequently, any fractional portion of the constant might be lost. See the specific target types of coded arithmetic data using FIXED DECIMAL as the source.

It is possible that during the initial conversion of the character data item to an intermediate fixed decimal number, the value might exceed the default size of the intermediate result. If this occurs, the SIZE condition is raised if it is enabled.

If a character string representing a complex number is assigned to a real target, the complex part of the string is not checked for valid arithmetic characters and CONVERSION cannot be raised, since only the real part of the string is assigned to the target.

A null string and a string of blanks both give the value zero.

BIT
If the conversion occurs during evaluation of an operational expression, the source bit string is converted to an unsigned value that is FIXED BINARY(M,0). See the specific target types of coded arithmetic data using FIXED BINARY as the source.

If the source string is longer than the allowable precision, bits on the left are ignored. If nonzero bits are lost, the SIZE condition is raised.

A null string gives the value zero.

GRAPHIC
Graphic variables and strings are converted to CHARACTER, and then follow the rules for character source described

WIDECHAR
Widechar variables and strings are converted to CHARACTER, and then follow the rules for character source described in 78.

Target: FIXED BINARY (p2,q2)

Source:

FIXED DECIMAL (p1,q1)
> The precision of the result is p2 = min(N,1+CEIL(p1*3.32)) and q2=CEIL(ABS(q1*3.32))*SIGN(q1).

FLOAT BINARY (p1)
> The precision conversion is as described under "Converting arithmetic precision" on page 76 with p1 as declared or indicated and q1 as indicated by the binary point position and modified by the value of the exponent.

FLOAT DECIMAL (p1)
> The precision conversion is the same as for FIXED DECIMAL to FIXED BINARY with p1 as declared or indicated and q1 as indicated by the decimal point position and modified by the value of the exponent.

Arithmetic character PICTURE
> See "Target: Coded Arithmetic" on page 77.

CHARACTER
> See "Target: Coded Arithmetic" on page 77.

BIT
> See "Target: Coded Arithmetic" on page 77.

GRAPHIC
> See "Target: Coded Arithmetic" on page 77.

WIDECHAR
> See "Target: Coded Arithmetic" on page 77.

Target: FIXED DECIMAL (p2,q2)

Source:

FIXED BINARY (p1,q1)
> The precision of the result is p2=1+CEIL(p1/3.32) and q2=CEIL(ABS(q1/3.32))*SIGN(q1).

FLOAT BINARY (p1)
> The precision conversion is the same as for FIXED BINARY to FIXED DECIMAL with p1 as declared or indicated and q1 as indicated by the binary point position and modified by the value of the exponent.

FLOAT DECIMAL (p1)
> The precision conversion is as described under "Converting arithmetic precision" on page 76 with p1 as declared or indicated and q1 as indicated by the decimal point position and modified by the value of the exponent.

Arithmetic character PICTURE
> See "Target: Coded Arithmetic" on page 77.

CHARACTER
> See "Target: Coded Arithmetic" on page 77.

BIT
> See "Target: Coded Arithmetic" on page 77.

GRAPHIC
See "Target: Coded Arithmetic" on page 77.

WIDECHAR
See "Target: Coded Arithmetic" on page 77.

Target: FLOAT BINARY (p2)

Source:

FIXED BINARY (p1,q1)
The precision of the result is p2=p1. The exponent indicates any fractional part of the value.

FIXED DECIMAL (p1,q1)
The precision of the result is p2=CEIL(p1*3.32). The exponent indicates any fractional part of the value.

FLOAT DECIMAL (p1)
The precision of the result is p2=CEIL(p1*3.32).

Arithmetic character PICTURE
See "Target: Coded Arithmetic" on page 77.

CHARACTER
See "Target: Coded Arithmetic" on page 77.

BIT
See "Target: Coded Arithmetic" on page 77.

GRAPHIC
See "Target: Coded Arithmetic" on page 77.

WIDECHAR
See "Target: Coded Arithmetic" on page 77.

Target: FLOAT DECIMAL (p2)

Source:

FIXED BINARY (p1,q1)
The precision of the result is p2=CEIL(p1/3.32). The exponent indicates any fractional part of the value.

FIXED DECIMAL (p1,q1)
The precision of the result is p2=p1. The exponent indicates any fractional part of the value.

FLOAT BINARY (p1)
The precision of the result is p2=CEIL(p1/3.32).

Arithmetic character PICTURE
See "Target: Coded Arithmetic" on page 77.

CHARACTER
See "Target: Coded Arithmetic" on page 77.

BIT
See "Target: Coded Arithmetic" on page 77.

GRAPHIC

See "Target: Coded Arithmetic" on page 77.

WIDECHAR

See "Target: Coded Arithmetic" on page 77.

Target: Arithmetic character PICTURE

The arithmetic character PICTURE data item is the character representation of a decimal fixed-point or floating-point value. The following descriptions for source to arithmetic character PICTURE target show those target attributes that allow assignment without loss of leftmost or rightmost digits.

Source:

FIXED BINARY (p1,q1)

The target must imply:

```
fixed decimal (1+x+q-y,q) or
float decimal (x)
```

where x>=CEIL(p1/3.32), y=CEIL(q1/3.32), and q>=y.

FIXED DECIMAL (p1,q1)

The target must imply:

```
fixed decimal (x+q-q1,q) or
float decimal (x)
```

where x>=p1 and q>=q1.

FLOAT BINARY (p1)

The target must imply:

```
fixed decimal (p,q) or
float decimal (p)
```

where p>=CEIL(p1/3.32) and the values of p and q take account of the range of values that can be held by the exponent of the source.

FLOAT DECIMAL (p1)

The target must imply:

```
fixed decimal (p,q) or
float decimal (p)
```

where p>= p1 and the values of p and q take account of the range of values that can be held by the exponent of the source.

Arithmetic character PICTURE

The implied attributes of the source will be either FIXED DECIMAL or FLOAT DECIMAL. See the respective entries for this target.

CHARACTER

See "Target: Coded Arithmetic" on page 77.

BIT(n)

The target must imply:

```
fixed decimal (1+x+q,q) or
float decimal (x)
```

where x>=ceil(n/3.32) and q>=0.

Source-to-target rules

GRAPHIC
See "Target: Coded Arithmetic" on page 77.

WIDECHAR
See "Target: Coded Arithmetic" on page 77.

Target: CHARACTER

Source:

FIXED BINARY, FIXED DECIMAL,
FLOAT BINARY, and FLOAT DECIMAL
The coded arithmetic value is converted to a decimal constant (preceded by a minus sign if it is negative) as described below. The constant is inserted into an intermediate character string whose length is derived from the attributes of the source. The intermediate string is assigned to the target according to the rules for string assignment.

The rules for coded-arithmetic-to-character-string conversion are also used for list-directed and data-directed output, and for evaluating keys (even for REGIONAL files).

FIXED BINARY (p1,q1)
The binary precision (p1,q1) is first converted to the equivalent decimal precision (p,q), where $p=1+CEIL(p1/3.32)$ and $q=CEIL(ABS(q1/3.32))*SIGN(q1)$. Thereafter, the rules are the same as for FIXED DECIMAL to CHARACTER.

FIXED DECIMAL (p1,q1)
If $p1>=q1>=0$ then:

- The constant is right adjusted in a field of width p1+3. (The 3 is necessary to allow for the possibility of a minus sign, a decimal or binary point, and a leading zero before the point.)
- Leading zeros are replaced by blanks, except for a single zero that immediately precedes the decimal point of a fractional number. A single zero also remains when the value of the source is zero.
- A minus sign precedes the first digit of a negative number. A positive value is unsigned.
- If q1=0, no decimal point appears; if q1>0, a decimal point appears and the constant has q fractional digits.

If $p1<q1$ or $q1<0$, a scaling factor appends to the right of the constant; the constant is an optionally-signed integer. The scaling factor appears even if the value of the item is zero and has the following syntax:

```
F{+|-}nn
```

where {+|-}nn has the value of -q1.

The length of the intermediate string is p1+k+3, where k is the number of digits necessary to hold the value of q1 (not including the sign or the letter F).

If the arithmetic value is complex, the intermediate string consists of the imaginary part concatenated to the real part. The left-hand, or real, part is generated as a real source. The right-hand, or imaginary, part is always signed, and it has the letter I appended. The generated string is a complex expression with no blanks between its elements. The length of the intermediate string is:

```
2*p1+7     for p1>=q1>=0
2*(p1+k)+7 for p1<q1 or q1<0
```

The following examples show the intermediate strings that are generated from several real and complex fixed-point decimal values:

```
Precision    Value      String

(5,0)        2947       'bbbb2947'
(4,1)        -121.7     'b-121.7'
(4,-3)       -3279000   '-3279F+3'
(2,1)        1.2+0.3I   'bbb1.2+0.3I'
```

FLOAT BINARY (p1)

The floating-point binary precision (p1) first converts to the equivalent floating-point decimal precision (p), where p=CEIL(p1/3.32). Thereafter, the rules are the same as for FLOAT DECIMAL to CHARACTER.

FLOAT DECIMAL (p1)

A decimal floating-point source converts as if it were transmitted by an E-format item of the form E(w,d,s) where:

w, the length of the intermediate string, is p1+8.

d, the number of fractional digits, is p1-1.

s, the number of significant digits, is p1.

If the arithmetic value is complex, the intermediate string consists of the imaginary part concatenated to the real part. The left-hand, or real, part is generated as a real source. The right-hand, or imaginary, part is always signed, and it has the letter I appended. The generated string is a complex expression with no blanks between its elements. The length of the intermediate string is 2*p+17.

The following examples show the intermediate strings that are generated from several real and complex floating-point decimal values:

```
Precision    Value          String

(5)          1735*10**5     'b1.7350E+0008'
(5)          -.001663       '-1.6630E-0003'
(3)          1              'b1.00E+0000'
(5)          17.3+1.5I      'b1.7300E+0001+1.5000E+0000I'
```

Arithmetic character PICTURE

A real arithmetic character field is interpreted as a character string and assigned to the target string according to the rules for converting string lengths. If the arithmetic character field is complex, the real and imaginary parts are concatenated before assignment to the target string. Insertion characters are included in the target string.

BIT

Bit 0 becomes the character 0 and bit 1 becomes the character 1. A null bit string becomes a null character string. The generated character string is assigned to the target string according to the rules for converting string lengths.

GRAPHIC

DBCS to SBCS conversion is possible only if there is a corresponding SBCS character. Otherwise, the CONVERSION condition is raised.

WIDECHAR

Conversion from widechar to character is performed only if all the widechars have a value less than '0080'wx. Otherwise, the CONVERSION condition is raised.

Target: BIT

Source:

**FIXED BINARY, FIXED DECIMAL,
FLOAT BINARY, and FLOAT DECIMAL**

If necessary, the arithmetic value converts to binary and both the sign and any fractional part are ignored. (If the arithmetic value is complex, the imaginary part is also ignored.) The resulting binary value is treated as a bit string. It is assigned to the target according to the rules for string assignments.

FIXED BINARY (p1,q1)

The length of the intermediate bit string is given by:

```
min(M,(p1-q1))
```

If (p1-q1) is negative or zero, the result is a null bit string.

The following examples show the intermediate strings that are generated from several fixed-point binary values:

```
Precision   Value     String

(1)             1     '1'B
(3)            -3     '011'B
(4,2)        1.25     '01'B
```

FIXED DECIMAL (p1,q1)

The length of the intermediate bit string is given by:

```
min(M,CEIL((p1-q1)*3.32))
```

If (p1-q1) is negative or zero, the result is a null bit string.

The following examples show the intermediate strings that are generated from several fixed-point decimal values:

```
Precision   Value     String

(1)             1      '0001'B
(2,1)         1.1      '0001'B
```

FLOAT BINARY (p1)

The length of the intermediate bit string is given by:

```
min(M,p1)
```

FLOAT DECIMAL (p1)

The length of the intermediate bit string is given by:

```
min(M,ceil(p1*3.32))
```

Arithmetic character PICTURE

Data is first interpreted as decimal with scale and precision determined by the corresponding PICTURE specification. The item then converts according to the rules given for FIXED DECIMAL or FLOAT DECIMAL to BIT.

CHARACTER

Character 0 becomes bit 0 and character 1 becomes bit 1. Any character other

than 0 or 1 raises the CONVERSION condition. A null string becomes a null bit string. The generated bit string, which has the same length as the source character string, is assigned to the target according to the rules for string assignment.

GRAPHIC

Graphic 0 becomes bit 0 and graphic 1 becomes bit 1. Any graphic other than 0 or 1 raises the CONVERSION condition. A null string becomes a null bit string. The generated bit string, which has the same length as the source graphic string, is then assigned to the target according to the rules for string assignment.

WIDECHAR

Widechar 0 ('0030'wx) becomes bit 0 and widechar 1 ('0031'wx) becomes bit 1. Any widechar other than 0 or 1 raises the CONVERSION condition. A null string becomes a null bit string. The generated bit string, which has the same length as the source widechar string, is then assigned to the target according to the rules for string assignment.

Target: GRAPHIC

Nongraphic source is first converted to character according to the rules in "Target: Character" on page 82. The resultant character string is then converted to a DBCS string.

Target: WIDECHAR

Source other than widechar is first converted to character according to the rules in "Target: Character" on page 82. The resultant character string is then converted to a widechar string.

Examples

DECIMAL FIXED to BINARY FIXED with fractions

```
dcl I fixed bin(31,5) init(1);
    I = I+.1;
```

The value of I is now 1.0625. This is because .1 is converted to FIXED BINARY (5,4), so that the nearest binary approximation is 0.0001B (no rounding occurs). The decimal equivalent of this is .0625. The result achieved by specifying .1000 in place of .1 would be different.

Arithmetic to bit string

```
dcl A bit(1),
    D bit(5);
A=1;          /* A has value '0'B     */
D=1;          /* D has value '00010'B */
D='1'B;       /* D has value '10000'B */
if A=1 then go to Y;
     else go to X;
```

The branch is to X, because the assignment to A resulted in the following sequence of actions:

Examples

1. The decimal constant, 1, has the attributes FIXED DECIMAL (1,0) and is assigned to temporary storage with the attributes FIXED BINARY(4,0) and the value 0001B.
2. This value now converts to a bit string of length (4), so that it becomes '0001'B.
3. The bit string is assigned to A. Since A has a declared length of 1, and the value to be assigned has acquired a length of 4, truncation occurs at the right, and A has a final value of '0'B.

For the comparison operation in the IF statement, '0'B and 1 convert to FIXED BINARY and compare arithmetically. They are unequal, giving a result of *false* for the relationship A=1.

In the first assignment to D, a sequence of actions similar to that described for A takes place, except that the value is extended at the right with a zero, because D has a declared length that is 1 greater than that of the assigned value.

Arithmetic to character

In the following example, the three blanks are necessary to allow for the possibility of a minus sign, a decimal or binary point, and provision for a single leading zero before the point:

```
dcl A char(4),
    B char(7);
A='0'; /*A has value '0bbb'*/
A=0;   /*A has value 'bbb0'*/
B=1234567; /*B has value 'bbb1234'*/
```

A conversion error

```
dcl Ctlno char(8) init('0');
do I=1 to 100;
Ctlno=Ctlno+1;
 :
 :
end;
```

For this example, FIXED DECIMAL precision 15 was used for the implementation maximum. The example raises the CONVERSION condition because of the following sequence of actions:

1. The initial value of CTLNO, that is, '0bbbbbbb' converts to FIXED DECIMAL(15,0).
2. The decimal constant, 1, with attributes FIXED DECIMAL(1,0), is added; in accordance with the rules for addition, the precision of the result is (16,0).
3. This value now converts to a character string of length 18 in preparation for the assignment back to CTLNO.
4. Because CTLNO has a length of 8, the assignment causes truncation at the right; thus, CTLNO has a final value that consists entirely of blanks. This value cannot be successfully converted to arithmetic type for the second iteration of the loop.

Chapter 6. Program organization

This chapter discusses how statements can be organized into different kinds of
blocks to form a PL/I program, how control flows among blocks, and how different
blocks can make use of the same data.

Proper division of a program into blocks simplifies the writing and testing of the program, particularly when many programmers are writing it. Proper division can also result in more efficient use of storage, since automatic storage is allocated on entry to the block in which it is declared and released when the block is terminated.

Programs

Program structure

PL/I is a block-structured language, consisting of packages, procedures, begin-blocks, statements, expressions, and built-in functions.

A PL/I *application* consists of one or more separately loadable entities, known as a *load module*. Each load module can consist of one or more separately compiled entities, known as a *compilation unit (CU)*. Unless otherwise stated, a *program* refers to a PL/I application or a compilation unit.

A compilation unit is a PL/I PACKAGE or an external PROCEDURE. Each package can contain zero or more procedures, some or all of which can be exported. A PL/I external or internal procedure contains zero or more blocks. A PL/I block is either a PROCEDURE or a BEGIN block, which contains zero or more statements and/or zero or more blocks.

A PL/I block allows you to produce highly-modular applications, because blocks can contain declarations that define variable names and storage class. Thus, you can restrict the scope of a variable to a single block or a group of blocks, or can make it known throughout the compilation unit or a load module.

By giving you freedom to determine the degree to which a block is self-contained, PL/I makes it possible to produce blocks that many compilation units and applications can share, leading to code reuse.

Figure 2 shows an application structure.
Packages are discussed in "Packages" on page 90.
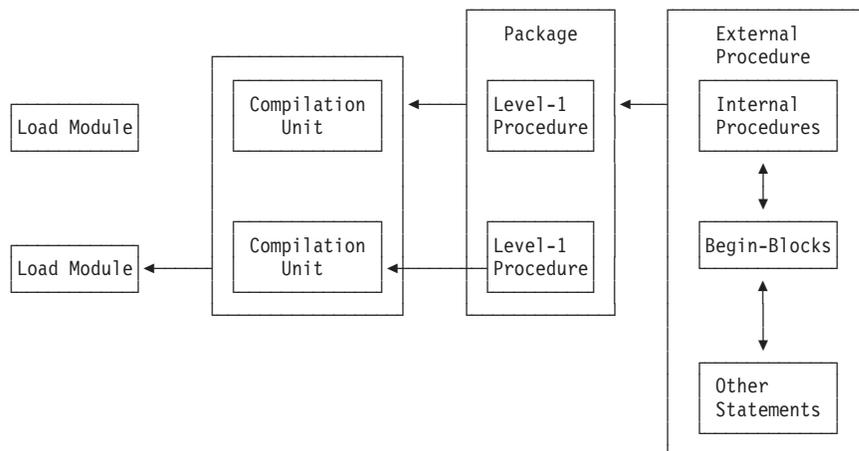
Figure 2. A PL/I application structure

Procedures are discussed in "Procedures" on page 92.

Begin-blocks are discussed in "Begin-blocks" on page 110.

Program activation

A PL/I program becomes active when a calling program invokes the *main procedure*. This calling program usually is the operating system, although it could be another program. The main procedure is the external procedure for which the statement has the OPTIONS(MAIN) specification. In the following example, `Contrl` is the main procedure and it invokes other external procedures in the program. The main procedure remains active for the duration of the program.

```
Contrl: procedure options(main);
  call A;
  call B;
  call C;
end Contrl;
```

Program termination

A program is terminated when the main procedure is terminated. Whether termination is normal or abnormal, control returns to the calling program. In the previous example, when control transfers from the C procedure back to the `Contrl` procedure, `Contrl` terminates. See "Procedure termination" on page 99 for more information.

Blocks

A block is a delimited sequence of statements that does the following:
* Establishes the scope of names declared within it
* Limits the allocation of automatic variables
* Determines the scope of DEFAULT statements (as described in "Defaults for attributes" on page 162).

The kinds of blocks are:
* Package
* Procedure
* Begin

These blocks can contain declarations that are treated as local definitions of names. This is done to establish the scope of the names and to limit the allocation of automatic variables. These declarations are not known outside their own block, and the names cannot be referred to in the containing block. See "Scope of declarations" on page 151 for more information.

Storage is allocated to automatic variables upon entry to the block where the storage is declared, and is freed upon exit from the block. See "Scope of declarations" on page 151, for more information.

Block activation

Each block plays the same role in the allocation and freeing of storage and in delimiting the scope of names. How activation occurs is discussed in "Procedures" on page 92 and "Begin-blocks" on page 110. Packages are neither activated nor terminated.

During block activation, the following are performed:
* Expressions that appear in declare statements are evaluated for extents and initial values (including iteration factors).

- Storage is allocated for automatic variables. Their initial values are set if specified.
- Storage is allocated for dummy arguments and compiler-created temporaries that might be created in this block.

Initial values and extents for automatic variables must not depend on the values or extents of other automatic variables declared in the same block. For example, the following initialization can produce incorrect results for J and K:

```
dcl I init(10),J init(K),K init(I);
```

Declarations of data items must not be mutually interdependent. For example, the following declarations are invalid:

```
dcl A(B(1)), B(A(1));
```

```
dcl D(E(1)), E(F(1)), F(D(1));
```

Errors can occur during block activation, and the ERROR condition (or other conditions) can be raised. If so, the environment of the block might be incomplete. In particular, some automatic variables might not have been allocated. Statements referencing automatic variables executed after the ERROR condition has been raised may reference unallocated storage. The results of referring to unallocated storage are undefined.

Block termination

There are a number of ways a block can be terminated. How termination occurs is discussed in "Procedures" on page 92 and "Begin-blocks" on page 110. Packages are neither activated nor terminated.

During block termination:
- The ON-unit environment is reestablished as it existed before the block was activated.
- Storage for all automatic variables allocated in the block is released.

Packages

A package is a block that can contain only declarations, default statements, and procedure blocks. The package forms a name scope that is shared by all declarations and procedures contained in the package, unless the names are declared again. Some or all of the level-1 procedures can be exported and made known outside of the package as external procedures. A package can be used for implementing multiple entry point applications.

```
►►──condition-prefix──:──package-name──:──PACKAGE──────────────────────────────►

                                                    ┌─────,─────┐
►──────────────────────────────────────────────────────────────────────────────►
     ┌─EXPORTS──(──┬──▼──procedure──┬──)─┘
     │             └──*──┘
                                              ┌─OPTIONS(options)─┐       ;
►────────────────────────────────────────────────────────────────────────────►
     ┌─RESERVES──(──┬──▼──variable name──┬──)─┘
     │              └──*──┘

      ┌──────▼──────┐
►──────┬─declare-statement───┬──END──────────────────;─────────────────►◄
       ├─default-statement───┤    └─package-name─┘
       └─procedure-statement─┘
```

procedure:

```
├──procedure-name──────────────────────────────────────┤
               └─EXTERNAL(environment-name)─┘
```
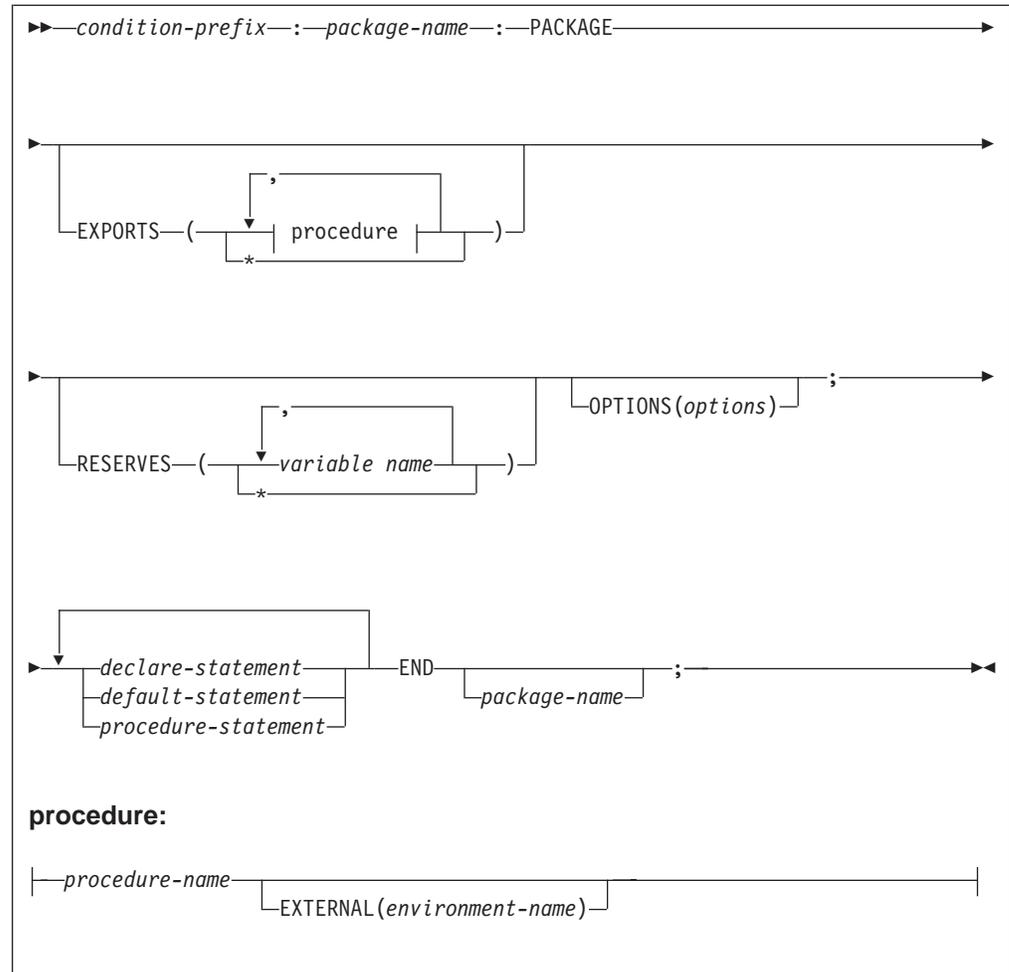
condition-prefix
Condition prefixes specified on a PACKAGE statement apply to all procedures contained in the package unless overridden on the PROCEDURE statement. For more information on condition prefixes, refer to "Condition prefixes" on page 331.

package-name
The name of the package.

EXPORTS
Specifies that all (EXPORTS(*)) or the named procedures are to be exported and thus made externally known outside of the package. If no EXPORTS option is specified, EXPORTS(*) is assumed.

procedure name
Is the name of a level-1 procedure within the package.

EXTERNAL (environment name)
Is a scope attribute discussed in "Scope of declarations" on page 151.

RESERVES
Specifies that this package reserves the storage for all (RESERVES(*)), or only for the named variables that have the RESERVED attribute, see "RESERVED attribute" on page 158.

variable name
Is the name of a level-1 external static variable.

OPTIONS option

For OPTIONS options applicable to a package statement, refer to "OPTIONS option and attribute" on page 126.

declare statement

All variables declared within a package but outside any contained level-1 procedure must have the storage class of static, based, or controlled. Automatic variables are not allowed. Default storage class is STATIC. Refer to Chapter 8, "Data declarations," on page 147.

default statement

Refer to "Defaults for attributes" on page 162.

procedure statement

Refer to "PROCEDURE and ENTRY statements" on page 93.

An example of the package statement appears in Figure 3.

```
*Process S A(F) LANGLVL(SAA2) LIMITS(EXTNAME(31)) NUMBER;
 Package_Demo: Package exports (Factorial);

 /***********************************************/
 /*                Common Data                  */
 /***********************************************/

 dcl N fixed bin(15);
 dcl Message char(*) value('The factorial of ');

 /***********************************************/
 /*                Main Program                 */
 /***********************************************/

 Factorial: proc options (main);
    dcl Result fixed bin(31);
    put skip list('Please enter a number whose factorial ' ||
                  'must be computed ');
    get list(N);
    Result = Compute_factorial(n);
    put list(Message || trim(N) || ' is ' || trim(Result));
 end Factorial;

 /***********************************************/
 /*                Subroutine                   */
 /***********************************************/

 Compute_factorial: proc (Input) recursive returns (fixed bin(31));
    dcl Input fixed bin(15);
    if Input <= 1 then
       return(1);
    else
       return( Input*Compute_factorial(Input-1) );
 end Compute_factorial;

 end Package_Demo;
```

Figure 3. Package statement

Procedures

A procedure is a sequence of statements delimited by a PROCEDURE statement and a corresponding END statement. A procedure can be a main procedure, a subroutine, or a function. An application must have exactly one external procedure that has OPTIONS(MAIN). In the following example, the name of the procedure is Name and represents the *entry point* of the procedure.

```
Name:
    procedure;
end Name;
```

The ENTRY statement can define a secondary entry point to a procedure. For example,

```
Name: procedure;
B: entry;
end Name;
```

B defines a secondary entry point to the Name procedure. The ENTRY statement is described in "ENTRY statement" on page 94.

A procedure must have a name. A procedure block nested within another procedure or begin-block is called an *internal procedure*. A procedure block not nested within another procedure or begin-block is called an *external procedure*. Level-1 exported procedures from a package also become external procedures. External procedures can be invoked by other procedures in other compilation units. Procedures can invoke other procedures.

A procedure can be recursive, which means that it can be reactivated from within itself or from within another active procedure while it is already active. You can pass arguments when invoking a procedure.

For more information on these subjects, see the following sections:
- "Scope of declarations" on page 151
- "Subroutines" on page 104
- "Functions" on page 106
- "Passing arguments to procedures" on page 108.

PROCEDURE and ENTRY statements

A procedure (subroutine or function) can have one or more entry points. The primary entry point to a procedure is established by the leftmost label of the procedure statement. Secondary entry points to a procedure are established by additional labels on the PROCEDURE statement and by the ENTRY statement. Each entry point has an entry name. See "INTERNAL and EXTERNAL attributes" on page 153 for a discussion of the rules for the creation of an external name.

A PROCEDURE statement identifies the procedure as a main procedure, a subroutine, or a function. Parameters expected by the procedure and other characteristics are also specified on the PROCEDURE statement.

PROCEDURE and ENTRY
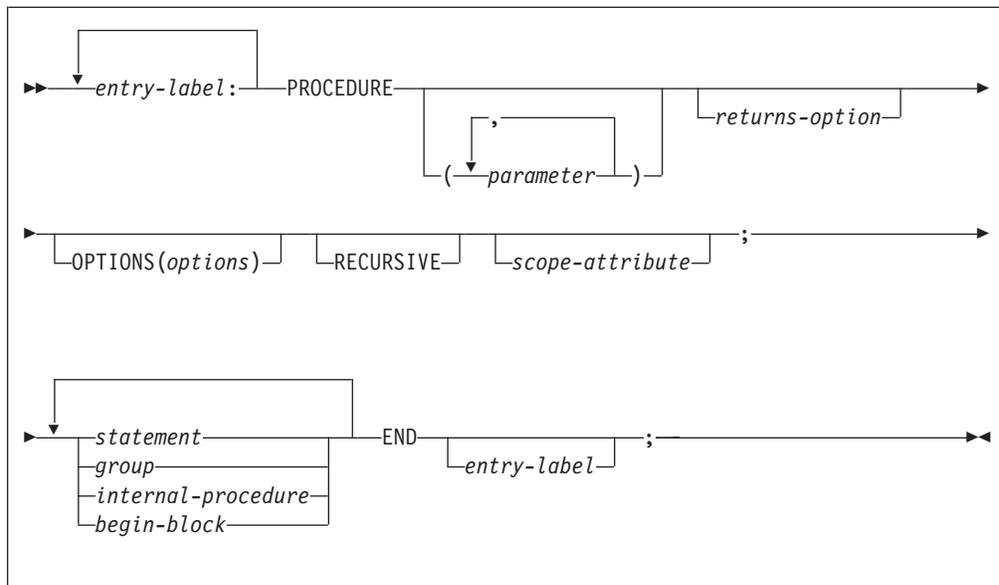
Abbreviations: PROC for PROCEDURE

entry-label
> The entry point to the procedure. External entries are explicitly declared in the invoking procedure. If multiple entry labels are specified, the leftmost name is the primary entry point and is the name returned by the PROCNAME and ONLOC built-in functions. For more information on entry data, refer to "Entry data" on page 111.

parameter
> Refer to "Parameter attribute" on page 95 and "Passing arguments to procedures" on page 108.

returns-option
> Applies only to function procedures. Refer to "Functions" on page 106 and "RETURNS option and attribute" on page 134.

OPTIONS option
> Refer to "OPTIONS option and attribute" on page 126.

RECURSIVE
> Refer to "Recursive procedures" on page 100.

scope-attribute
> Refer to "Scope of declarations" on page 151.

ENTRY statement

The ENTRY statement specifies a secondary entry point of a procedure. The ENTRY statement must be internal to the procedure for which it defines a secondary entry point. It cannot be within a do-group that specifies repetitive execution, or internal to a ON-unit.

```
>>──┬─entry-label:─┬─ ENTRY ────────────────────────────────────────>

        │←──────────────│
        └─ ,────────────┘
                ┌────────,←──────┐
              └─(─▼─parameter─┴─)─┘

 >──┬──────────────────────────────┬──┬──────────────────────┬─ ; ──><
    │              ┌────,←─────┐    │  └─ OPTIONS(options) ──┘
    └─ RETURNS(─▼─attribute─┴─)─┘
```
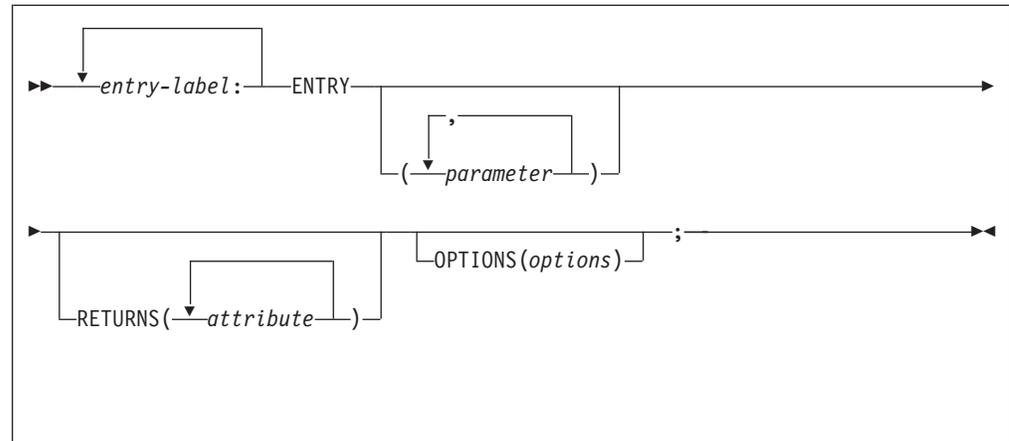
entry-label
> The secondary entry point to the procedure.

parameter
> Refer to "Parameter attribute" and "Passing arguments to procedures" on page 108.

RETURNS option
> Refer to "RETURNS option and attribute" on page 134.

OPTIONS option
> Refer to "OPTIONS option and attribute" on page 126.

All parameters on an ENTRY statement must be BYADDR, and for a procedure containing ENTRY statements, all non-pointer parameters to that procedure must be BYADDR.

If a procedure containing ENTRY statements has the RETURNS option (or if any of its contained ENTRY statements have the RETURNS option), then

- the BYADDR attribute must be specified (or implied by the compile-time option DEFAULT(RETURNS(BYADDR)) in all of the RETURNS options for that procedure and its ENTRY statements.
- All routines that call one of these entry points must also either declare the entry with RETURNS(BYADDR) or be compiled with the DEFAULT(RETURNS(BYADDR)) compiler option.

When a procedure contains ENTRY statements and some, but not all of its entry points have the RETURNS attribute, the ERROR condition is detected under the following circumstances:

- If the code executes a RETURN statement with an expression when the procedure was entered at an entry point which did not have the RETURNS attribute.
- If the code executes a RETURN statement without an expression when the procedure was entered at an entry point that has the RETURNS attribute.

Parameter attribute

A parameter is contextually declared with the parameter attribute by its specification in a PROCEDURE or ENTRY statement. The parameter should be explicitly declared with appropriate attributes. The PARAMETER attribute can also be

specified in the declaration. If attributes are not supplied in a DECLARE statement, default attributes are applied. The parameter name must not be subscripted or qualified.

```
►►──PARAMETER────────────────────────────────────────────►◄
```

Table 8 on page 27, and the following discussion, describe the attributes that can be declared for a parameter.

A parameter always has the INTERNAL attribute.

If the parameter is a structure or union, it must specify the level-1 name.

A parameter cannot have any storage class attributes except CONTROLLED. A controlled parameter must have a controlled argument, and can also have the INITIAL attribute.

Parameters used in record-oriented input/output, or as the base variable for DEFINED items, must be in connected storage. The CONNECTED attribute must be specified both in the declaration in the procedure and in the descriptor list of the procedure entry declaration.

Simple Parameter Bounds, Lengths, and Sizes
Bounds, lengths, and sizes of simple parameters must be specified either by asterisks or by restricted expressions. When the actual length, bounds, or size can be different for different invocations, each can be specified in a DECLARE statement by an asterisk. When an asterisk is used, the length, bounds, or size are taken from the current generation of the associated argument.

An asterisk is not allowed as the length specification of a string that is an element of an aggregate, if the associated argument creates a dummy. The string length must be specified as an integer.

Controlled Parameter Bounds, Lengths, and Sizes
The bounds, length, or size of a controlled parameter can be specified in a DECLARE statement either by asterisks or by element expressions.

Asterisk Notation: When asterisks are used, length, bounds, or size of the controlled parameter are taken from the current generation of the associated argument. Any subsequent allocation of the controlled parameter uses these same bounds, length, or size, unless they are overridden by a different length, bounds, or size specification in the ALLOCATE statement. If no current generation of the argument exists, the asterisks determine only the dimensionality of the parameter, and an ALLOCATE statement in the invoked procedure must specify bounds, length, or size for the controlled parameter before other references to the parameter can be made.

Expression Notation: Each time the parameter is allocated, the expressions are evaluated to give current bounds, lengths, or sizes for the new allocation. However, such expressions in a DECLARE statement can be overridden by a bounds, length, or size specification in the ALLOCATE statement itself.

Example of array argument with parameters: In Figure 4, when Sub1 is invoked, A and B, which have been allocated, are passed.

```
%process or('|') num  margins(1,72);
Package:package exports(*);


Main: procedure options(main);
declare (A(NA), B(NB), C(NC), D(ND) ) controlled;
declare (NA init(20), NB init(30), NC init(100),
ND init(100) ) fixed bin(31);
declare Sub1 entry((*) controlled, (*) controlled);
declare Sub2 entry ((*) ctl, (*) ctl, fixed bin);


allocate A,B;  /* A(20), B(30) */
display ('Gen1: DIM(A)=' || dim(A) || ', ' || "DIM(B)=" || dim(B));
call Sub1(A,B);


display ('Gen2: Allocn(A)=' || allocn(a) || ', ' ||
'Allocn(B)=' || allocn(B) );
display ('Gen2: DIM(A)=' || dim(A) || ', ' || "DIM(B)=" || dim(B));
free A,B;
display ('Gen1: Allocn(A)=' || allocn(A) || ', ' ||
'Allocn(B)=' || allocn(B) );
display ('Gen1: DIM(A)=' || dim(A) || ', ' || "DIM(B)=" || dim(B));
free A,B;
display ('Gen0: Allocn(A)=' || allocn(A) || ', ' ||
'Allocn(B)=' || allocn(B) );
call Sub2 (C,D,10);


display ('Gen1: Allocn(C)=' || allocn(C) || ', ' ||
'Allocn(D)=' || allocn(D) );
display ('Gen1: DIM(C)=' || dim(C) || ', ' || "DIM(D)=" || dim(D));
free C,D;
display ('Gen0: Allocn(C)=' || allocn(c) || ', ' ||
'Allocn(D)=' || allocn(D) );
end Main;


Sub1: procedure (U,V);
dcl (U(UB), V(*)) controlled,
UB fixed bin(31);
display ('Gen1: Allocn(U)=' || allocn(U) || ', ' ||
'Allocn(V)=' || allocn(V) );
display ('Gen1: DIM(U)=' || dim(U) || ', ' || "DIM(V)=" || dim(V));
UB=200;
allocate U,V; /* U(200), V(30) */
display ('Gen2: Allocn(U)=' || allocn(U) || ', ' ||
'Allocn(V)=' || allocn(V) );
display ('Gen2: DIM(U)=' || dim(U) || ', ' || "DIM(V)=" || dim(V));
end Sub1;


Sub2: procedure (X,Y,N);
dcl (X(N),Y(N)) controlled,
N fixed bin;
display ('Gen0: Allocn(X)=' || allocn(X) || ', ' ||
'Allocn(Y)=' || allocn(Y) );
allocate X,Y;  /* X(10), Y(10) */
display ('Gen1: Allocn(X)=' || allocn(X) || ', ' ||
'Allocn(Y)=' || allocn(Y) );
display ('Gen1: DIM(X)=' || dim(X) || ', ' || "DIM(Y)=" || dim(Y));
end Sub2;


end Package;
```

Figure 4. Array argument with parameters

Controlled parameter bounds, lengths, and sizes

The ALLOCATE statement in `Sub1` allocates a second generation of `A` and `B`. `B` has the same bounds for both generations while `A` has different bounds for the second generation.

On returning to `Main`, the first FREE statement frees the second generation of `A` and `B` (allocated in `Sub1`). The second FREE statement frees the first generation of `A` and `B` (allocated in `Main`).

In `Sub2`, `X` and `Y` are declared with bounds that depend on the value of `N`. When `X` and `Y` are allocated, their values determine the bounds of the allocated arrays.

On returning to `Main` from `Sub2`, the FREE statement frees the only generation of `C` and `D` (allocated in `Sub2`).

Procedure activation

Sequential program flow passes around a procedure, from the statement before the PROCEDURE statement to the statement after the END statement for that procedure. The only way that a procedure can be activated is by a *procedure reference*. ("Program activation" on page 89 tells how to activate the main procedure.) The execution of the invoking procedure is suspended until the invoked procedure returns control to it.

A procedure reference is the appearance of an entry expression in one of the following contexts:

- Using a CALL statement to invoke a subroutine, as described in "CALL statement" on page 124
- Invoking a function, as described in "Functions" on page 106

The information in this section is relevant to each of these contexts. However, the examples in this chapter use CALL statements.

When a procedure reference occurs, the procedure containing the specified entry point is said to be *invoked*. The point at which the procedure reference appears is called the *point of invocation* and the block in which the reference is made is called the *invoking block*. An invoking block remains active even though control is transferred from it to the procedure it invokes.

When a procedure is invoked at its primary entry point, arguments and parameters are associated and execution begins with the first statement in the invoked procedure. When a procedure is invoked at a secondary entry point with the ENTRY statement, execution begins with the first statement following the ENTRY statement. The environment established on entry to a block at the primary entry point is identical to the environment established when the same block is invoked at a secondary entry point.

Communication between two procedures is by means of arguments passed from an invoking procedure to the invoked procedure, by a value returned from an invoked procedure, and by names known within both procedures. Therefore, a procedure can operate upon different data when it is invoked from different points. For example,

```
Readin: procedure;
statement-1
statement-2
```

```
Errt: entry;
statement-3
statement-4
end Readin;
```

can be activated by any of these entry references:

```
  call Readin;
  call Errt;
```

The statement `call Readin` invokes `Readin` at its primary entry point and execution begins with `statement-1`; the statement `call Errt` invokes the `Readin` procedure at the secondary entry point `Errt` and execution begins with `statement-3`. The entry constant (`Readin`) can also be assigned to an entry variable that is used in a procedure reference. For example:

```
  declare Readin entry,
          Ent1 entry variable;
  Ent1 = Readin;
  call Ent1;
  call Readin;
```

The two CALL statements have the same effect.

Procedure termination

A procedure is terminated when, by some means other than a procedure reference, control passes back to the invoking program, block, or to some other active block.

Procedures terminate *normally* when:
- Control reaches a RETURN statement within the procedure. The execution of a RETURN statement returns control to the point of invocation in the invoking procedure. If the point of invocation is a CALL statement, execution in the invoking procedure resumes with the statement following the CALL. If the point of invocation is a function reference, execution of the statement containing the reference is resumed.
- Control reaches the END statement of the procedure. Effectively, this is equivalent to the execution of a RETURN statement.

Procedures terminate *abnormally* when:
- Control reaches a GO TO statement that transfers control out of the procedure. The GO TO statement can specify a label in a containing block, or it can specify a parameter that has been associated with a label argument passed to the procedure. A STOP statement is executed in the current thread of a single-threaded program or in any thread of a multithreaded program.
- An EXIT statement is executed.
- The ERROR condition is raised and there is no established ON-unit for ERROR or FINISH. Also, if one or both of the conditions has an established ON-unit, ON-unit exit is by normal return rather than by a GO TO statement.
- The procedure calls or invokes another procedure that terminates abnormally.

Transferring control out of a procedure using a GO TO statement can sometimes result in the termination of several procedures and/or begin-blocks. Specifically, if the transfer point specified by the GO TO statement is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated. In the following example:

Procedure termination

```
A: procedure options(main);
   statement-1
   statement-2
   B: begin;
      statement-b1
      statement-b2
      call C;
      statement-b3
   end B;
   statement-3
   statement-4
   C: procedure;
      statement-c1
      statement-c2
      statement-c3
      D: begin;
         statement-d1
         statement-d2
         go to Lab;
         statement-d3
      end D;
      statement-c4
   end C;
   statement-5
 Lab: statement-6
   statement-7
 end A;
```

`A` activates `B`, which activates `C`, which activates `D`. In `D`, the statement `go to Lab` transfers control to `statement-6` in `A`. Since this statement is not contained in `D`, `C`, or `B`, all three blocks are terminated; `A` remains active. Thus, the transfer of control out of `D` results in the termination of intervening blocks `B` and `C` as well as the termination of block `D`.

Recursive procedures

An active procedure that is invoked from within itself or from within another active procedure is a *recursive* procedure. Such an invocation is called recursion.

A procedure that is invoked recursively must have the RECURSIVE attribute specified in the PROCEDURE statement.

```
►►──RECURSIVE────────────────────────────────────►◄
```

The environment (that is, values of automatic variables and the like) of every invocation of a recursive procedure is preserved in a manner analogous to the stacking of allocations of a controlled variable (see "Controlled storage and attribute" on page 224). Think of an environment as being *pushed down* at a recursive invocation, and *popped up* at the termination of that invocation. A label constant in the current block is always a reference to the current invocation of the block that contains the label.

If a label constant is assigned to a label variable in a particular invocation, and the label variable is not declared within the recursive procedure, a GO TO statement naming that variable in another invocation restores the environment that existed when the assignment was performed, terminating the current and any intervening procedures and begin-blocks.

The environment of a procedure that was invoked from within a recursive procedure by means of an entry variable is the one that was current when the entry constant was assigned to the variable. Consider the following example:

```
I=1;
call A;                                /* First invocation of A     */

A: proc recursive;
   declare Ev entry variable static;
   if I=1 then
      do;
        I=2;
        Ev=B;
        call A;                        /* 2nd invocation of A       */
      end;
   else call Ev;                       /* Invokes B with environment */
                                       /* of first invocation of A   */
B: proc;
   go to Out;
end B;
Out: end A;
```

The GO TO statement in the procedure B transfers control to the END A statement in the first invocation of A, and terminates B and both invocations of A.

Effect of recursion on automatic variables

The values of variables allocated in one activation of a recursive procedure must be protected from change by other activations. This is arranged by stacking the variables. A stack operates on a last-in, first-out basis. The most recent generation of an automatic variable is the only one that can be referenced. Static variables are not affected by recursion. Thus, they are useful for communication across recursive invocations. This also applies to automatic variables that are declared in a procedure that contains a recursive procedure and to controlled and based variables. In the following example:

```
A: proc;
dcl X;
 .
 .
 .
B: proc recursive;
 dcl Z,Y static;
 call B;
 .
 .
 .
end B;
end A;
```

A single generation of the variable X exists throughout invocations of procedure B. The variable Z has a different generation for each invocation of procedure B. The variable Y can be referred to only in procedure B and is not reallocated at each invocation. (The concept of stacking variables is also of importance in the discussion of controlled variables in "Controlled storage and attribute" on page 224.)

Dynamic loading of an external procedure

A DLL can be dynamically fetched (loaded) or released (deleted) by a PL/I program using FETCH and RELEASE statements.

A procedure invoked by a procedure reference usually is resident in main storage throughout the execution of the program. However, a procedure can be loaded into main storage for only as long as it is required. The invoked procedure can be dynamically loaded into, and dynamically deleted from, main storage during execution of the calling procedure.

Dynamic loading of an external procedure

Dynamic loading and deletion of procedures is particularly useful when a called procedure is not necessarily invoked every time the calling procedure is executed, and when conservation of main storage is more important than a short execution time.

The appearance of an entry constant in a FETCH statement indicates that the referenced procedure needs to be loaded into main storage before it can be executed, unless a copy already exists in main storage. Provided the name is referenced in a FETCH statement, a procedure can also be loaded from the disk by:
- Execution of a CALL statement or the CALL option of an INITIAL attribute
- Execution of a function reference.

It is not necessary that control pass through a FETCH or RELEASE statement, either before or after execution of the CALL or function reference.

Whichever statement loaded the procedure, execution of the CALL statement or option or the function reference invokes the procedure in the normal way.

It is not an error if the procedure has already been loaded into main storage. The fetched procedure can remain in main storage until execution of the whole program is completed. Alternatively, the storage it occupies can be freed for other purposes at any time by means of the RELEASE statement.

Rules and features

FETCH and RELEASE have the following rules and features:
- Only external procedures can be fetched.
- EXTERNAL files and CONDITION conditions are shared across the entire application (main and fetched modules). Other external variables are shared only within a single module.
-
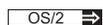
  OS/2

  For dynamically loaded (FETCHed) DLLs, you can set the STEPLIB environment variable to a specified search path. The path you specify in STEPLIB is searched before the path specified in the LIBPATH environment variable. For more information on STEPLIB and LIBPATH, see the Programming Guide.

- Storage for STATIC variables in the fetched procedure is allocated when the load module containing the procedure is loaded into memory. Each time a load module is loaded into memory, the STATIC variables are given the initial values indicated by their declarations.
- The FETCH and RELEASE statements must specify entry constants. An entry constant for a fetched procedure can be assigned to an entry variable provided the procedure has been fetched.

FETCH statement

The FETCH statement checks main storage for the named procedures. Procedures not already in main storage are loaded from the disk.

```
        ┌────────,◄──────────────────────────────────────┐
►►──FETCH─┴─entry-constant──┬──────────────────┬──┬──────────────────────┬─┴──────►
                            └─SET─(─ptr-ref─)──┘  └─TITLE─(─char-expr─)──┘

►──;─────────────────────────────────────────────────────►◄
```
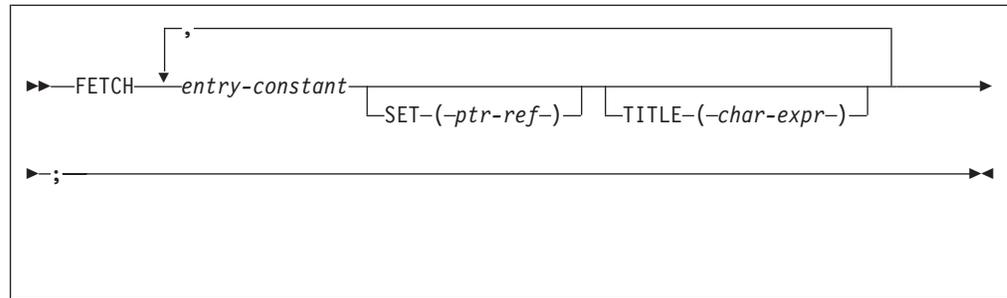
entry-constant
> Specifies the name by which the procedure to be fetched is known to the operating system. Details of the linking considerations for fetchable procedures are given in the Programming Guide.
>
> The entry-constant must be the same as the one used in the corresponding CALL statement, CALL option, or function reference.

SET
> Specifies a pointer reference (*ptr-ref*) that will be set to the address of the entry point of the loaded module. This option can be used to load tables (non-executable load modules). It can also be used for entries that are fetched and whose addresses need to be passed to non-PL/I procedures.
>
> If the load module is later released by the RELEASE statement, and the load module is accessed (through the pointer), unpredictable results can occur.

TITLE
> For TITLE, char-expr is any character expression or an expression that can be converted to a character expression. If TITLE is specified, the load module name specified is searched for and loaded. If it is not specified, the load module name used is the environment name specified in the EXTERNAL attribute for the variable (if present) or the entry constant name itself.
>
> For example:
>
> ```
> dcl A entry;
> dcl B entry ext('C');
> dcl T char(20) varying;
> T = 'Y';
>
> fetch A title('X');        /* X is loaded */
> fetch A;                   /* A is loaded */
> fetch B title('Y');        /* Y is loaded */
> fetch B;                   /* C is loaded */
> fetch B title(T);   /* Y is loaded */
> ```
>
> For more detailed information on title strings, refer to the Programming Guide.

RELEASE statement

The RELEASE statement frees the main storage occupied by procedures identified by its specified entry constants.

```
┌─────────,─────────┐
▶▶──RELEASE──┬─▼──entry-constant──┴──;──────────────────▶◀
             └─*─┘
```
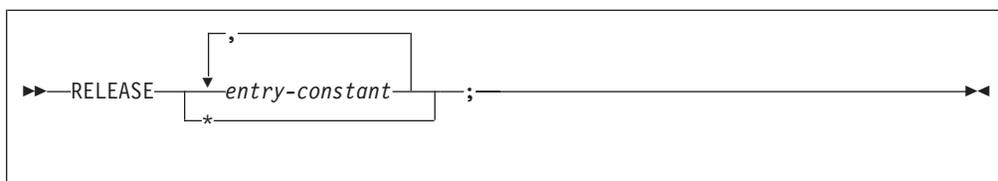
entry constant

Must be the same as the one used in the corresponding CALL statement, CALL option, function reference, and FETCH statements. RELEASE * releases all previously FETCHed PL/I modules. It must not be executed from within a FETCHed module.

Consider the following example, in which `ProgA` and `ProgB` are entry names of procedures resident on disk:

```
      Prog:  procedure;
```

|1| ```
          fetch ProgA;
```
|2| ```
          call ProgA;
```
|3| ```
          release ProgA;
```

|4| ```
          call ProgB;
          go to Fin;
```

```
          fetch ProgB;
      Fin:  end Prog;
```

|1| `ProgA` is loaded into main storage by the first FETCH statement.

|2| `ProgA` executes when the first CALL statement is reached.

|3| Storage for `ProgA` is released when the RELEASE statement is executed.

|4| `ProgB` is loaded and executed when the second CALL statement is reached, even though the FETCH statement referring to this procedure is never executed.

The same results would be achieved if the statement FETCH `ProgA` were omitted. The appearance of `ProgA` in a RELEASE statement causes the statement CALL `ProgA` to load the procedure, as well as invoke it.

The fetched procedure is compiled and linked separately from the calling procedure. You must ensure that the entry constant specified in FETCH, RELEASE, and CALL statements; CALL options; and in function references is the name known on the disk. This is discussed in the Programming Guide.

Subroutines

A *subroutine* is an internal or external procedure that is invoked by a CALL statement. For the syntax of a subroutine, see "Procedures" on page 92.

The arguments of the CALL statement are associated with the parameters of the invoked procedure. The subroutine is activated, and execution begins. The arguments (zero or more) can be input only, output only, or both.

A subroutine is normally terminated by the RETURN or the END statement. Control is then returned to the invoking block. A subroutine can be abnormally terminated as described in "Procedure termination" on page 99.

A subroutine procedure must:

- Not have the RETURNS option on the procedure statement
- Not be declared as an entry with the RETURNS attribute if it is an external procedure
- Be invoked using the CALL statement, not a function reference
- Not return a result value using the RETURN statement

The following examples illustrate the invocation of subroutines that are external to and internal to the invoking block.

Example 1

```
          Prmain:  procedure;
                   declare Name character (20),
                   Item bit(5),
 4                 Outsub entry;
 1                 call Outsub (Name, Item);
          end Prmain;

 2        Outsub:  procedure (A,B);
                   declare A character (20),
                   B bit(5);
 3                 put list (A,B);
          end Outsub;
```

1   The CALL statement in `Prmain` invokes the procedure `Outsub` in **2** with the arguments `Name` and `Item`.

2   `Outsub` associates `Name` and `Item` passed from `Prmain` with its parameters, `A` and `B`. When `Outsub` is executed, each reference to `A` is treated as a reference to `Name`. Each reference to `B` is treated as a reference to `Item`.

3   The `put list (A,B)` statement transmits the values of `Name` and `Item` to the default output file, SYSPRINT.

4   In the declaration of `Outsub` as an entry constant, no parameter descriptor has to be given with the ENTRY attribute, because the attributes of the arguments and parameters match. Also see "ENTRY attribute" on page 113.

Example 2

```
          A:  procedure;
              declare Rate float (10),
                      Time float(5),
                      Distance float(15),
                      Master file;
 1            call Readcm (Rate, Time, Distance, Master);
 3          Readcm:
 2            procedure (W,X,Y,Z);
              declare W float (10),
                      X float(5),
                      Y float(15), Z file;
              get File (Z) list (W,X,Y);
              Y = W * X;
              if Y > 0 then
                return;
              else
                put list('ERROR READCM');
              end Readcm;

            end A;
```

1   The arguments `Rate`, `Time`, `Distance`, and `Master` are passed to the procedure `Readcm` in **3** and associated with the parameters W, X, Y, and Z.

2 A reference to `W` is the same as a reference to `Rate`, `X` the same as `Time`, `Y` the same as `Distance`, and `Z` the same as `Master`.

3 Note that `Readcm` is not explicitly declared in `A`. It is implicitly declared with the ENTRY attribute by its specification on the PROCEDURE statement.

Built-in subroutines

You can use *built-in subroutines*, which provide ready-made programming tasks. Their *built-in names* can be explicitly declared with the BUILTIN attribute. (For more information on the BUILTIN attribute or for the description of any built-in function, see Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365.)

Functions

A *function* is a procedure that has zero or more arguments and is invoked by a *function reference* in an expression. The function reference transfers control to a function procedure; the function procedure returns control and a value, which replaces the function reference in the evaluation of the expression. Aggregates cannot be returned; ENTRY variables cannot be returned unless they have the LIMITED attribute. The evaluation of the expression then continues.

A function procedure must:
- Have the RETURNS option on the procedure statement.
- Be declared as an entry with the RETURNS attribute, if it is an external procedure.
- Be invoked using a function reference. The CALL statement can be used to invoke it only if the returned value has the OPTIONAL attribute. In this case, the returned value is discarded upon return. Using END instead of RETURN can cause unpredictable results.
- Have matching attributes in the RETURNS option and in the RETURNS attribute.
- Use the RETURN statement to return control and the result value.

Whenever a function is invoked, the arguments in the invoking expression are associated with the parameters of the entry point. Control is then passed to that entry point. The function is activated and execution begins.

The RETURN statement terminates a function and returns the value specified in its expression to the invoking expression. See "RETURN statement" on page 125 for more information.

A function can be abnormally terminated as described in "Procedure termination" on page 99. If this method is used, evaluation of the expression that invoked the function is not completed, and control goes to the designated statement.

In some instances, a function can be defined so that it does not require an argument list. In such cases, the appearance of an external function name within an expression is recognized as a function reference only if the function name has been explicitly declared as an entry name. See "Entry invocation or entry value" on page 124 for additional information.

Examples

The following examples illustrate the invocation of functions that are internal to and external to the invoking block.

Example 1
In the following example, the assignment statement contains a reference to the
Sprod function:

```
      Mainp: procedure;
             get list (A, B, C, Y);
 1           X = Y**3+Sprod(A,B,C);

 2    Sprod: procedure (U,V,W)
             returns (bin float(21));
             dcl (U,V,W) bin float(53);
             if U > V + W then
 3             return (0);
             else
 3               return (U*V*W);
       end Sprod;
```

1   When Sprod is invoked, the arguments A, B, and C are associated with the
      parameters U, V, and W in 2 , respectively.

2   Sprod is a function because RETURNS appears in the procedure statement.
      It is internal, and therefore needs no explicit entry declaration. If Sprod were
      external, Mainp would contain an entry declaration with RETURNS
      specified.

3   Sprod returns either zero or the value represented by U*V*W, along with
      control to the expression in Mainp. The returned value is taken as the value
      of the function reference, and evaluation of the expression continues.

Example 2

```
      Mainp:  procedure;
             dcl Tprod entry (bin float(53),
                              bin float(53),
                              bin float(53),
                              label) external
               returns (bin float(21));
             get list (A,B,C,Y);
 1           X = Y**3+Tprod(A,B,C,Lab1);
      Lab1:  call Errt;
      end Mainp;
 1     Tprod:  procedure (U,V,W,Z)
               returns (bin float(21));
               dcl (U,V,W) bin float(53);
               declare Z label;

 2             if U > V + W then
                   go to Z;
 3             else
                   return (U*V*W);
        end Tprod;
```

1   When Tprod is invoked, Lab1 is associated with parameter Z.

2   If U is greater than V + W, control returns to Mainp at the statement labeled
      Lab1. Evaluation of the assignment in 1 is discontinued.

3   If U is not greater than V + W, U*V*W is calculated and returned to Mainp in
      the normal fashion. Evaluation of the assignment in 1 continues.

Notice that Tprod is an external procedure. It has an explicit entry declaration in
Mainp, which contains RETURNS.

Built-in functions

Besides allowing programmer-written function procedures, PL/I provides a set of *built-in functions*. Built-in functions include commonly used arithmetic functions, as well as functions for manipulating strings and arrays, using storage, and others. You invoke built-in functions the same way that you invoke programmer-defined functions. However, many built-in functions can return an array of values, whereas a programmer-defined function can return only an element value. The built-in names for built-in functions can be explicitly declared with the BUILTIN attribute. (For more information on the BUILTIN attribute or for the description of any built-in function, see Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365.)

Passing arguments to procedures

When a function or a subroutine is invoked, parameters are associated, from left to right, with the passed arguments.

In general:
- Computational data arguments can be passed to parameters of any computational data type.
- Program-control data arguments must be passed to parameters of the same type, with these exceptions.
    - Pointer and offset can be passed to each other.
    - LIMITED ENTRY can be passed to ENTRY, but ENTRY cannot be passed to LIMITED ENTRY.
    - An array of label constants cannot be used as an argument.

Arguments that require aggregate temporaries derived from structures are not allowed, unless the structure argument is declared with constant extents.

Expressions in the argument list are evaluated in the invoking block before the subroutine or function is invoked. A parameter has no storage associated with it. It is merely a means of allowing the invoked procedure to access storage allocated in the invoking procedure.

Using BYVALUE and BYADDR

Unless an argument is passed BYVALUE, a reference to an argument, not its value, is generally passed to a subroutine or function. This is known as passing arguments by reference, or BYADDR. A reference to a parameter in a procedure is a reference to the corresponding argument. Any change to the value of a parameter is actually a change to the value of the corresponding argument. However, this is not always possible or desirable. Constants, for example, should not be altered by an invoked procedure. For arguments that should not change, a *dummy argument* containing the value of the original argument is passed. Any reference to the parameter then is a reference to the dummy argument and not to the original argument.

When you specify BYADDR, the compiler puts the address of the corresponding argument in the parameter list. When you specify BYVALUE, puts the value of the argument in the parameter list.

When you specify BYVALUE, a dummy argument is notcreated; however, as is also true for dummy arguments, any change to the corresponding parameter in the called routine will not be visible in the calling routine.

BYVALUE can be specified only for scalar arguments and parameters that have lengths and sizes known at compile time.

Dummy arguments

A dummy argument is created when the argument is any of the following:
- A constant (unless the parameter has the NONASSIGNABLE attribute).
- An expression with operators, parentheses, or function references.
- A variable whose data attributes, alignment attributes, or connected attribute are different from the attributes declared for the parameter.

  This does not apply to noncontrolled parameters when only bounds, lengths, or size differ and these are declared with asterisks, nor when an expression other than a constant is used to define the extents of a controlled parameter. In the latter case, argument and parameter extents are assumed to match.

  In the case of an argument and parameter with the PICTURE attribute, a dummy argument is created unless the picture specifications match exactly, after any repetition factors are applied. The only exception is that an argument or parameter with a + sign in a scaling factor matches a parameter or argument without the + sign.
- A controlled string or area (because an ALLOCATE statement could change the length or extent).
- A string or area with an adjustable length or size that is associated with a noncontrolled parameter whose length or size is a constant.

Deriving dummy argument attributes
PL/I derives the attributes of dummy arguments from:
- The attributes declared for the associated parameter in an internal procedure.
- The attributes specified in the parameter descriptor for the associated parameter in the declaration of the external entry. If there was not a descriptor for this parameter, the attributes of the constant or expression are used.
- The extents (when specified by an asterisk in a declaration) of the argument for the bounds of an array, the length of a string, or the size of an area.

Rules for dummy arguments
The following rules apply to dummy arguments:
- If a parameter is an element (that is, a variable that is neither a structure nor an array), the argument must be an element expression.
- When a VARYING or VARYINGZ string element is passed to a NONVARYING parameter, whose length is undefined (that is, specified by an asterisk), a dummy argument with the current length of the original is created.
- Entry variables passed as arguments are assumed to be aligned; therefore, no dummy argument is created when only the alignments of argument and parameter differ. See "Generic entries" on page 121, for a description of generic name arguments for entry parameters.
- If the parameter is of the program-control data type (except locator), the argument must be a reference of the same data type.
- If a parameter is a locator (pointer or offset), the argument must be a locator. If the types differ, a dummy argument is created. The parameter descriptor of an offset parameter must not specify an associated area.
- A noncontrolled parameter can be associated with an argument of any storage class. However, if more than one generation of the argument exists, the parameter is associated only with that generation existing at the time of invocation.

- If the parameter is controlled, you must explicitly state this in the parameter descriptor for the ENTRY declaration. In addition, a controlled parameter must always have a corresponding controlled argument that:
  - is not subscripted
  - is not an element of a structure
  - does not cause a dummy to be created

If more than one generation of the argument exists at the time of invocation, the parameter corresponds to the entire stack of generations in existence. Consequently, at the time of invocation, a controlled parameter represents the current generation of the corresponding argument. A controlled parameter can be allocated and freed in the invoked procedure, allowing the manipulation of the allocation stack of the associated argument.

If the extents of the controlled parameter are specified as asterisks or nonrestricted expressions, the original declaration must have extents declared as nonrestricted expressions.

Passing arguments to the MAIN procedure

The PROCEDURE statement for the main procedure can have a parameter list. Such parameters require no special considerations in PL/I. However, you must be aware of any requirements of the invoking program (for example, when not to use such a parameter as the target of an assignment).

When the invoking program is the operating system and when compiled with the SYSTEM(MVS) compiler option:

- A single argument is passed to the MAIN procedure, and that parameter must be declared as CHARACTER VARYING.
- The current length of this parameter is set equal to the argument length at run-time. So, in the following example:

```
Tom:  proc (Param) options (main);
   dcl Param char(100) varying;
```

storage is allocated only for the current length of the argument.
- The contents of this parameter depend on a second option that may be specified along with OPTIONS(MAIN):
  - If you specify OPTIONS(MAIN NOEXECOPS), then the string passed by the operating system to PL/I is passed as is to your program. NOEXECOPS is recommended.
  - If you specify only OPTIONS(MAIN), then the string passed by the operating system to PL/I is stripped of all characters up to and including the first '/'. This means that if the string contains no '/', then your program receives a null string.

Begin-blocks

A begin-block is a sequence of statements delimited by a BEGIN statement and a corresponding END statement. For example:

```
B:  begin;
statement-1
statement-2
 :
statement-n
end B;
```

BEGIN statement

The BEGIN statement and a corresponding END statement delimit a begin-block.

```
►►──BEGIN──────────────────────────────;─────────────────►◄
          └─OPTIONS─(─options─)─┘
```
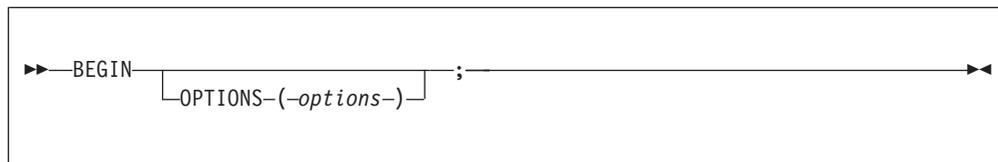
OPTIONS option
For begin-block options, refer to "OPTIONS option and attribute" on page 126.

Begin-block activation

Begin-blocks are activated through sequential flow or as a unit in an IF, ON, WHEN, or OTHERWISE statement.

You can transfer control to a labeled BEGIN statement using the GO TO statement.

Begin-block termination

A begin-block is terminated when control passes to another active block by some means other than a procedure reference. These means are:

* The END statement for the begin-block is executed. Control continues with the statement physically following the END, except when the block is an ON-unit.
* A GO TO statement within the begin-block (or within any block internal to it) is executed, transferring control to the point outside the block.
* A STOP or an EXIT statement is executed.
* Control reaches a RETURN statement that transfers control out of the begin-block and out of its containing procedure.

A GO TO statement can also terminate other blocks if the transfer point is contained in a block that did not directly activate the block being terminated. In this case, all intervening blocks in the activation sequence are terminated. For an example of this, see the example in "Procedure termination" on page 99.

Entry data

The entry data can be an entry constant or the value of an entry variable.

An entry constant is a name prefixed to a PROCEDURE or ENTRY statement, or a name declared with the ENTRY attribute and not the VARIABLE attribute. It can be assigned to an entry variable. In the following example, P, E1, and E2 are entry constants. Ev is an entry variable.

```
P:  procedure;
    declare Ev entry variable,
     (E1,E2) entry;


  Ev = E1;
  call Ev;
  Ev = E2;
  call Ev;
```

The first CALL statement invokes the entry point E1. The second CALL invokes the entry point E2.

Entry data

The following example declares `F(5)`, a subscripted entry variable.

The five entries `A`, `B`, `C`, `D`, and `E` are each invoked with the parameters `X`, `Y`, and `Z`.

```
declare (A,B,C,D,E) entry,
declare F(5) entry variable initial (A,B,C,D,E);
do I = 1 to 5;
  call F(I) (X,Y,Z);
end;
```

When an entry constant that is an entry point of an internal procedure is assigned to an entry variable, the assigned value remains valid only as long as the block that the entry constant was internal to remains active (and, for recursive procedures, remains current).

Entry constants

The appearance of a label prefix to a PROCEDURE or ENTRY statement explicitly declares an entry constant. A parameter-descriptor list is obtained from the parameter declarations, if any, and by defaults.

External entry constants must be explicitly declared. This declaration:

- Defines an entry point to an external procedure.
- Optionally specifies a parameter-descriptor list (the number of parameters and their attributes), if any, for the entry point.
- Specifies the attributes of the value that is returned by the procedure if the entry is a function.
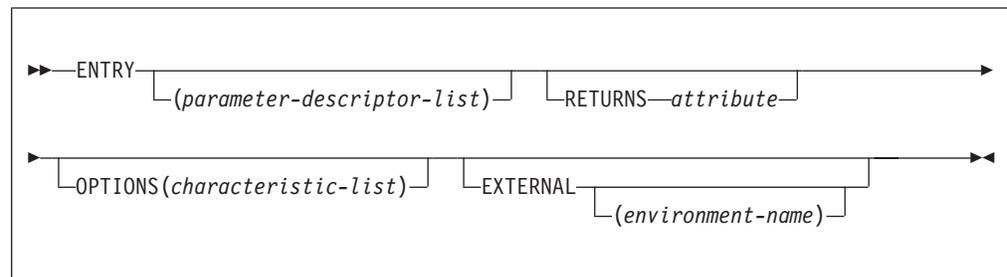
The attributes can appear in any order.

ENTRY attribute
  For complete ENTRY attribute syntax, refer to "ENTRY attribute" on page 113.

OPTIONS attribute
  For complete OPTIONS attribute syntax, refer to "OPTIONS option and attribute" on page 126.

RETURNS attribute
  For complete RETURNS attribute syntax, refer to "RETURNS option and attribute" on page 134.

EXTERNAL attribute
  If you don't specify an *environment-name*, the name is the same as the declaration. For a complete description of the EXTERNAL attribute refer to "INTERNAL and EXTERNAL attributes" on page 153.

Entry variables

An entry variable can contain both internal and external entry values. It can be part of an aggregate. For structuring and array dimension attributes, refer to "Arrays" on page 167 and "Structures" on page 170.

```
►►──ENTRY──────────────────────────────────────────────────────────────────►
            └─(parameter-descriptor-list)─┘  └─OPTIONS(characteristic-list)─┘

►──────────────────────────────────────────────────────────────────────────►
   └─VARIABLE─┘  └─LIMITED─┘  └─RETURNS──attribute─┘

►──────────────────────────────────────────────────────────────────────────►◄
   └─EXTERNAL─┬───────────────────────┬─┘
             └─(environment-name)─┘
```

The options can appear in any order.

ENTRY attribute
> Refer to "ENTRY attribute."

OPTIONS attribute
> Refer to "OPTIONS option and attribute" on page 126.

VARIABLE attribute
> The VARIABLE attribute establishes the name as an entry variable. This variable can contain entry constants and variables. Refer to "VARIABLE attribute" on page 49 for syntax information.

LIMITED attribute
> Refer to "LIMITED attribute" on page 120.

RETURNS attribute
> Refer to "RETURNS option and attribute" on page 134.

EXTERNAL attribute
> Refer to "Scope of declarations" on page 151.

ENTRY attribute

The ENTRY attribute specifies that the name being declared is either an external entry constant, or an entry variable. It also describes the attributes of the parameters of the entry point.

ENTRY

```
►►──ENTRY──────────────────────────────────────────────────────►◄
         │        ┌─,──────────────────────┐              │
         │        ▼                         │              │
         └──(──┬───parameter-descr────────┬──)──┘
               └───structure-union-descr──┘
```

parameter-descr:

```
       ┌──────────────┐
       ▼              │
  ├──┬───attribute────┴────────────────────────────────────────────┤
     │                          ┌─OPTIONAL─┐
     └─*──┬──────────────┬──────┴──────────┘
          ├─ALIGNED───┐
          └─UNALIGNED─┘
          ├─ASSIGNABLE────┐
          └─NONASSIGNABLE─┘
          ├─CONNECTED────┐
          └─NONCONNECTED─┘
          ├─BYADDR──┐
          └─BYVALUE─┘
```

structure-union-descr:

```
           ┌───────────┐              ┌───────────┐
           ▼           │              ▼           │
  ├──1──┬────attribute──┴──┬──────────┬──,──level─┴──attribute──┤
        │                  └─OPTIONAL─┘
```

ENTRY

The ENTRY attribute, without a parameter descriptor list, is implied by the RETURNS attribute.

parameter-descr (parameter-descriptor)

A parameter descriptor list can be given to describe the attributes of the parameters of the associated external entry constant or entry variable. It is used for argument and parameter attribute matching and the creation of dummy arguments.

If no parameter descriptor list is given, the default is for the argument attributes to match the parameter attributes. Thus, the parameter descriptor list must be supplied if argument attributes do not match the parameter attributes.

Each parameter descriptor corresponds to one parameter of the entry point invoked and, if given, specifies the attributes of that parameter.

The parameter descriptors must appear in the same order as the parameters they describe. If a descriptor is absent, the default is for the argument to match the parameter.

If a descriptor for a parameter is not required, the absence of the descriptor must be indicated by an asterisk. For example:

```
entry(character(10),*,*,fixed dec)        Indicates four arguments.
```

`entry(*)`	Indicates one argument.
`entry( )`	Specifies that the entry name must never have any arguments.
`entry`	Specifies that it can have any number of arguments.
`entry(float binary,*)`	Indicates two arguments.

attribute

The allowed attributes are any of the data attributes listed under "Data attributes" on page 23. The attributes can appear in any order in a parameter descriptor. For an array parameter-descriptor, the dimension attribute must be the first one specified.

* An asterisk specifies that, for that parameter, any data type is allowed. The only attributes which are valid following the asterisk are:
- ALIGNED or UNALIGNED
- ASSIGNABLE or NONASSIGNABLE
- BYADDR or BYVALUE
- CONNECTED or NONCONNECTED
- OPTIONAL

No conversions are done.

OPTIONAL

This attribute is discussed in "OPTIONAL attribute" on page 116.

structure-union-descr (structure-union-descriptor)

For a structure-union descriptor, the descriptor level-numbers need not be the same as those of the parameter, but the structuring must be identical. The attributes for a particular level can appear in any order.

Defaults are not applied if an asterisk is specified. For example, in the following declaration defaults are applied only for the second parameter.

```
dcl X entry(* optional, aligned); /* defaults applied for 2nd parm */
```

Extents (lengths, sizes, and bounds) in parameter descriptors must be specified as constants or as asterisks. Controlled parameters must have asterisks.

RETURNS attribute implies the ENTRY attribute. For example:

Example parameter descriptors   **Declarations for example descriptors**

```
  Test:  procedure (A,B,C,D,E,F);          declare Test entry
                                                  (decimal fixed (5),
                                                   binary float (21),
   declare A fixed decimal (5),                 *,
           B float binary (21),                 1,
           C pointer,                           2,
           1 D,                                 2,
             2 P,                                3 decimal fixed,
             2 Q,                               *,
               3 R fixed decimal,             (4) char(10));
           1 E,
             2 X,
             2 Y,
               3 Z,
           F(4) character (10);
   end Test;
```

In the previous example, the parameter C, and the structure parameter E do not have descriptors.

OPTIONAL attribute

OPTIONAL can be specified as part of the parameter-descriptor list or as an attribute in the parameter declaration.

```
►►──OPTIONAL─────────────────────────────────────────────────►◄
```
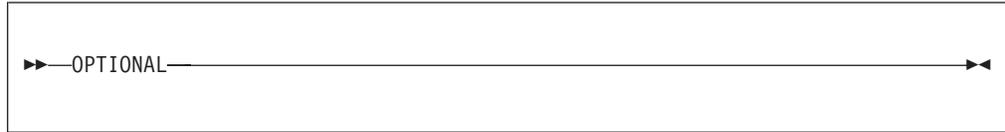
OPTIONAL arguments can be omitted in calls and function references by specifying an asterisk for the argument. An omitted item can be anywhere in the argument list, including at the end. However, the omitted item is counted as an argument. With its inclusion in an entry, the number of arguments must not exceed the maximum number allowed for the entry.

Using OPTIONAL and BYVALUE for the same item is invalid, unless the item is a LIMITED ENTRY.

The receiving procedure can use the OMITTED built-in function to determine if an OPTIONAL parameter/argument was omitted in the invocation of the entry. (For more information on the OMITTED built-in function, see "OMITTED" on page 442.)

Figure 5 on page 117 shows both valid and invalid CALL statements for the procedure Vrtn.

```
  Caller: proc;
    dcl Vrtn entry (
            fixed bin,
            ptr optional,
            float,
            * optional);

/*  The following calls are valid:  */

    call Vrtn(10, *, 15.5, 'abcd');
    call Vrtn(10, *, 15.5, *);
    call Vrtn(10, addr(x), 15.5, *);

/*  The following calls are invalid:  */

    call Vrtn(10, addr(x), 15.5);
    call Vrtn(*, addr(x));
    call Vrtn(10,addr(x));
    call Vrtn(10);
    call Vrtn;
  end Caller;

  Vrtn: proc (Fb, P, Fl, C1);
    dcl Fb fixed bin,
        P ptr optional,
        Fl float,
        C1 char(8) optional;

    if ¬omitted(C1) then display (C1);
    if ¬omitted(P) then P=P+10;
end;
```

Figure 5. Valid and invalid call statements

`Vrtn` determines if OPTIONAL parameters were omitted, and takes the appropriate action.

LIST attribute

LIST can be specified on the last parameter in a parameter-descriptor list or as an attribute on the last parameter to a procedure.

```
►►──LIST───────────────────────────────────────────────────►◄
```
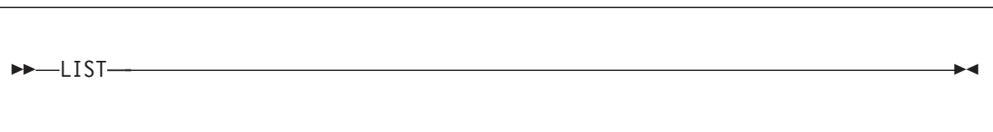
When the LIST attribute is specified in an entry declaration, it indicates that zero or more additional arguments may be passed to that entry. For example, the following declare specifies that `vararg` must be invoked with one character varyingz parameter and may be invoked with any number of other parameters.

```
  dcl vararg  external
            entry( list byaddr char(*) varz nonasgn )
            options( nodescriptor byvalue );
```

When the LIST attribute is specified in the declaration of the last parameter in a procedure, it indicates that zero or more additional arguments may have been passed to that procedure.

When the LIST attribute is specified, no descriptors are allowed.

The address of the first of these additional parameters may be obtained via the VARGLIST built-in function. This address may be used to obtain the addresses of any additional parameters as follows:

- if the additional parameters to this procedure were passed byvalue, successively incrementing this initial address by the value returned by the VARGSIZE built-in function will return the addresses of any additional parameters

- if the additional parameters to this procedure were passed byaddr, successively incrementing this initial address by the size of a pointer will return the addresses of any additional parameters

The following sample program, which implements a simple version of printf, illustrates how to use the LIST attribute. The routine varg1 illustrates how to walk a variable argument list with byvalue parameters, while varg2 illustrates how to walk such a list with byaddr parameters.

```
*process rules(ans) dft(ans) gn;
*process langlvl(saa2);

 vararg: proc options(main);

   dcl i1      fixed bin(31) init(1729);
   dcl i2      fixed bin(31) init(6);
   dcl d1      float bin(53) init(17.29);

   call varg1( 'test byvalue' );
   call varg1( 'test1 parm1=%i', i1 );
   call varg1( 'test2 parm1=%i parm2=%i', i1, i2 );
   call varg1( 'test3 parm1=%d', d1 );

   call varg2( 'test byaddr' );
   call varg2( 'test1 parm1=%i', i1 );
   call varg2( 'test2 parm1=%i parm2=%i', i1, i2 );
   call varg2( 'test3 parm1=%d', d1 );
 end;

*process ;
  varg1:
    proc( text )
    options( nodescriptor byvalue );

    dcl text    list byaddr nonasgn varz char(*);

    dcl jx      fixed bin;
    dcl iz      fixed bin;
    dcl ltext   fixed bin;
    dcl ptext   pointer;
    dcl p       pointer;
    dcl i       fixed bin(31) based;
    dcl d       float bin(53) based;
    dcl q       float bin(64) based;
    dcl chars   char(32767) based;
    dcl ch      char(1) based;
```

Figure 6. Sample program illustrating LIST attribute (Part 1 of 3)

```
      ptext = addr(text);
      ltext = length(text);
      iz = index( substr(ptext->chars,1,ltext), '%' );
      p = varglist();
      do while( iz > 0 );
        if iz = 1 then;
        else
          put edit( substr(ptext->chars,1,iz-1) )(a);
        ptext += iz;
        ltext -= iz;
        select( ptext->ch );
          when( 'i' )
            do;
              put edit( trim(p->i) )(a);
              p += vargsize( p->i );
            end;
          when( 'd' )
            do;
              put edit( trim(p->d) )(a);
              p += vargsize( p->d );
            end;
        end;
        ptext += 1;
        ltext -= 1;
        if ltext <= 0 then leave;
        iz = index( substr(ptext->chars,1,ltext), '%' );
      end;
      if ltext = 0 then;
      else
        put edit( substr(ptext->chars,1,ltext) )(a);
      put skip;
    end;
```

Figure 6. Sample program illustrating LIST attribute (Part 2 of 3)

```
*process ;
  varg2:
    proc( text )
    options( nodescriptor byaddr );

    dcl text    list byaddr nonasgn varz char(*);

    dcl jx      fixed bin;
    dcl iz      fixed bin;
    dcl ltext   fixed bin;
    dcl ptext   pointer;
    dcl p       pointer;
    dcl p2      pointer based;
    dcl i       fixed bin(31) based;
    dcl d       float bin(53) based;
    dcl q       float bin(64) based;
    dcl chars   char(32767) based;
    dcl ch      char(1) based;

    ptext = addr(text);
    ltext = length(text);
    iz = index( substr(ptext->chars,1,ltext), '%' );
    p = varglist();
    do while( iz > 0 );
      if iz = 1 then;
      else
        put edit( substr(ptext->chars,1,iz-1) )(a);
      ptext += iz;
      ltext -= iz;
      select( ptext->ch );
        when( 'i' )
          do;
            put edit( trim(p->p2->i) )(a);
            p += size( p );
          end;
        when( 'd' )
          do;
            put edit( trim(p->p2->d) )(a);
            p += size( p );
          end;
      end;
      ptext += 1;
      ltext -= 1;
      if ltext <= 0 then leave;
      iz = index( substr(ptext->chars,1,ltext), '%' );
    end;
    if ltext = 0 then;
    else
      put edit( substr(ptext->chars,1,ltext) )(a);
    put skip;
  end;
```

Figure 6. Sample program illustrating LIST attribute (Part 3 of 3)

LIMITED attribute

The LIMITED attribute indicates that the entry variable has only non-nested entry constants as values. A entry variable that is not LIMITED can have any entry constants as values.
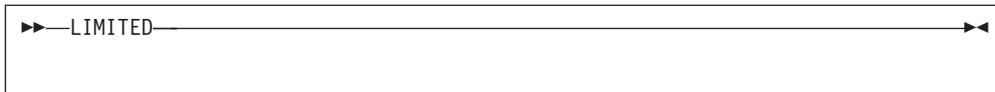
►►──LIMITED──►◄

Example:

```
Example: proc options(reorder reentrant);
dcl (Read, Write) entry;
dcl FuncRtn(2) entry limited
static init (Read, Write);


dcl (Prt1) entry;
dcl PrtRtn(2) entry variable limited
static init (Prt1,      /* legal   */
             Prt2);     /* illegal */
Prt2: proc;
:
end Prt2;
end Example;
```

A LIMITED static entry variable can be initialized with the value of a non-nested entry constant, thus allowing generation of more efficient code. It also uses less storage than a non-LIMITED entry variable.

Generic entries

A generic entry declaration specifies a generic name for a set of entry references and their descriptors. During compilation, invocation of the generic name is replaced by one of the entries in the set.

GENERIC attribute

The generic name must be explicitly declared with the GENERIC attribute.

GENERIC attribute

```
►►──generic-name──GENERIC──(──┤ references ├──────────────────────────────►
                                            └─,──entry-ref──OTHERWISE─┘

►──)──;──┤ generic-descriptor: ├──data-attributes─────────────────────────►
                                                  ├─ALIGNED───┤
                                                  └─UNALIGNED─┘

►──┬─ASSIGNABLE──────────┬──┬─CONNECTED─────────┬──┬─HEXADEC─┬─────────────►
   │              (1)     │  │            (1)    │  └─IEEE────┘
   └─NONASSIGNABLE────────┘  └─NONCONNECTED──────┘

►──┬─BIGENDIAN────┬──┬─OPTIONAL─┬─────────────────────────────────────────►◄
   └─LITTLEENDIAN─┘  └──────────┘
```

references

```
    ┌─,──────────────────────────────────────────┐
├────▼──entry-ref──WHEN─(──┬──────────────────┬──)──┬──────────────────┤
                           │  ┌─,───────────┐ │
                           └──▼─generic-descriptor─┘
                              └─*────────────┘
```

Notes:

1   If specified, this keyword is ignored.
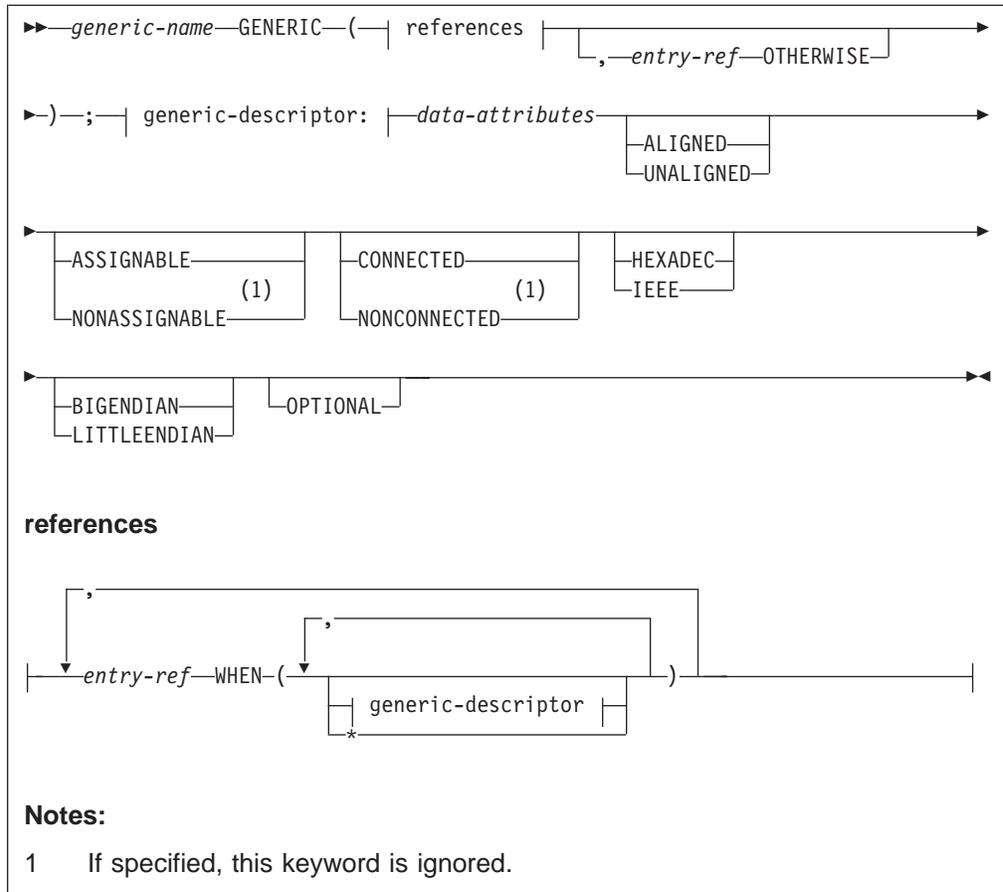
Abbreviation: OTHER for OTHERWISE

For the general declaration syntax, see "DECLARE statement" on page 148.

entry-ref
>   Must not be subscripted or defined. The same entry reference can appear more than once within a single GENERIC declaration with different lists of descriptors.

generic-descriptor
>   Corresponds to a single argument. It specifies an attribute that the corresponding argument must have so that the associated entry reference can be selected for replacement.
>
>   Structures or unions cannot be specified.
>
>   Where a descriptor is not required, its absence must be indicated by an asterisk.
>
>   The descriptor that represents the absence of all arguments in the invoking statement is expressed by omitting the generic descriptor in the WHEN clause of the entry. It has the form:
>
>   ```
>   generic (... entry1 when( ) ...)
>   ```

data-attributes
>   Listed in "Data types and attributes" on page 22.

ALIGNED and UNALIGNED
>   Discussed in "ALIGNED and UNALIGNED attributes" on page 159.

ASSIGNABLE and NONASSIGNABLE
> Discussed in "ASSIGNABLE and NONASSIGNABLE attributes" on page 242.

CONNECTED and NONCONNECTED
> Discussed in "CONNECTED and NONCONNECTED attributes" on page 245.

HEXADEC and IEEE
> Discussed in "HEXADEC and IEEE attributes" on page 244.

BIGENDIAN and LITTLEENDIAN
> Discussed in "BIGENDIAN and LITTLEENDIAN attributes" on page 243.

OPTIONAL
> Discussed in "OPTIONAL attribute" on page 116.

When an invocation of a generic name is encountered, the number of arguments specified in the invocation and their attributes are compared with descriptor list of each entry in the set. The first entry reference for which the descriptor list matches the arguments both in number and attributes replaces the generic name.

In the following example, an entry reference that has exactly two descriptors with the attributes DECIMAL or FLOAT, and BINARY or FIXED is searched for.

```
declare Calc generic (
            Fxdcal when (fixed,fixed),
            Flocal when (float,float),
            Mixed  when (float,fixed),
            Error otherwise);
Dcl    X decimal float (6),
       Y binary fixed (15,0);


       Z = X+Calc(X,Y);
```

If an entry with the exact number of descriptors with the exact attributes is not found, the entry with the OTHERWISE clause is selected if present. In the previous example, `Mixed` is selected as the replacement.

In a similar manner, an entry can be selected based on the dimensionality of the arguments.

```
dcl  D generic (D1 when ((*))),
                D2 when((*,*))),
     A(2),
     B(3,5);
call D(A);       /* D1 selected because A has one dimension  */
call D(B);       /* D2 selected because B has two dimensions */
```

If all of the descriptors are omitted or consist of an asterisk, the first entry reference with the correct number of descriptors is selected.

An entry expression used as an argument in a reference to a generic value matches only a descriptor of type ENTRY. If there is no such description, the program is in error.

Entry invocation or entry value

There are times when it might not be apparent whether an entry value itself will be used or the value returned by the entry invocation will be used. The following table and example help you understand which happens when.

If the entry reference . . .	It is . . .
Is a built-in function	Invoked
Has an argument list, even if null	Invoked
Is referenced in a CALL statement	Invoked
Has no argument list and is not referenced in a CALL statement	Not Invoked

In the following example, `A` is invoked, `B(C)` passes `C` as an entry value, and `D( C() )` invokes `C`.

```
dcl ( A, B, C returns (fixed bin), D) entry;

call A;                   /* A is invoked                      */
call B(C);                /* C is passed as an entry value     */
call D( C() );            /* C is invoked                      */
```

In the following example, the first assignment is invalid because it represents an attempt to assign an entry constant to an integer. The second assignment is valid.

```
dcl A fixed bin,
    B entry returns ( fixed bin );

A = B;
A = B();
```

CALL statement

The CALL statement invokes a subroutine.

```
►►──CALL──┬─entry-reference─┬──┬─(─┬──────────────┬─)─┬─;──────────────►◄
          ├─generic-name────┤  │   │ ┌──────,─────┐│  │
          └─built-in-name───┘  │   ▼ │            ││
                               │     ├─argument──┘│
                               └─────┴─*──────────┘
```
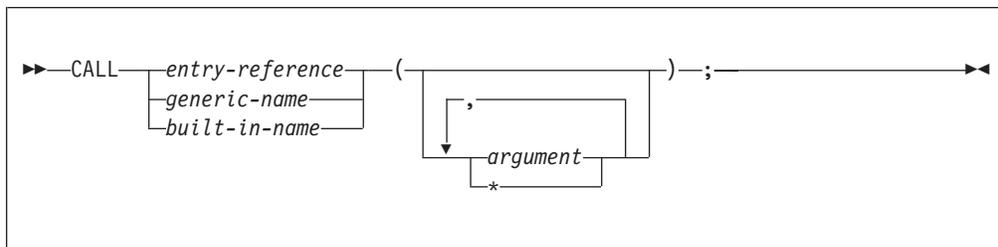
entry-reference
> Specifies that the name of the subroutine to be invoked is declared with the ENTRY attribute (discussed in "Entry data" on page 111).

generic-name
> Specifies that the name of the subroutine to be invoked is declared with the GENERIC attribute (discussed in "Generic entries" on page 121).

built-in name
> Specifies the name of the subroutine to be invoked is declared with the BUILTIN attribute (see "BUILTIN attribute" on page 371).

argument
> Is an element, an element expression, or an aggregate to be passed to the invoked subroutine. See "Passing arguments to procedures" on page 108.

References and expressions in the CALL statement are evaluated in the block in which the call is executed. This includes execution of any ON-units entered as the result of the evaluations.

RETURN statement

The RETURN statement terminates execution of the subroutine or function procedure that contains the RETURN statement and returns control to the invoking procedure. Control is returned to the point immediately following the invocation reference.

The RETURN statement with an expression should not be used within a procedure with OPTIONS(MAIN).

Return from a subroutine

To return from a subroutine, the RETURN statement syntax is:

```
►►──RETURN──;──────────────────────────────────────────────────►◄
```

If the RETURN statement terminates the main procedure, the FINISH condition is raised prior to program termination.

Return from a function

To return from a function, the RETURN statement syntax is:

```
►►──RETURN──(expression)──;─────────────────────────────────────►◄
```

The value returned to the function reference is the value of the expression specified, converted to conform to the attributes specified in the RETURNS option of the ENTRY or PROCEDURE statement at which the function was entered. For example:

```
F: procedure returns(fixed bin(15));
   .
   .
   .
G: entry returns(fixed dec(7,2));
   .
   .
   .
dcl D fixed bin(31);
   .
   .
   .
return (D);
```

If this function was entered at F, then D is converted to the attributes specified in the RETURNS option for the procedure F (FIXED BIN(15)). But, if this function was entered at G, then D is converted to the attributes specified in the RETURNS option for the entry G (FIXED DEC(7,2)).

You cannot specify an expression for the RETURN statement in a begin-block.

OPTIONS option and attribute

The OPTIONS option can be specified on PACKAGE, PROCEDURE, ENTRY, and BEGIN statements. The OPTIONS attribute can be specified on ENTRY declarations. It is used to specify processing characteristics that apply to the block and the invocation of a procedure. The options shown in the following syntax diagrams are listed alphabetically and discussed starting in 128.

BEGIN statement

```
►►─OPTIONS──(─┬─────────────────────┬─)──────────────────────►◄
              │  ┌─ORDER─────────┐   │
              ├──┤               ├───┤
              │  └─REORDER───────┘   │
              │  ┌─NOCHARGRAPHIC─┐   │
              ├──┤               ├───┤
              │  └─CHARGRAPHIC───┘   │
              │  ┌─NOINLINE──────┐   │
              └──┤               ├───┘
                 └─INLINE────────┘
```

ENTRY declaration

```
►►─OPTIONS──(─┬──────────────────────────┬─)──────────────────►◄
              ├─ASSEMBLER─┬──────────┬────┤
              │           └─RETCODE──┘    │
              ├─COBOL─────────────────────┤
              ├─FORTRAN───────────────────┤
              ├─FETCHABLE─────────────────┤
              │  ┌─BYADDR──┐              │
              ├──┤         ├──────────────┤
              │  └─BYVALUE─┘              │
              ├─┬─DESCRIPTOR───┬──────────┤
              │ └─NODESCRIPTOR─┘          │
              ├─LINKAGE──(─linkage─)──────┤
              │  ┌─IRREDUCIBLE─┐          │
              ├──┤             ├──────────┤
              │  └─REDUCIBLE───┘          │
              ├─NOMAP────┬───────────────┬┤
              │          └─parameter-list┘│
              ├─NOMAPIN──┬───────────────┬┤
              │          └─parameter-list┘│
              └─NOMAPOUT─┬───────────────┬┘
                         └─parameter-list┘
```

ENTRY statement

```
>>--OPTIONS--(--+------------------------------+--REENTRANT----)----><
                |  +-ASSEMBLER--+----------+-+                       
                |  |            +-RETCODE-+ |                        
                |  +-COBOL------------------+                        
                |  +-FORTRAN----------------+                        
                |                                                    
                |  +-BYADDR-+                                        
                |  |        |                                        
                |  +----------------+                               
                |          +-BYVALUE-+                               
                |                                                    
                |  +-DESCRIPTOR----+                                
                |  +-NODESCRIPTOR--+                                
                |  +-LINKAGE(linkage)--------+                      
                |                                                    
                |  +-NOMAP-----+--------------+                     
                |  |           +-parameter-list-+                   
                |  +-NOMAPIN---+--------------+                     
                |  |           +-parameter-list-+                   
                |  +-NOMAPOUT--+--------------+                     
                |              +-parameter-list-+                   
                |                                                    
                |  +-IRREDUCIBLE-+                                  
                |  |             |                                  
                |  +----------------+                              
                |         +-REDUCIBLE-+                            
```

PACKAGE statement

```
>>--OPTIONS--(--+--+-NOCHARGRAPHIC--+--+--)-----------------------><
                |  |                |  |                            
                |  +------------------+                            
                |       +-CHARGRAPHIC-+                            
                |                                                  
                |  +-ORDER-+                                      
                |  |       |                                      
                |  +---------+                                    
                |       +-REORDER-+                               
                |                                                 
                |       +-REENTRANT-+                             
```

OPTIONS option and attribute

```
PROCEDURE statement

►►──OPTIONS──(─┬───────────────────┬─────────────)──────────►◄
              │  ┌─ASSEMBLER─┐     │
              ├──┼─COBOL─────┼──┤
              │  └─FORTRAN───┘
              ├──MAIN──┬──────────────┬──
              │        └─NOEXECOPS─┘
              │  ┌─BYADDR─┐
              ├──┤        ├──
              │  └─BYVALUE─┘
              │  ┌─NOCHARGRAPHIC─┐
              ├──┤              ├──
              │  └─CHARGRAPHIC──┘
              │  ┌─DESCRIPTOR───┐
              ├──┼─NODESCRIPTOR─┤
              ├──FROMALIEN──
              ├──LINKAGE(linkage)──
              │  ┌─NOMAP────┬──────────────┐
              ├──┤          └─parameter-list─┘
              │  ├─NOMAPIN──┬──────────────┐
              │  │          └─parameter-list─┘
              │  └─NOMAPOUT─┬──────────────┐
              │             └─parameter-list─┘
              │  ┌─NOINLINE─┐
              ├──┤          ├──
              │  └─INLINE───┘
              │  ┌─ORDER────┐
              ├──┤          ├──
              │  └─REORDER──┘
              │  ┌─IRREDUCIBLE─┐
              ├──┤             ├──
              │  └─REDUCIBLE──┘
              ├──REENTRANT──
              ├──RETCODE──
              └──WINMAIN──
```

The options are separated by blanks or commas. They can appear in any order.

ASSEMBLER

  Abbreviation: ASM

  The ASSEMBLER option has the same effect as NODESCRIPTOR.

  If a procedure has the ASSEMBLER option, then upon exit from that procedure, the PLIRETV() value will be used as the return value for the procedure.

  For more information, refer to the Programming Guide.

BYADDR or BYVALUE

These specify how arguments and parameters are passed and received. BYADDR is the default.

BYVALUE can be specified only for scalar arguments and parameters that have known lengths and sizes.

The BYVALUE and BYADDR attributes can also be specified in the description list of an entry declaration and in the attribute list of a parameter declaration. Specifying BYVALUE or BYADDR in an entry or a parameter declaration overrides the option specified in an OPTIONS statement.

The following examples show BYVALUE and BYADDR in both entry declarations and in the OPTIONS statement. The examples assume that the compiler option DEFAULT(BYADDR) is in effect.

Example 1

```
 MAINPR: proc options(main);


    dcl D entry (fixed bin byaddr,
                 ptr,
                 char(4) byvalue)          /*  byvalue not needed      */
          options(byvalue);
    dcl E2 entry;                          /* default(byaddr) in effect */
    dcl Length fixed bin,
        P      pointer,
        Name   char(4);


    call D(Length, P, Name);               /* Length is passed byaddr */
                                           /* P is passed by value */
                                           /* Name is passed by value */


    call E2(P);                            /* P is passed by address */


 D: proc(I, Q, C)
    options(byvalue);
    dcl I fixed bin byaddr,
        Q ptr,
        C char(4) byvalue;


 E2: proc(Q);
     dcl Q ptr;
```

Example 2

```
 dcl F entry (fixed bin byaddr,         /*  byaddr not needed       */
              ptr,
              char(4) byvalue)
       options(byaddr);
 dcl E3 entry;
 dcl E4 entry (fixed bin byvalue);


 call F(Length, P, Name);              /* Length is passed byaddr */
                                       /* P is passed byaddr      */
                                       /* Name is passed by value */


 call E3(Name);                        /* Name is passed byaddr   */
 call E4(Length);                      /* Length is passed by value */
```

```
F: proc(I,P,C) options(byaddr);
  dcl I fixed bin byaddr;              /*  byaddr not needed        */
  dcl P ptr byaddr;                    /*  byaddr not needed        */
  dcl C char(4) byvalue;               /*  byvalue needed           */


E3: proc(L);
   dcl L char(4);


E4: proc(N);
   dcl N fixed bin byvalue;
```

CHARGRAPHIC or NOCHARGRAPHIC
Abbreviations: CHARG, NOCHARG

The default for an external procedure is NOCHARG. Internal procedures and begin-blocks inherit their defaults from the containing procedure.

When CHARG is in effect, the following semantic changes occur:

- All character string assignments are considered to be mixed character assignments.
- STRINGSIZE condition causes MPSTR built-in function to be used. STRINGSIZE must be enabled for character assignment that can cause truncation and intelligent DBCS truncation is required. (For information on the MPSTR BUILTIN see "MPSTR" on page 440.) For example:

```
  Name: procedure options(chargraphic);
        dcl A char(5);
        dcl B char(8);

  /* the following statement...                           */

        (stringsize): A=B;

  /*...is logically transformed into...                   */

        A=mpstr(B,'vs',length(A));
```

When NOCHARG is in effect, no semantic changes occur.

COBOL

This option has the same effects as NODESCRIPTOR (see below), but additionally OPTIONS(COBOL)

- implies LINKAGE(SYSTEM) unless a different linkage is specified on the entry declaration or procedure statement.
- permits the use of the NOMAP, NOMAPIN and NOMAPOUT options
- implies, if specified on a procedure statement, that upon exit from that procedure, the PLIRETV() value will be used as the return value for the procedure.

DESCRIPTOR or NODESCRIPTOR
These indicate whether the procedure specified in the entry declaration or procedure statement will be passed a descriptor list when it is invoked.

If DESCRIPTOR appears, the compiler passes descriptors, if necessary.

If NODESCRIPTOR appears, the compiler does not pass descriptors.

If neither appears, DESCRIPTOR is assumed only when one of the invoked procedure's parameters is a string, array, area, structure, or union.

It is an error for NODESCRIPTOR to appear on a procedure statement or entry declaration in which any of the parameters or elements:

- Use the asterisk (*) to indicate the extents, length, or size
- If any parameter is NONCONNECTED

However, NODESCRIPTOR is allowed if the parameters with unspecified extents are NONASSIGNABLE VARYING or VARYINGZ strings.

FETCHABLE

This option indicates the procedure is dynamically fetched if necessary before invoking it.

FORTRAN

This option causes no descriptors to be passed except for character variables.

FROMALIEN

This option indicates that this procedure can be called from a non-PL/I routine. FROMALIEN can be specified on any procedure; however, this would incur unnecessary overhead.

INLINE or NOINLINE

INLINE and NOINLINE are optimization options that can be specified for begin-blocks and non-nested level-one procedures in a package.

INLINE indicates that whenever the begin-block or procedure is invoked in the package that defines it, the code for the begin-block or procedure should be executed *inline* at the point of its invocation. Even if INLINE is specified, the compiler can choose not to inline the begin-block or procedure.

NOINLINE indicates that the begin-block or procedure is never to be executed inline.

OPTIONS(INLINE) makes it easier to write well-structured, readable code. For instance, a program could be written as a series of calls to a set of procedures, and OPTIONS(INLINE) could be used to eliminate the overhead of actually calling these procedures one by one.

If a procedure or begin-block is executed inline, the values returned by built-in functions like ONLOC return the name of the procedure into which it is inlined. Similarly, traceback information does not include the called procedure.

Some procedures and begin-blocks are never inlined. These include, but are not limited to:

- Procedures and begin-blocks in packages in which condition enablement varies
- Procedures and begin-blocks containing ON or REVERT statements
- Procedures and begin-blocks containing data-directed input/output statements
- Procedures and begin-blocks containing assignments or comparisons of ENTRY, FORMAT, or LABEL constants

If a non-nested procedure with the INLINE option is not external and not referenced, no code will be generated for it. If neither INLINE nor NOINLINE is specified for a procedure, the option is set by the DEFAULT compiler option.

For more information about using INLINE and NOINLINE, refer to the Programming Guide.

LINKAGE

This option specifies the calling convention used and may be specifid on PROCEDURE statements and ENTRY declarations.

CDECL (INTEL only)

This option specifies the CDECL linkage convention used by 32-bit C compilers.

OPTLINK

This option is the default, and is the fastest linkage convention. It is not standard linkage for most compilers.

STDCALL (Windows Only)

This option specifies the STDCALL linkage which is the standard linkage convention used by all Windows APIs.

SYSTEM

This option is the calling convention which should be used for calls to the operating system. Although this option is slower than OPTLINK, it is standard for all OS/2, MVS, and AIX applications and is used for calling OS/2 application programming interfaces.

For more information about calling conventions, refer to the Programming Guide.

MAIN

This option indicates that this external procedure is the initial procedure of a PL/I program. MAIN is valid, and required, only on one external procedure per program. The operating system control program invokes it as the first step in the execution of that program.

A PL/I program that contains more than one procedure with OPTIONS(MAIN) can produce unpredictable results.

NOEXECOPS

The NOEXECOPS option is valid only with the MAIN option. It specifies that the run-time options will not be specified on the command or statement that invokes the program. Only parameters for the main procedure will be specified.

NOMAP, NOMAPIN, NOMAPOUT

These options prevent the automatic manipulation of data aggregates at the interface between either COBOL or FORTRAN and PL/I.

Each option argument-list may specify the parameters to which the option applies. Parameters may appear in any order and are separated by commas or blanks. If there is no argument-list for an option, the default list is all the parameters of the entry name.

NOMAP, NOMAPIN and NOMAPOUT may all appear in the same OPTIONS specification. This specification should not include the same parameter in more than one specified or default argument list.

These options are accepted but ignored unless the COBOL option applies.

ORDER or REORDER

ORDER and REORDER are optimization options that are specified for a procedure or begin-block.

ORDER indicates that only the most recently assigned values of variables modified in the block are available for ON-units that are entered because of computational conditions raised during statement execution and expressions in the block.

The REORDER option allows the compiler to generate optimized code to produce the result specified by the source program when error-free execution takes place.

For more information on using the ORDER and REORDER options, refer to the Programming Guide.

If neither option is specified for the external procedure, the default is set by the DEFAULT compiler option. Internal blocks inherit ORDER or REORDER from the containing block.

REDUCIBLE or IRREDUCIBLE
  Abbreviations: RED, IRRED

REDUCIBLE indicates that a procedure or entry need not be invoked multiple times if the argument(s) stays unchanged, and that the invocation of the procedure has no side effects.

For example, a user-written function that computes a result based on unchanging data should be declared REDUCIBLE. A function that computes a result based on changing data, such as a random number or time of day, should be declared IRREDUCIBLE.

REENTRANT

This option is ignored. On the Intel and AIX platforms, all PL/I programs are reentrant. On the z/OS platform, all programs compiled with the RENT compiler option are reentrant, and other programs are reentrant if they do not alter any static variables (which may require use of the NOWRITABLE compiler option).

RETCODE

This option specifies that if the ENTRY point also has the ASM or COBOL option, then the ENTRY will return a value that will be saved, after the ENTRY is invoked, as the PL/I return code. Essentially, after such an ENTRY is invoked, its return value will be passed to the PLIRETC subroutine.

WINMAIN (Windows only)

WIN ⇒

This option automatically implies LINKAGE(STDCALL) and EXT('WinMain'). The associated routine should contain four parameters:
1. An instance handle
2. A previous handle
3. A pointer to the command line
4. An integer to be passed to ShowWindow.

These are the same four parameters expected by the C WinMain and the calls made from this routine are the same as those expected from a C routine.

◁⊟

RETURNS option and attribute

If a procedure is a function procedure, you must specify the RETURNS option on the procedure statement. Further, in the invoking procedure or package, you must declare such a procedure as an entry with the RETURNS attribute. The RETURNS option and the RETURNS attribute are used to specify the attributes of the value that is being returned. The attributes in the RETURNS option must match the attributes in the RETURNS attribute.

Procedures that are subroutines (and are therefore invoked using the CALL statement) must not have the RETURNS option on the procedure statement and their entry declaration must not have the RETURNS attribute.

```
►►──RETURNS──(──┬──────────┬──)────────────────────►◄
                ▲          │
                └─attribute─┘
```
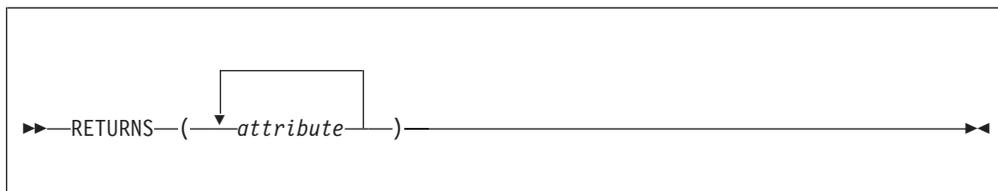
If more than one attribute is specified, they must be separated by blanks (except attributes such as precision that are enclosed in parentheses).

The attributes are specified in the same way as they are in a declare statement. Defaults are applied in the normal way.

The attributes that can be specified are any of the data attributes and alignment attributes for scalar variables (as shown in Table 8 on page 27). ENTRY variables must have the LIMITED attribute. In addition, you can specify the TYPE attribute to name user-defined types, ordinals, and typed structures and unions.

String lengths and area sizes must be specified by constants. The returned value has the specified length or size.

BYADDR and BYVALUE can also be specified in the list of attributes for RETURNS. The BYADDR attribute must be in effect if a procedure contains any ENTRY statements and the procedure or any of its secondary entry points returns:
- no value, or
- an aggregate value

On z/OS, BYADDR is the default for RETURNS. If a C function is called, BYVALUE must be specified in the list of attributes for RETURNS. For a discussion of these attributes, see "Using BYVALUE and BYADDR" on page 108.

Chapter 7. Type definitions

In a programming language, a *type* is a description of a set of values and a set of allowed operations on those values. PL/I has many built-in data types. Each type can specify a number of elementary attributes. Chapter 3, "Data elements," on page 21 describes these built-in data types.

PL/I allows you to define your own types using the built-in data types. This chapter discusses user-defined types (aliases, ordinals, structures, and unions), declarations of variables with these types, handles, and type functions.

User-defined types (aliases)

An *alias* is a type name that can be used wherever an explicit data type is allowed. Using the DEFINE ALIAS statement, you can define an alias for a collection of data attributes. In this way, you can assign meaningful names to data types and improve the understandability of a program. By defining an alias, you can also provide a shorter notation for a set of data attributes, which can decrease typographical errors.

DEFINE ALIAS statement

The DEFINE ALIAS statement specifies a name that can be used as a synonym for the set of data type attributes you give to the alias.

```
>>--DEFINE--ALIAS--alias-name--+--attribute--+--;--------------><
                               |      <----,----|
```
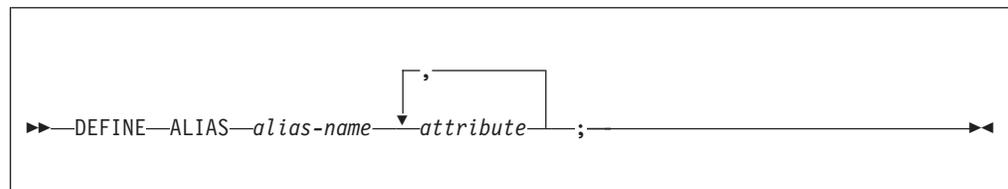
alias-name
> Specifies the name that can be used wherever the explicit data type defined by the specified attributes is allowed

attributes

The attributes that can be specified are any of the attributes for variables that can be returned by a function (for example, those attributes valid in the RETURNS option and attribute). These valid attributes are listed in Table 8 on page 27. Therefore, you cannot specify an alias for an array or a structured attribute list. However, you can specify an alias for a type that is defined in a DEFINE ORDINAL, or DEFINE STRUCTURE statement (see "DEFINE ORDINAL statement" and "Defining typed structures and unions" on page 138), or in another DEFINE ALIAS statement. Also, as in the RETURNS option and attribute, any string lengths or area sizes must be restricted expressions.

Missing data attributes are supplied using PL/I defaults.

Example

```
define alias Name   char(31) varying;
define alias Salary fixed dec(7);     /* real by default       */
define alias Zip    char(5)           /* nonvarying by default */
```

Whenever Name is used in a DECLARE statement, it has the attributes char(31) varying.

Defining ordinals

An ordinal is a named set of ordered values. Using the DEFINE ORDINAL statement, you can define an ordinal and assign meaningful names to be used to refer to that set. For example, you can define an ordinal called "color". The "color" ordinal could include the members "red", "yellow", "blue", etc. The members of the "color" set can then be referred to by these names instead of by their associated fixed binary value, making code much more self-documenting. Furthermore, a variable declared with the ordinal type can be assigned and compared only with an ordinal of the same type, or with a member of that ordinal type. This automatic checking provides for better program reliability.

DEFINE ORDINAL statement

The DEFINE ORDINAL statement specifies a named type representing a set of named ordered values.
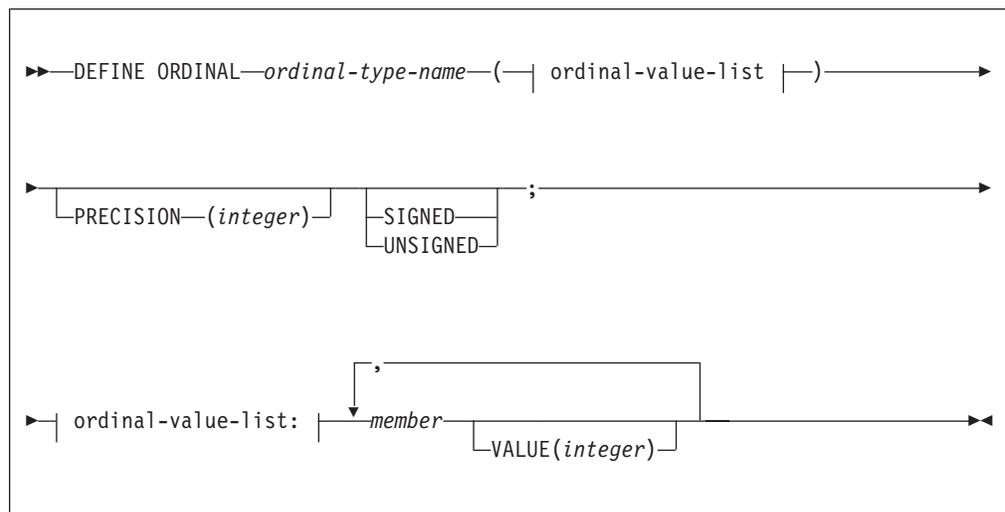
ordinal-type-name

Ordinal-type-name specifies the name of the set of ordinal values. This name

can be used only in DECLARE statements with the ORDINAL attribute. Use of this name elsewhere results in it being treated as any other nonordinal name.

member

Specifies the name of a member within the set.

VALUE

The VALUE attribute specifies the value of a particular member within the set. If the VALUE attribute is omitted for the first member, a value of zero is used. If the VALUE attribute is omitted for any other member, the next greater integer value is used.

The value in the given (or assumed) VALUE attribute must be an integer, can be signed, and must be strictly increasing. The value in the given (or assumed) VALUE attributed may also be specified as an XN constant.

PRECISION

Abbreviation: PREC

The PRECISION attribute specifies the precision of a particular ordinal value. If omitted, the precision is determined by the range of ordinal values.

The maximum precision is the same as that for data items declared FIXED BINARY.

SIGNED or UNSIGNED

These attributes indicate whether ordinal values can assume negative values. If omitted, they are determined by the range of ordinal values. For example, if any value is negative, the SIGNED attribute is applied.

For more information on SIGNED and UNSIGNED, refer to "SIGNED and UNSIGNED attributes" on page 30.

In the following example, `Red` has the value 0, `Orange` has the value 1, etc. But `Low` has the value 2 and `Medium` has the value 3.

Example

```
define ordinal Color ( Red,           /* is 0, since VALUE is omitted */
                       Orange,
                       Yellow,
                       Green,
                       Blue,
                       Indigo,
                       Violet );

define ordinal Intensity ( Low    value(2),
                           Medium,
                           High   value(5));
```

Defining typed structures and unions

The DEFINE STRUCTURE statement specifies a named structure or union type.
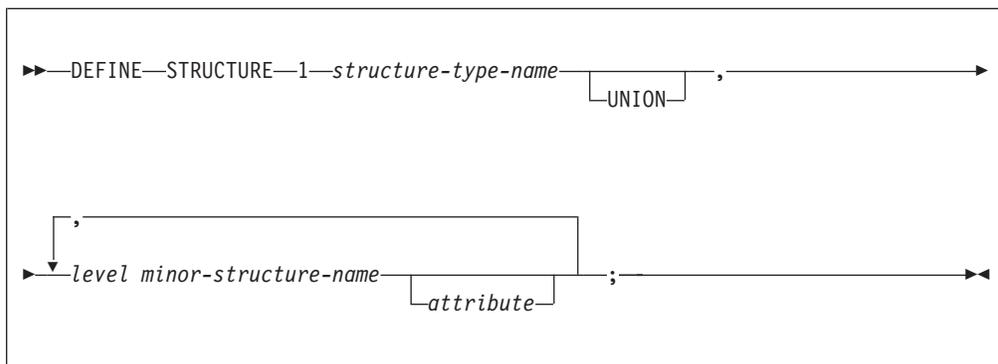
structure-type-name
  Specifies the name given to this structure type (see "Structures" on page 170 for more information on major structures). This name cannot have dimensions, although substructures can.

UNION
  Is discussed in "UNION attribute" on page 172.

minor-structure-name
  Specifies the name given to a deeper level. (see "Structures" on page 170 for more information on minor structures).

attributes
  Specifies attributes for the minor-structure name. Only data attributes are allowed.

Any string lengths, area sizes, or array dimensions specified in a DEFINE STRUCTURE statement must be restricted expressions.

Missing data attributes are supplied using PL/I defaults.

The DEFINE STRUCTURE statement defines a "strong" type. In other words, variables declared with that type can only be assigned to variables (or parameters) having the same type. Typed structures can not be used in data-directed input/output statements.

A DEFINE STRUCTURE statement that merely names the structure to be defined without specifying any of its members defines an ″unspecified structure″.

- An unspecified structure cannot be dereferenced, but it may be used to declare a HANDLE which, of course, cannot be dereferenced either.
- An unspecified structure may also be the subject of a later DEFINE STRUCTURE statement which does specifies its members.

Unspecified structure definitions are useful when a structure definition contains is a handle to a second structure which also contains is a handle to the first structure. For instance, in the following example, the parent structure contains a handle to the child structure, but the child structure also contains a handle to the parent structure.

```
define structure 1 child;

define structure
 1 parent,
```

```
      2 first_child    handle child,
      2 parent_data    fixed bin(31);

   define structure
    1 child,
      2 parent         handle parent,
      2 next_child     handle child,
      2 child_data     fixed bin(31);
```

HANDLE attribute

You can use the HANDLE attribute to declare a variable as a pointer to a structure type. Such a variable is called a *handle*.

```
►►──HANDLE──┬──structure-type-name────┬──────────────────────►◄
            └─(structure-type-name)───┘
```
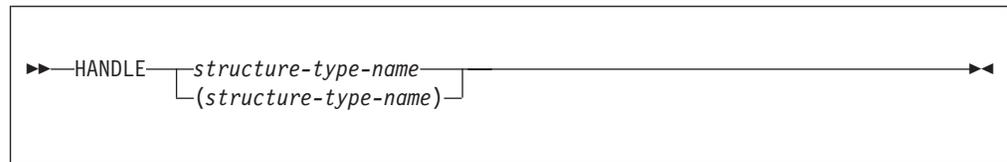
structure-type-name
Specifies the typed structure this handle points to.

Like defined structures, handles are strongly typed: they can only be assigned to or compared with handles for the same structure type. No arithmetic operations are permitted on handles.

You cannot use the ADDR built-in function to assign the address of a typed structure to a handle because the ADDR built-in function returns a pointer, and pointers cannot be assigned to handles. However, the HANDLE built-in function takes a typed structure as its argument and returns a handle to that type. In the following example, using the `tm` structure type defined on page 140, a handle is declared which locates the `tm` type and the address of `Daterec` is assigned to that handle.

```
dcl P_Daterec handle tm;
dcl Daterec type tm;

P_Daterec = handle(Daterec);
```

You can convert a handle to a pointer using the POINTERVALUE built-in function.

Declaring typed variables

By using the TYPE attribute, a variable can be declared with the type specified in a DEFINE ALIAS, DEFINE STRUCTURE or DEFINE ORDINAL statement.

TYPE attribute

```
►►──TYPE──┬──defined-type-name────┬──────────────────────────►◄
          └─(defined-type-name)───┘
```
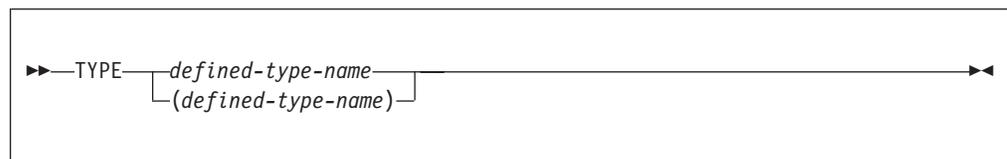
defined-type-name
Specifies the name of a previously defined alias, defined structure, or ordinal type.

Examples

```
define alias Name char(31) varying;
              /* Name has attributes char(31) varying */
dcl Employee_Name type Name;
              /* Employee_Name type char(31) varying */
define alias Rate fixed dec(3,2);
              /* Rate has attributes fixed dec real */

define structure
1 Payroll,
2 Name,
3 Last   type Name,
3 First type Name,
2 Hours,
3 Regular  fixed dec(5,2),
3 Overtime fixed dec(5,2),
2 Rate,
3 Regular  type Rate,
3 Overtime type Rate;

dcl Non_Exempt type Payroll;    /* Has Payroll structure type */
dcl Exempt     type Payroll;    /* Has Payroll structure type */
```

The TYPE attribute can be used in a DEFINE ALIAS statement to specify an alias for a type defined in a previous DEFINE ALIAS statement. For example:

```
  define alias Word fixed bin(31);
  define alias Short type word;
```

The following example defines several named types, a structure type (tm), and declares the C function that gets a handle to this typed structure:

```
  define alias int     fixed bin(31);
  define alias time_t  fixed bin(31);
  define structure
    1 tm
     ,2 tm_sec   type int   /* seconds after the minute (0-61)   */
     ,2 tm_min   type int   /* minutes after the hour (0-59)     */
     ,2 tm_hour  type int   /* hours since midnight (0-23)        */
     ,2 tm_mday  type int   /* day of the month (1-31)            */
     ,2 tm_mon   type int   /* months since January (0-11)        */
     ,2 tm_year  type int   /* years since 1900                   */
     ,2 tm_wday  type int   /* days since Sunday (0-6)            */
     ,2 tm_yday  type int   /* days since January 1 (0-365)       */
     ,2 tm_isdst type int   /* Daylight Saving Time flag          */
     ;

  dcl localtime    ext('localtime')
          entry( nonasgn byaddr type time_t )
          returns( byvalue handle tm );

  dcl time     ext('time')
          entry( byvalue pointer )
          returns( byvalue type time_t );
```

ORDINAL attribute

By using the TYPE or ORDINAL attribute, variables can be declared with an ordinal type. See "TYPE attribute" on page 139 for the syntax for the TYPE attribute.

```
►►──ORDINAL──ordinal-type-name─────────────────────────────────────►◄
```

ordinal-type-name
> Specifies the name of a previously defined set of ordinal values.

For example:

```
dcl Wall_color ordinal Color;
```

The ORDINAL attribute conflicts with other data attributes such as FIXED or SIGNED, but it is allowed with attributes such as BASED or DIMENSION.

Typed structure qualification

You reference a member of a typed structure using the . operator or a handle with the => operator. Unlike names in a typical untyped structure, the names in a typed structure form their own "name space" and cannot be referenced by themselves. For example, given the following declares and definitions

```
dcl 1 A,
2 B fixed bin,
2 C fixed bin;

define structure
1 X,
2 Y fixed bin,
2 Z fixed bin;
dcl S type X;
```

B is a valid reference, but Y is not.

Type names are also in a separate name space from declared names. Therefore, you can use the name of a type as a variable name also.

```
define alias Hps pointer;
declare Hps type Hps;
```

Using the ″.″ operator

The syntax for referring to a typed structure member using the ″.″ operator is:

```
►►──typed-structure-name──.──typed-structure-member──────────────────►◄
```

typed-structure-reference
> Name of the declared typed structure

typed-structure-member
> Name of the referenced major or minor structure member of the structure type

For example, given the structure type `tm` and function `localtime` defined as in the example on page 140, the following code obtains the system date and displays the time:

```
dcl Daterec type tm;

dcl ltime   type time_t;
dcl ptime   handle tm;
```

```
        ltime = time( null() );
        ptime = localtime( ltime );

        Daterec = ptime => tm;

display ( edit(Daterec.Hours,'99') || ':' ||
edit(Daterec.Minutes,'99') || ':' ||
edit(Daterec.Seconds,'99'));
```

Combinations of arrays and typed structures or unions

As described in "Combinations of arrays, structures, and unions" on page 176, given this untyped structure:

```
dcl 1 a(3),
      2 b(4) fixed bin,
      2 c(5) fixed bin;
```

a(1).b(2), a.b(1,2), and a(1,2).b have the same meaning.

However, given the following typed structure:

```
define structure
  1 t,
    2 b(4) fixed bin,
    2 c(5) fixed bin;

dcl x(3) type t;
```

only x(1).b(2) is valid. In addition, the assignment statement x.b = 0 is invalid, but x(1).b = 0; is valid.

Given the structure type t defined previously and the following function f:

```
dcl f entry returns( type t );
```

display(f().b(2)) is valid.

Using handles

Handles access members of a typed structure with the => operator. In the following example, given the tm type defined on page 140, the time is displayed using a handle to that type:

```
dcl P_Daterec handle tm;
P_Daterec = handle(Daterec);

display ( edit(P_Daterec=>tm_hours,'99') || ':' ||
edit(P_Daterec=>tm_min,'99') || ':' ||
edit(P_Daterec=>tm_sec,'99') );
```

Handles can locate any member in a typed structure, including the level-1 name (the type name itself). A reference by a handle to its type name constitutes a reference to the typed structure which is pointed to by that handle. This allows reference to this aggregate data by its handle. For example, given that H1 and H2 point to two allocated structures, you can swap two structures by:

```
define structure 1 T, 2 U, 2 V, 2 W;
dcl (H1, H2) handle T;
dcl Temp     type T;

Temp = H1=>T;
H1=>T = H2=>T;
H2=>T = Temp;
```

Using ordinals

When using ordinals, keep in mind the following:

- Ordinals are *strongly-typed*; that is, an ordinal can only be compared with or assigned to another ordinal of the same type. The ordinal must have been explicitly declared in a DECLARE statement.
- The ordinal-type-name in a DEFINE ORDINAL statement cannot used in comparisons or assignments.
- Ordinals can be passed/received as arguments/parameters like any other data type.
- Ordinals are invalid as arguments for all built-in functions requiring arguments with computational types. However, in support of ordinals, built-in functions have been defined and BINARYVALUE has been extended. These built-in functions are listed in Table 23, and their descriptions can be found in Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365. Each of the built-in functions listed takes exactly one argument, which must be a reference having type ORDINAL.

Table 23. Ordinal-handling built-in functions

Function	Description
BINARYVALUE	Converts an ordinal to a binary value
ORDINALPRED	Returns the next lower value for an ordinal
ORDINALSUCC	Returns the next higher value for an ordinal
ORDINALNAME	Returns a character string giving an ordinal's name

For example, in the following sample code, the first DO loop below would list, in ascending order, the members of the *Color* set; the second DO loop would list them in descending order. The example uses the ordinal definition from "Example" on page 137.

Example

```
dcl Next_color ordinal Color;

do Next_color = first (:Color:)
             repeat ordinalsucc( Next_color )
             until (Next_color = last (:Color:));

   display( ordinalname( Next_color ) );
end;

do Next_color = last (:Color:)
             repeat ordinalpred( Next_color)
             until (Next_color = first(:Color:);

   display( ordinalname( Next_color));
end;
```

The sample output for the first loop would be:

```
RED
ORANGE
YELLOW
GREEN
BLUE
INDIGO
VIOLET
```

An ordinal cannot be used as an index into an array and cannot define an extent for a variable, including the lower or upper bound of an array. However, an ordinal can be converted to binary using the BINARYVALUE built-in function. The value which is returned by this function can then be used to index into an array or define an extent.

For example, the following package defines an array *usage_count* to hold the number of times each color is used, a procedure *Record_usage* to update this array, and a procedure *Show_usage* to display the values in this array.

Example

```
Usage: package exports(*);

  define ordinal Color ( Red,
                         Orange,
                         Yellow,
                         Green,
                         Blue,
                         Indigo,
                         Violet );

  dcl Usage_count(   binvalue( first(:Color:))
                  : binvalue( last(:Color:)) )
         static fixed bin(31) init( (*) 0 );
              /* first(:Color:)  = Red    */
              /* last(:Color:) = Violet */

Record_usage: proc (Wall_color );
  dcl Wall_color type Color parm byvalue;

  Usage_count( binvalue(Wall_color) )
      = 1 + Usage_count( binvalue(Wall_color) );
end Record_usage;

Show_usage: proc;
  dcl Next_color type Color;

  do Next_color = Red upthru Violet;
    put skip list( ordinalname( Next_color) );
    put list( Usage_count( binvalue(Next_color) ));
  end;
end Show_usage;

end Usage;
```

Ordinals can be used to create functions that are easy to maintain and enhance, but which are as efficient as table look-ups.

In the following example, the function *Is_mellow* returns a bit indicating whether a color is or is not "mellow". If more colors are defined, the "mellow" ones can be added to the list of colors in the select-group. In a select-group, unlike a hand-built table, the colors do not have to be in the same order as in the DEFINE statement, or in any particular order at all.

However, since all of the statements inside the select-group consist of RETURN statements that return constant values, the compiler will convert the entire select-group into a simple table look-up.

Example

```
Is_mellow: proc( Test_color ) returns( bit(1) aligned );

  dcl Test_color type Color parm byvalue;
```

```
    select (Test_color);
      when( Yellow, Indigo)
          return( '1'b );
      otherwise
          return( '0'b );
    end;

  end;
```

This feature can also be used to define your own version of the ORDINALNAME built-in function. Your own version can return the name you want to be displayed for each ordinal value. For example, the following function `Color_name` returns the color name associated with each name with the first letter capitalized:

```
  Color_name: proc( Test_color ) returns( char(8) varying );

    dcl Test_color type Color parm byvalue;

    select (Test_color);
      when ( Blue   ) return( 'Blue');
      when ( Green  ) return( 'Green');
      when ( Orange ) return( 'Orange');
      when ( Red    ) return( 'Red');
      when ( Yellow ) return( 'Yellow');
      otherwise return (");
    end;

  end;
```

Type functions

Since type names are in a separate name space from declared names, they cannot be used where variable references are required, in particular as arguments to built-in functions. However, type names can be used as arguments to *type functions.* (In ANSI terminology, these type functions are known as *enquiry functions.*) These type functions are listed in Table 24.

Table 24. Type functions

Function	Description
BIND	Converts a pointer to a handle for a type
CAST	Converts an expression to a specified type using C conversion rules
FIRST	Returns the first value in an ordinal set
LAST	Returns the last value in an ordinal set
NEW	Acquires storage for a structure type and returns a handle to the acquired storage
RESPEC	Changes the attributes of an expression to a specified type without changing the bit pattern of the expression
SIZE	Returns the amount of storage needed to represent a type

Descriptions for these type functions can be found in Chapter 20, "Type Functions," on page 495.

Chapter 8. Data declarations

When a PL/I program is executed, it can manipulate many different data items of particular data types. Each data item, except an unnamed arithmetic or string constant, is referred to in the program by a name. Each data name is given attributes and a meaning by a declaration (explicit or implicit).

Most attributes of data items are known at the time the program is compiled. For nonstatic items, attribute values (the bounds of the dimensions of arrays, the lengths of strings, area sizes, initial values) and some file attributes can be determined during execution of the program. Refer to "Block activation" on page 89 for more information.

Data items, types, and attributes are introduced in Chapter 3, "Data elements," on page 21.

This chapter discusses explicit and implicit declarations, scalar, array, structure, and union declarations, scope of names, data alignment, and default attributes.

Explicit declaration

A name is explicitly declared if it appears:
- In a DECLARE statement. The DECLARE statement explicitly declares attributes of names.

Explicit declaration

- As an entry constant. Labels of PROCEDURE and ENTRY statements constitute declarations of the entry constants within the containing procedure.
- As a label constant. A label constant explicitly declares a label.
- As a format constant. A label on a FORMAT statement constitutes an explicit declaration of the label.

The scope of an explicit declaration of a name is the block containing the declaration. This includes all contained blocks, except those blocks (and any blocks contained within them) to which another explicit declaration of the same name is internal. In the following diagram, the lines indicate the scope of the declaration of the names.

```
P  A  B  B'                 C  C' D  Q  R


         P: PROC;

            DCL A, B;

            Q: PROC;

               DCL B, C;

               R: PROC;

                  DCL C,D;

               END R;

            END Q;

         END P;
```
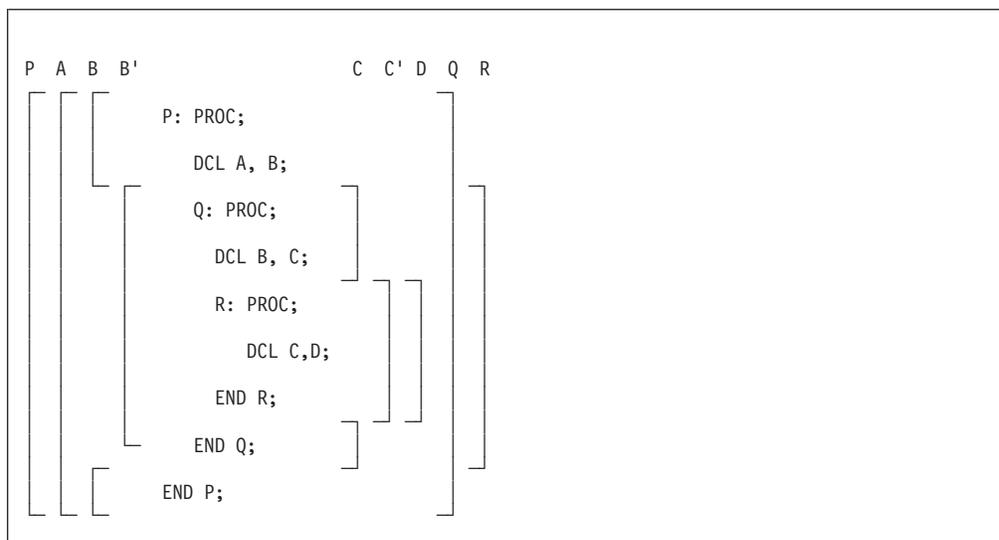
`B` and `B'` indicate the two distinct uses of the name `B`; `C` and `C'` indicate the two uses of the name `C`.

For more information about scope, refer to "Scope of declarations" on page 151.

DECLARE statement

The DECLARE statement specifies some or all of the attributes of a name. If the attributes are not explicitly declared and cannot be determined by context, default attributes are applied.

DECLARE statements can be an important part of the documentation of a program. Consequently, you can make liberal use of declarations, even when default attributes suffice or when an implicit declaration is possible. Because there are no restrictions on the number of DECLARE statements, you can use different DECLARE statements for different groups of names. Any number of names can be declared in one DECLARE statement.

```
                  ,
          ┌───────────────────┐
          │                   │          ┌────────────────────────────┐
   ▶▶─DECLARE─┬───────┬─┬──name──┬─│─┬────────────────┬─────────┬──;─────────────▶
             └─level─┘ └───*────┘   └─│─ attributes ─┘         │
```

```
   ┌─┐                     ┌─────────────────────────────────────┐
   ▶─┤ attributes: ├───│──┬──data-attributes──────────┬──────────│────────────▶◀
                          ├──alignment-attributes──────┤
                          ├──scope-attributes──────────┤
                          ├──storage-attributes────────┤
                          └──complementary-attributes──┘
```
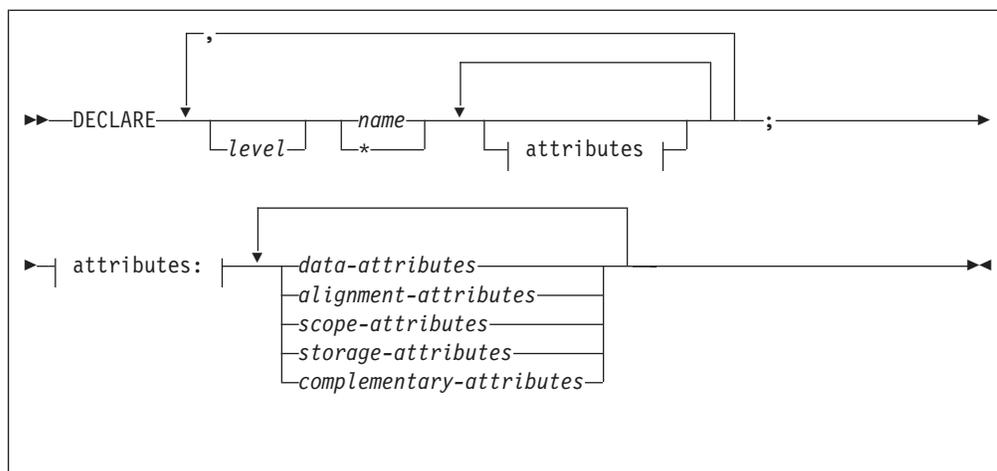
Abbreviation: DCL

For more information about declaring arrays, structures, and unions, refer to "Arrays" on page 167, "Structures" on page 170, or "Unions" on page 172.

* Cannot be used as the *name* of an INTERNAL or an EXTERNAL scalar or as the name of a level-1 EXTERNAL structure or union unless the EXTERNAL attribute specifies an environment name (see "INTERNAL and EXTERNAL attributes" on page 153).

attributes

The attributes can appear in any order.

All attributes given explicitly for the name must be declared together in a DECLARE statement, except that:

Names having the FILE attribute can also be given attributes in an OPEN statement (or have attributes implied by an implicit opening). For more information on the OPEN statement, see "OPEN statement" on page 265.

The parameter attribute is contextually applied by the appearance of the name in a parameter list. A DECLARE statement internal to the block can specify additional attributes.

Attributes of external names, in separate blocks and compilations, must be consistent.

For more information about attributes and the members of the given groups, refer to "Data types and attributes" on page 22.

level

A nonzero integer. If a level-number is not specified, *level 1* is the default for element and array variables. *Level 1* must be specified for major structure and union names.

name

Each level-1 name must be unique within a block. For more information on level-1 names, refer to "Structures" on page 170.

Condition prefixes and labels cannot be specified on a DECLARE statement.

Factoring attributes

Attributes common to several names can be factored to eliminate repeated specification of the same attributes. Factoring is achieved by enclosing the names in parentheses followed by the set of attributes which apply to all of the names. Factoring can be nested. The dimension attribute can be factored. Factoring can also be used on elementary names within structures and unions. A factored level-number must precede the parenthesized list.

Names within the parenthesized list are separated by commas. No factored attribute can be overridden for any of the names, but any name within the list can be given other attributes as long as there is no conflict with the factored attributes.

The following examples show factoring. The last declaration in the set of examples shows nested factoring.

```
declare (A,B,C,D) binary fixed (31);

declare (E decimal(6,5), F character(10)) static;

declare 1 A, 2(B,C,D) (3,2) binary fixed (15);

declare ((A,B) fixed(10),C float(5)) external;
```

Implicit declaration

If a name appears in a program and is not explicitly declared, it is implicitly declared. The scope of an implicit declaration is determined as if the name were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name is used.

With the exception of files, entries, and built-in functions, implicit declaration has the same effect as if the name were declared in the outermost procedure. For files and built-in functions, implicit declaration has the same effect as if the names were declared in the logical package outside any procedures.

Note: Using implicit declarations for anything other than built-in functions and the files SYSIN and SYSPRINT is in violation of the 1987 ANSI standard and should be avoided.

Some attributes for a name declared implicitly can be determined from the context in which the name appears. These cases, called *contextual declarations*, are:

- A name of a built-in function.
- A name that appears in a CALL statement or the CALL option of INITIAL, or that is followed by an argument list, is given the ENTRY and EXTERNAL attributes.
- A name that appears in the parameter list of a PROCEDURE or ENTRY statement is given the PARAMETER attribute.
- A name that appears in a FILE or COPY option, or a name that appears in an ON, SIGNAL, or REVERT statement for a condition that requires a file name, is given the FILE attribute.
- A name that appears in an ON CONDITION, SIGNAL CONDITION, or REVERT CONDITION statement is given the CONDITION attribute.
- A name that appears in the BASED attribute, in a SET option, or on the left-hand side of a locator qualification symbol is given the POINTER attribute.
- A name that appears in an IN option, or in the OFFSET attribute, is given the AREA attribute.

Examples of contextual declaration are:

```
read file (PREQ) into (Q);

allocate X in (S);
```

In these statements, `PREQ` is given the FILE attribute, and `S` is given the AREA attribute.

Implicit declarations that are not contextual declarations acquire all attributes by default, as described in "Defaults for attributes" on page 162. Because a contextual declaration cannot exist within the scope of an explicit declaration, it is impossible for the context of a name to add to the attributes established for that name in an explicit declaration.

Scope of declarations

The part of the program to which a name applies is called the *scope of the declaration* of that name. In most cases, the scope of the declaration of a name is determined entirely by the position where the name is declared within the program. Implicit declarations are treated as if the name were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure.

It is not necessary for a name to have the same meaning throughout a program. A name explicitly declared within a block has a meaning only within that block. Outside the block, the name is unknown unless the same name has also been declared in the outer block. Each declaration of the name establishes a scope and in this case, the name in the outer block refers to a different data item. This enables you to specify local definitions and, hence, to write procedures or begin-blocks without knowing all the names used in other parts of the program.

In the following example, the output for `A` is actually `C.A`, which is `2`. The output for `B` is `1`, as declared in procedure `X`.

```
X: proc options(main);
   dcl (A,B) char(1) init('1');
   call Y;
   return;

   Y: proc;
     dcl 1 C,
         3 A char(1) init('2');
     put data(A,B);
     return;
   end Y;
end X;
```

Thus, for nested procedures, PL/I uses the variable declared within the current block before using any variables that are declared in containing blocks.

In order to understand the scope of the declaration of a name, you must understand the terms *contained in* and *internal to*.

All of the text of a block, from the PACKAGE, PROCEDURE, or BEGIN statement through the corresponding END statement (including condition prefixes of BEGIN, PACKAGE, and PROCEDURE statements), is said to be contained in that block. However, the labels of the BEGIN or PROCEDURE statement heading the block, as

Scope of declarations

well as the labels of any ENTRY statements that apply to the block, are not contained in that block. Nested blocks are contained in the block in which they appear.

Text that is contained in a block, but not contained in any other block nested within it, is said to be internal to that block. Entry names of a procedure (and labels of a BEGIN statement) are not contained in that block. Consequently, they are internal to the containing block. Entry names of an external procedure are treated as if they were external to the external procedure.

Table 25 illustrates the scopes of data declarations.

Table 25. Scopes of data declarations

```
          A_and_D: package exports(*);
 1,2         dcl X static,
                 Y based;

 2                                            P   Q   Q'  R   R'  S   I
 3            ┌─   A:  procedure;             ┌   ┐
                      declare P, Q;               ┘
 3            │   ┌─  B:  Procedure;                  ┌       ┐
 4            │   │       declare Q;
              │   │       R = Q;
              │   │   ┌─  C:  begin;                          ┌       ┐
              │   │   │       declare R;
              │   │   │       do I = 1 to 10;
              │   │   │       end;
 5            │   │   └─  end C;                              └       ┘
              │   └─  end B;                          └       ┘
              │   ┌─  D:  procedure;          │           ┌       ┐
              │   │       declare S;
              │   └─  end D;                  │           ┘       ┘
              └─  end A;                      └   ┘       ┘       ┘

          end A_and_D;
```
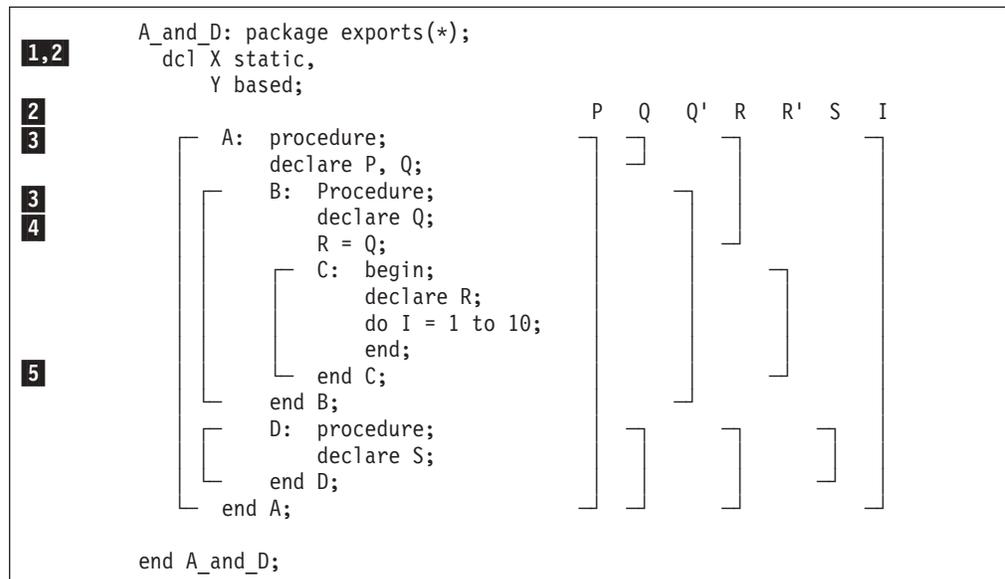
The brackets to the left indicate the block structure; the brackets to the right show the scope of each declaration of a name. The scopes of the two declarations of Q and R are shown as Q and Q' and R and R'.

Note that X and Y are visible to all of the procedures contained in the package.

[1]   P is declared in the block A and known throughout A because it is not redeclared.

[2]   Q is declared in block A, and redeclared in block B. The scope of the first declaration of Q is all of A except B; the scope of the second declaration of Q is block B only.

[3]   R is declared in block C, but a reference to R is also made in block B. The reference to R in block B results in an implicit declaration of R in A, the external procedure. Therefore, two separate names (R and R' in Table 25) with different scopes exist. The scope of the explicitly declared R is block C; the scope of the implicitly declared R in block B is all of A except block C.

[4]   I is referred to in block C. This results in an implicit declaration in the external procedure A. As a result, this declaration applies to all of A, including the contained procedures B, C, and D.

[5]   S is explicitly declared in procedure D and is known only within D.

Table 26 illustrates the scopes of entry constant and statement label declarations.

Table 26. Scopes of entry and label declarations

```
1        A_and_D:  package exports(*);
2                                            L1   L1'  L2   A   B   C   D   E
3          ┌─A:  procedure;
4          │    declare E entry;
6          │    L1:  P = Q;
5          │  ┌─ B:  procedure;
3          │  │     L2:    call C;
           │  │   ┌─ C:  procedure;
           │  │   │    L1:  X = Y;
           │  │   │    call E:
3          │  │   └─ end C;
           │  │      go to L1;
4          │  └─ end B;
           │   ┌─ D:  procedure;
           │   └─   end D;
           │    call B;
2          └─end A;

           ┌─E:  procedure;
           └─end E;

         end A_and_D;
```
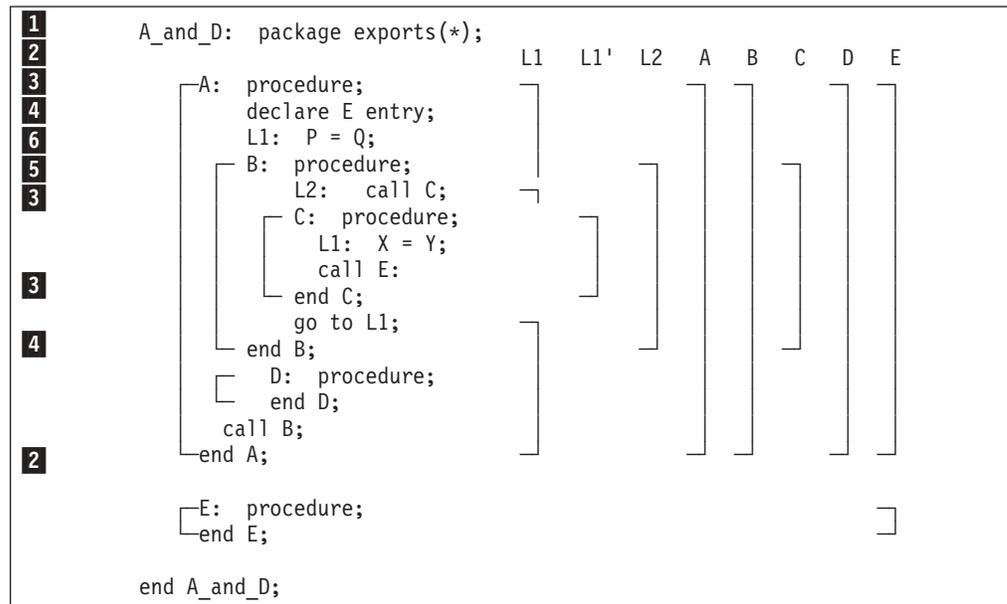
Table 26 shows two external procedures, `A` and `E`.

1   The scope of the declaration of the name `A` is only all of the block `A`, and not `E`.

2   `E` is explicitly declared in `A` as an external entry constant. The explicit declaration of `E` applies throughout block `A`. It is not linked to the explicit declaration of `E` that applies throughout block `E`. The scope of the declaration of the name `E` is all of block `A` and all of block `E`.

3   The label `L1` appears with statements internal to `A` and to `C`. Two separate declarations are therefore established; the first applies to all of block `A` except block `C`, the second applies to block `C` only. Therefore, when the GO TO statement in block `B` executes, control transfers to `L1` in block `A`, and block `B` terminates.

4   D and `B` are explicitly declared in block `A` and can be referred to anywhere within `A`; but because they are INTERNAL, they cannot be referred to in block `E`.

5   `C` is explicitly declared in `B` and can be referred to from within `B`, but not from outside `B`.

6   `L2` is declared in `B` and can be referred to in block `B`, including `C`, which is contained in `B`, but not from outside `B`.

INTERNAL and EXTERNAL attributes

The INTERNAL and EXTERNAL attributes define the scope of a name.

INTERNAL and EXTERNAL

```
►►─────┬─INTERNAL─┬──────────────────────────────────────────►◄
       └─EXTERNAL─┘
                  └─(─environment-name─)─┘
```
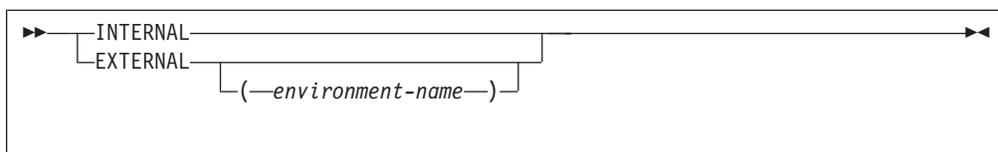
Abbreviations: INT for INTERNAL, EXT for EXTERNAL

environment-name

Specifies the name by which the procedure or variable is known outside of the compilation unit.

When so specified, the name being declared effectively becomes internal and is not known outside of the compilation unit. The environment name is known instead.

The environment name must be a character string constant, and is used as is without any translation to uppercase.

For example:

```
dcl X entry external ('koala');
```

Environment names should not start with a break character (_). Names starting with this character are reserved for the library.

On platforms where the linker *decorates* environment names, if an environment name is specified with the external attribute, it will still be decorated if it differs only in case from the variable name. In the following declaration:

```
dcl abc ext('kLm'), xyz ext('xYz' );
```

The name for xyz is decorated. For more information on the decoration of environment names, refer to the VisualAge PL/I *Programming Guide.*, under the section ″Understanding linkage considerations″ in the ″Calling conventions″ chapter.

INTERNAL is the default for entry names of internal procedures and for variables with any storage class except controlled. INTERNAL specifies that the name can be known only in the declaring block. Any other explicit declaration of that name refers to a new object with a different scope that does not overlap.

Note: INTERNAL may be specified on level-1 procedures in a package. If the package is declared with EXPORTS(*), an INTERNAL procedure is not visible outside the package.

EXTERNAL is the default for file constants, entry constants, programmer-defined conditions, and controlled variables. A name with the EXTERNAL attribute can be declared more than once, either in different external procedures or within blocks contained in external procedures. All declarations of the same name with the EXTERNAL attribute refer to the same data. The scope of each declaration of the name (with the EXTERNAL attribute) includes the scopes of all the declarations of that name (with EXTERNAL) within the application.

When a major structure or union name is declared EXTERNAL in more than one block, the attributes of the members must be the same in each case, although the corresponding member names need not be identical.

In the following example:

```
ProcA: procedure;
declare 1 A external,
2 B,
2 C;
  .
  .
  .

end ProcA;


%process;
ProcB: procedure;
declare 1 A external,
2 B,
2 D;
  .
  .
  .
end ProcB;
```

If A.B is changed in ProcA, it is also changed for ProcB, and vice versa; if A.C is changed in ProcA, A.D is changed for ProcB, and vice versa.

Members of structures and unions always have the INTERNAL attribute.

Because external declarations for the same name all refer to the same data, they must all result in the same set of attributes. When EXTERNAL names are declared in different external procedures, the user has the responsibility to ensure that the attributes are matching. Figure 7 on page 156 illustrates a variety of declarations and their scopes.

```
                Scope_Example:  package exports(*);
1                  A: procedure;
2                     declare S character (20);
7                     dcl Set entry(fixed decimal(1)),
7                         Out entry(label);
                      call Set (3);
9                  E: get list (S,M,N);
8                  B: begin;
4,5                    declare X(M,N), Y(M);
                       get list (X,Y);
                       call C(X,Y);
9,5                C:  procedure (P,Q);
                        declare
                          P(*,*),
                          Q(*),
12,2                      S binary fixed external;
                        S = 0;
6                       do I = 1 to M;
                         if sum (P(I,*)) = Q(I) then
8                           go to B;
                          S = S+1;
                          if S = 3 then
9                             call Out (E);
                          Call D(I);
8          B:  end;
               end C;

9          D:  procedure (N);
                   put list ('Error in row ',
2,3                    N, 'Table Name ', S);
               end D;
               end B;
               go to E;
               end A;
9          Out:  procedure (R);
                   Declare
                     R Label,
11                   (K static internal,
11,7                    L static external) init (0),
12                   S binary fixed external,
                     Z fixed decimal(1);
                   K = K+1; S=0;
                   if K<L then
                     stop;
10                 else go to R;
           end;
           Set:  procedure (Z);
                   declare Z fixed dec(1);
7                  L=Z;
                   declare L external init(0);
                   return;
                   end;
             end Scope_Example;
```

Figure 7. Example of scopes of various declarations

1   A is an external procedure name. Its scope is all of block A, plus any other blocks where A is declared as external.

2   S is explicitly declared in block A and block C. The character variable declaration applies to all of block A except block C. The fixed binary declaration applies only within block C. Notice that although D is called from within block C, the reference to S in the PUT statement in D is to the character variable S, and not to the S declared in block C.

3 N appears as a parameter in block D, but is also used outside the block. Its appearance as a parameter establishes an explicit declaration of N within D. The references outside D cause an implicit declaration of N in block A. These two declarations of the name N refer to different objects, although in this case, the objects have the same data attributes, which are, by default, FIXED BINARY(15,0) and INTERNAL. Under DEFAULT(ANS), the precision is (31,0).

4 X and Y are known throughout B and can be referred to in block C or D within B, but not in that part of A outside B.

5 P and Q are parameters, and therefore if there were no other declaration of these names within the block, their appearance in the parameter list would be sufficient to constitute a contextual declaration. However, a separate, explicit declaration statement is required in order to specify that P and Q are arrays. Although the arguments X and Y are declared as arrays and are known in block C, it is still necessary to declare P and Q in a DECLARE statement to establish that they, too, are arrays. (The asterisk notation indicates that the bounds of the parameters are the same as the bounds of the arguments.)

6 I and M are not explicitly declared in the external procedure A. Therefore, they are implicitly declared and are known throughout A, even though I appears only within block C.

7 The Out and Set external procedures in the example have an external declaration of L that is common to both. They also must be declared explicitly with the ENTRY attribute in procedure A. Because ENTRY implies EXTERNAL, the two entry constants Set and Out are known throughout the two external procedures.

8 The label B appears twice in the program—first in A, as the label of a begin-block, which is an explicit declaration, and then redeclared as a label within block C by its appearance as a prefix to an END statement. The go to B statement within block C, therefore, refers to the label of the END statement within block C. Outside block C, any reference to B is to the label of the begin-block.

9 Blocks C and D can be called from any point within B but not from that part of A outside B, nor from another external procedure. Similarly, because label E is known throughout the external procedure A, a transfer to E can be made from any point within A. The label B within block C, however, can be referred to only from within C. Transfers out of a block by a GO TO statement can be made; but such transfers into a nested block generally cannot. An exception is shown in the external procedure Out, where the label E from block C is passed as an argument to the label parameter R.

Note that, with no files specified in the GET and PUT statements, SYSIN and SYSPRINT are implicitly declared.

10 The statement else go to R; transfers control to the label E, even though E is declared within A, and not known within Out.

11 The variables K (INTERNAL) and L (EXTERNAL) are declared as STATIC within the Out procedure block; their values are preserved between calls to Out.

12 In order to identify the S in the procedure Out as the same S in the procedure C, both are declared with the attribute EXTERNAL.

RESERVED attribute

The RESERVED attribute implies STATIC EXTERNAL. Moreover, if a variable has the RESERVED attribute, then the application must comply with the following conditions:

- All declarations of the variable must specify RESERVED.
- The variable name must appear in the RESERVES option of exactly one package.

If a variable has the RESERVED attribute, any INITIAL values are ignored except in the package reserving it.

```
►►──RESERVED──────────────────────────────────────────►◄
            └─(IMPORTED)─┘
```
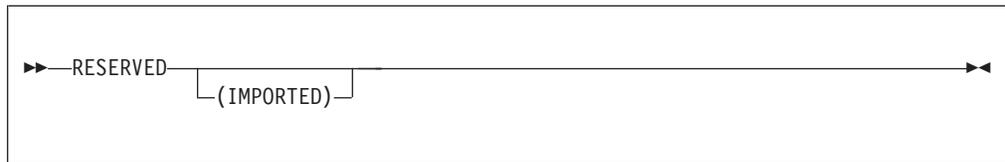
If a compilation unit has a variable with the RESERVED attribute and is not the reserving package for that variable, then that compilation unit either must be part of the load module containing the reserving package or must import the variable from another load module containing the reserving package. In the latter case, the declaration must specify the IMPORTED option of the RESERVED attribute.

```
owns_x:
  package
  exports(*)
  reserves(x);

  dcl x char(256) reserved init( ... );
  dcl y char(256) reserved init( ... );
  dcl z char(256) reserved(imported) init( ... );

end;

owns_y:
  package
  exports(*)
  reserves(y);

  dcl x char(256) reserved init( ... );
  dcl y char(256) reserved init( ... );
  dcl z char(256) reserved(imported) init( ... );

end;

owns_z:
  package
  exports(*)
  reserves(z);

  dcl z char(256) reserved(imported) init( ... );

end;
```

In the preceding example, the package *owns_x* reserves and initializes the storage for the variable *x*. It must be linked into the same load module as the package *owns_y*. This load module must import the variable *z* from the load module into which package *owns_z* is linked.

Data alignment

The computer holds information in multiples of units of 8 bits. Each 8-bit unit of information is called a *byte*.

The computer accesses bytes singly or as halfwords, words, or doublewords. A *halfword* is 2 consecutive bytes. A *fullword* is 4 consecutive bytes. A *doubleword* is 8 consecutive bytes. Byte locations in storage are consecutively numbered starting with 0; each number is the address of the corresponding byte. Halfwords, words, and doublewords are addressed by the address of their leftmost byte.

Your programs can execute faster if halfwords, words, and doublewords are located in main storage on an integral boundary for that unit of information. That is, the unit of information's address is a multiple of the number of bytes in the unit, as can be seen in Table 27.

Table 27. Alignment on integral boundaries of halfwords, words, and doublewords

ADDRESSES IN A SECTION OF STORAGE							
5000	5001	5002	5003	5004	5005	5006	5007
byte	byte	byte	byte	byte	byte	byte	byte
halfword		halfword		halfword		halfword	
fullword				fullword			
doubleword							

PL/I allows data alignment on integral boundaries. However, unused bytes between successive data elements can increase storage use. For example, when the data items are members of aggregates used to create a data set, the unused bytes increase the amount of auxiliary storage required. The ALIGNED and UNALIGNED attributes allow you to choose whether or not to align data on the appropriate integral boundary.

ALIGNED and UNALIGNED attributes

ALIGNED specifies that the data element is aligned on the storage boundary corresponding to its data-type requirement. UNALIGNED specifies that each data element is mapped on the next byte boundary, except for fixed-length bit strings, which are mapped on the next bit.

```
►►─┬─ALIGNED───┬──────────────────────────────────────────────────►◄
   └─UNALIGNED─┘
```
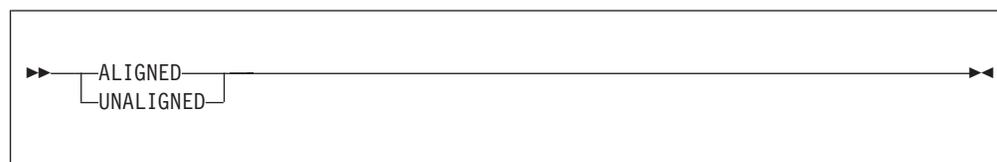
Defaults are applied at element level. UNALIGNED is the default for bit data, character data, graphic data, widechar data and numeric character data. ALIGNED is the default for all other types of data.

Requirements for the ALIGNED attribute are shown in Table 28 on page 160.

ALIGNED and UNALIGNED attributes

Table 28. Alignment requirements

Variable Type	Stored Internally as:	Storage Requirements (Bytes)	Alignment Requirements	
			ALIGNED Data	UNALIGNEDData
Note:				
Alignment and storage requirements for program control data can vary across supported systems.				
Complex data requires twice as much storage as its real counterpart, but the alignment requirements are the same.				
BIT (n)	ALIGNED:One byte for each group of 8 bits (or part thereof) UNALIGNED: As many bits as are required, regardless of byte boundaries	ALIGNED: CEIL(n/8) UNALIGNED: n bits	Byte(Data can begin on any byte, 0 through 7)	Bit(Data can begin on any bit in any byte, 0 through 7)
CHARACTER (n)	One byte per character	n	Byte(Data can begin on any byte, 0 through 7)	Byte(Data can begin on any byte, 0 through 7)
CHARACTER (n)VARYINGZ	One byte per character plusone byte for the nullterminator	n+1		
GRAPHIC (n)	Two bytes per graphic	2n		
GRAPHIC (n)VARYINGZ	Two bytes per graphic plustwo bytes for the nullterminator	2n+2		
WIDECHAR (n)	Two bytes per widechar.	2n		
WIDECHAR (n)VARYINGZ	Two bytes per widechar plustwo bytes for the nullterminator	2n+2		
PICTURE	One byte for each PICTURE character (except V, K, and the F scaling factor specification)	Number of PICTURE characters other than V, K, and F specification		
DECIMAL FIXED (p,q)	Packed decimal format (1/2 byte per digit, plus 1/2 byte for sign)	CEIL((p+1)/2)		
BINARY FIXED(p,q) SIGNED 1 <= p <= 7 UNSIGNED 1 <= p <= 8 ORDINAL SIGNED 1 <= p <= 7 UNSIGNED 1 <= p <= 8	One byte	1		

Table 28. Alignment requirements (continued)

Variable Type	Stored Internally as:	Storage Requirements (Bytes)	Alignment Requirements	
			ALIGNED Data	UNALIGNEDData
BIT(n) VARYING	Two-byte prefix plus 1 byte for each group of 8 bits (or part thereof) of the declared maximum length	ALIGNED:2+CEIL(n/8) UNALIGNED: 2 bytes+n bits	Halfword (Data can begin on byte 0, 2, 4, or 6)	Byte(Data can begin on any byte, 0 through 7)
CHARACTER(n)VARYING	Two-byte prefix plus 1 byte per character of the declared maximum length	n+2		
GRAPHIC(n)VARYING	Two-byte prefix plus 2 bytes per graphic of the declared maximum length	2n+2		
WIDECHAR(n)VARYING	Two-byte prefix plus 2 bytes per widechar of the declared maximum length	2n+2		
BINARY FIXED(p,q) SIGNED 8 <= p <= 15 UNSIGNED 9 <= p <= 16 ORDINAL SIGNED 8 <= p <= 15 UNSIGNED 9 <= p <= 16	Halfword	2		
BINARY FIXED(p,q) SIGNED 16 <= p <= 31 UNSIGNED 17 <= p <= 32 ORDINAL SIGNED 16 <= p <= 31 UNSIGNED 17 <= p <= 32	Fullword	4	Fullword (Data can begin on byte 0 or 4)	Byte(Data can begin on any byte, 0 through 7)
BINARY FLOAT(p)1<=p<=21 DECIMAL FLOAT(p)1<=p<=6	Short floating-point			
POINTER	–	4	Fullword(Data can begin on byte 0 or 4)	Byte(Data can begin on any byte, 0 through 7)
HANDLE	–			
OFFSET	–			
FILE	–			
ENTRY LIMITED	–			
ENTRY	–	8		
LABEL or FORMAT	–			
TASK	–	16		

ALIGNED and UNALIGNED attributes

Table 28. Alignment requirements (continued)

Variable Type	Stored Internally as:	Storage Requirements (Bytes)	Alignment Requirements	
			ALIGNED Data	UNALIGNEDData
AREA	–	16+size	Doubleword (Data can begin on byte 0)	AREA data cannot be unaligned
BINARY FIXED(p,q) SIGNED 32 <= p <= 63 UNSIGNED 33 <= p <= 64	–	8		byte(Data can begin on any byte, 0 through 7)
BINARY FLOAT(p) 22 <= p <= 53 DECIMAL FLOAT(p) 7 <= p <= 16	Long floating-point			

BINARY FLOAT(p)
   54<=p
 	
  DECIMAL FLOAT(p)
      17<=p
	Extended floating-point	16		

ALIGNED or UNALIGNED can be specified for element, array, structure, or union variables. The application of either attribute to a structure or union is equivalent to applying the attribute to all contained elements that are not explicitly declared ALIGNED or UNALIGNED.

The following example illustrates the effect of ALIGNED and UNALIGNED declarations for a structure and its elements:

```
declare 1 S,
          2 X bit(2),      /* unaligned by default  */
          2 A aligned,     /* aligned explicitly    */
           3 B,            /* aligned from A        */
           3 C unaligned,  /* unaligned explicitly  */
            4 D,           /* unaligned from C      */
            4 E aligned,   /* aligned explicitly    */
            4 F,           /* unaligned from C      */
           3 G,            /* aligned from A        */
          2 H;             /* aligned by default    */
```

For more information about structures and unions, refer to "Structures" on page 170 and "Unions" on page 172.

Defaults for attributes

Every name in a PL/I program requires a complete set of attributes. Arguments passed to a procedure must have attributes matching the procedure's parameters. Values returned by functions must have the attributes expected. However, the attributes that you specify need rarely include the complete set of attributes.

The set of attributes for:
- Explicitly declared names
- Implicitly (including contextually) declared names
- Attributes to be included in parameter descriptors
- Values returned from function procedures

can be completed by using the language-specified defaults, or by defaults that you can define (using the DEFAULT statement) either to modify the language-specified defaults or to develop a completely new set of defaults.

Attributes applied by default cannot override attributes applied to a name by explicit or contextual declaration.

Language-specified defaults

When a variable has not been declared with any data attributes, it is given arithmetic attributes by default. If mode, scale, and base are not specified by a DECLARE or DEFAULT statement, the DEFAULT compiler option determines its attributes as follows:

- If DEFAULT(IBM) is in effect, variables with names beginning with the letters I through N are given the attributes REAL FIXED BINARY(15,0); all other variables are given the attributes REAL FLOAT DECIMAL(6).
- If DEFAULT(ANS) is in effect, all variables are given the attributes REAL FIXED BINARY(31,0).

If a scaling factor is specified in the precision attribute, the attribute FIXED is applied before any other attributes. Therefore, a declaration with the attributes BINARY(p,q) is always equivalent to a declaration with the attributes FIXED BINARY(p,q).

If a precision is not specified in an arithmetic declaration, the DEFAULT compiler option determines the precision as indicated in Table 29. The language-specified defaults for scope, storage and alignment attributes are shown in Table 8 on page 27 and Table 7 on page 26.

If no description list is given in an ENTRY declaration, the attributes for the argument must match those specified for the corresponding parameter in the invoked procedure. For example, given the following declaration:

```
dcl X entry;
call X( 1 );
```

The argument has the attributes REAL FIXED DECIMAL(1,0). This would be an error if the procedure x declared its parameter with other attributes, as shown in the following example:

```
X: proc( Y );
  dcl Y fixed bin(15);
```

This potential problem can be easily avoided if the entry declaration specifies the attributes for all of its parameters.

Table 29. Default arithmetic precisions

Attributes	DEFAULT(IBM)	DEFAULT(ANS)
DECIMAL FIXED	(5,0)	(10,0)
BINARY FIXED	(15,0)	(31,0)
DECIMAL FLOAT	(6)	(6)
BINARY FLOAT	(21)	(21)

DEFAULT statement

The DEFAULT statement specifies data-attribute defaults (when attribute sets are not complete). Any attributes not applied by the DEFAULT statement for any partially-complete explicit or contextual declarations, and for implicit declarations, are supplied by language-specified defaults.

DEFAULT

The DEFAULT statement overrides all other attribute specifications, except that a name declared with the ENTRY or FILE attribute, but none of the attributes that would imply the VARIABLE attribute, will be given the implicit CONSTANT attribute by PL/I before any DEFAULT statements are applied. Consequently, in the following example, PL/I gives `Xtrn` the CONSTANT attribute and not the STATIC attribute.

```
Sample: proc;

  default range(*) static;
  dcl Xtrn entry;

end;
```

Structure and union elements are given default attributes according to the name of the element, not the qualified element name. The DEFAULT statement cannot be used to create a structure or a union.
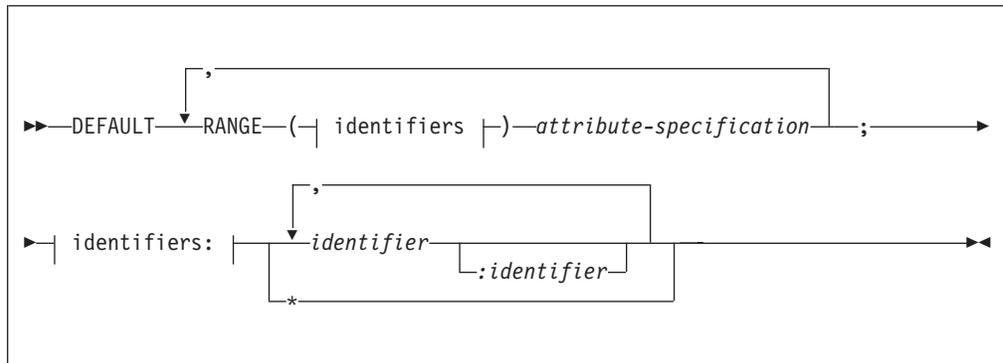
Abbreviation: DFT

RANGE(identifier)
  Specifies that the defaults apply to names that begin with the same letters as in the identifier specified. For example:

```
RANGE (ABC)
```

  applies to these names:

```
ABC
ABCD
ABCDE
```

  but not to:

```
ABD
ACB
AB
A
```

  Hence a one-letter identifier in the range-specification applies to all names that start with that letter. The RANGE identifier may be specified in DBCS.

RANGE(identifier : identifier)
  Specifies that the defaults apply to names with initial letters that either match the two identifiers specified or fall between the two in alphabetic sequence. The letters may be in DBCS, but in determining if a RANGE specification applies to a name, all comparisons are based solely on the hex values of the letters involved. The letters given in the specification must be in increasing alphabetic order. For example:

```
RANGE(A:G,I:M,T:Z)
```

RANGE(*)

Specifies all names in the scope of the DEFAULT statement. For example:

```
DFT RANGE (*) PIC '99999';
```

This statement specifies default attributes REAL PICTURE '99999' for all names.

An example of a *factored-specification* with the range options is:

```
DEFAULT (RANGE(A)FIXED, RANGE(B)
        FLOAT)BINARY;
```

This statement specifies default attributes FIXED BINARY for names with the initial letter A, and FLOAT BINARY for those with the initial letter B.

DESCRIPTORS

Specifies that the attributes are included in any parameter descriptors in a parameter descriptor list of an explicit entry declaration, provided that:

* The inclusion of any such attributes is not prohibited by the presence of alternative attributes of the same class.
* At least one attribute is already present. (The DESCRIPTORS default attributes are not applied to null descriptors).

For example:

```
DEFAULT DESCRIPTORS BINARY;
DCL X ENTRY (FIXED, FLOAT);
```

The attribute BINARY is added to each parameter descriptor in the list, producing the equivalent list:

```
(FIXED BINARY, FLOAT BINARY)
```

attribute-list

Specifies a list of attributes from which selected attributes are applied to names in the specified range. Attributes in the list can appear in any order and must be separated by blanks.

Only those attributes that are necessary to complete the declaration of a data item are taken from the list of attributes.

If FILE is used, it implies the attributes VARIABLE and INTERNAL.

The dimension attribute is allowed, but only as the first item in an attribute specification. The bounds can be specified as an arithmetic constant or an expression and can include the REFER option. For example:

```
DFT RANGE(J) (5);
DFT RANGE(J) (5,5) FIXED;
```

Although the DEFAULT statement can specify the dimension attribute for names that have not been declared explicitly, a subscripted name is contextually declared with the attribute BUILTIN. Therefore, the dimension attribute can be applied by default only to explicitly declared names.

The INITIAL attribute can be specified.

Attributes that conflict, when applied to a data item, do not necessarily conflict when they appear in an attribute specification. For example:

```
DEFAULT RANGE(S) BINARY VARYING;
```

DEFAULT

This means that any name that begins with the letter S and is declared explicitly with the BIT, CHARACTER, or GRAPHIC attribute receives the VARYING attribute; all others (that are not declared explicitly or contextually as other than arithmetic data) receive the BINARY attribute.

VALUE

Can appear anywhere within an attribute-specification except before a dimension attribute.

VALUE establishes any default rules for an area size, string length, and precision.

The size of AREA data, or length of BIT, CHARACTER, or GRAPHIC data, can be an expression or an integer and can include the REFER option, or can be specified as an asterisk.

For example:

```
DEFAULT RANGE(A:C)
        VALUE (FIXED DEC(10),
               FLOAT DEC(14),
               AREA(2000));
DECLARE B FIXED DECIMAL,
        C FLOAT DECIMAL,
        A AREA;
```

These statements are equivalent to:

```
DECLARE B FIXED DECIMAL(10),
        C FLOAT DECIMAL(14),
        A AREA(2000);
```

The base and scale attributes in value-specification must be present to identify a precision specification with a particular attribute. The base and scale attributes can be factored (see "Factoring attributes" on page 150).

The only attributes that the VALUE option can influence are area size, string length, and precision. Other attributes in the option, such as CHARACTER and FIXED BINARY in the above examples, merely indicate which attributes the value is to be associated with. Consider the following example:

```
DEFAULT RANGE(I) VALUE(FIXED DECIMAL(8,3));
I = 1;
```

If it is not declared explicitly, I is given the language-specified default attributes FIXED BINARY(15,0). It is *not* influenced by the default statement, because this statement specifies only that the default precision for FIXED DECIMAL names is to be (8,3).

For example:

```
DFT RANGE(*) VALUE(FIXED BINARY(31));
```

specifies precision for identifiers already known to be FIXED BINARY, while

```
DFT RANGE(*) FIXED BINARY VALUE(FIXED BINARY(31));
```

specifies both the FIXED BINARY attribute as a default and the precision.

There can be more than one DEFAULT statement within a block. The scope of a DEFAULT statement is the block in which it occurs, and all blocks within that block which neither include another DEFAULT statement with the same range, nor are contained in a block having a DEFAULT statement with the same range.

A DEFAULT statement in an internal block affects only explicitly declared names. This is because the scope of an implicit declaration is determined as if the names were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name appears.

It is possible for a containing block to have a DEFAULT statement with a range that is partly covered by the range of a DEFAULT statement in a contained block. In such a case, the range of the DEFAULT statement in the containing block is reduced by the range of the DEFAULT statement in the contained block. For example:

```
 P:  PROCEDURE;
L1:  DEFAULT RANGE (XY) FIXED;
 Q:  BEGIN;
L2:  DEFAULT RANGE (XYZ) FLOAT;
     END P;
```

The scope of DEFAULT statement L1 is procedure P and the contained block Q. The range of DEFAULT statement L1 is all names in procedure P beginning with the characters XY, together with all names in begin-block Q beginning with the characters XY, except for those beginning with the characters XYZ.

Labels can be prefixed to DEFAULT statements. A branch to such a label is treated as a branch to a null statement. Condition prefixes cannot be attached to a DEFAULT statement.

Restoring language-specified defaults

The following statement:

```
  dft range(*) system;
```

overrides, for all names, any programmer-defined default rules established in a containing block. It can be used to restore language-specified defaults for contained blocks.

Arrays

An *array* is an n-dimensional collection of elements that have identical attributes. Only the array itself is given a name. An individual item of an array is referred to by giving its position within the array. You indicate that a name is an *array variable* by providing the dimension attribute.

Every array must have at least one element.

DIMENSION attribute

The *dimension attribute* specifies the number of dimensions of an array and upper and lower bounds of each.

Bounds that are nonrestricted expressions are evaluated and converted to FIXED BINARY(31,0) when storage is allocated for the array.

The *extent* is the number of integers between, and including, the lower and upper bounds.

DIMENSION attribute

```
        ┌─────────,────────┐
        │                  │
►►──┬──────────────┬──(──▼─┤ bound ├──)──┤ bound: ├─────────────────────►
    └──DIMENSION───┘                              ┌──────────────────┐
                                                  ├─┤ lower-bound ├─:─┤
                                                  └─*:───────────────┘

►──┬─────────────────┬───────────────────────────────────────────────►◄
   ├─┤ upper-bound ├─┤
   └─*───────────────┘


lower-bound:

   ┌─1────────────────────────────────┐
├──┤                                  ├──────────────────────────────────┤
   └─expression─┬───────────────────┬─┘
               └─REFER(variable)───┘


upper-bound:

├────expression─┬───────────────────┬──────────────────────────────────┤
               └─REFER(variable)───┘
```
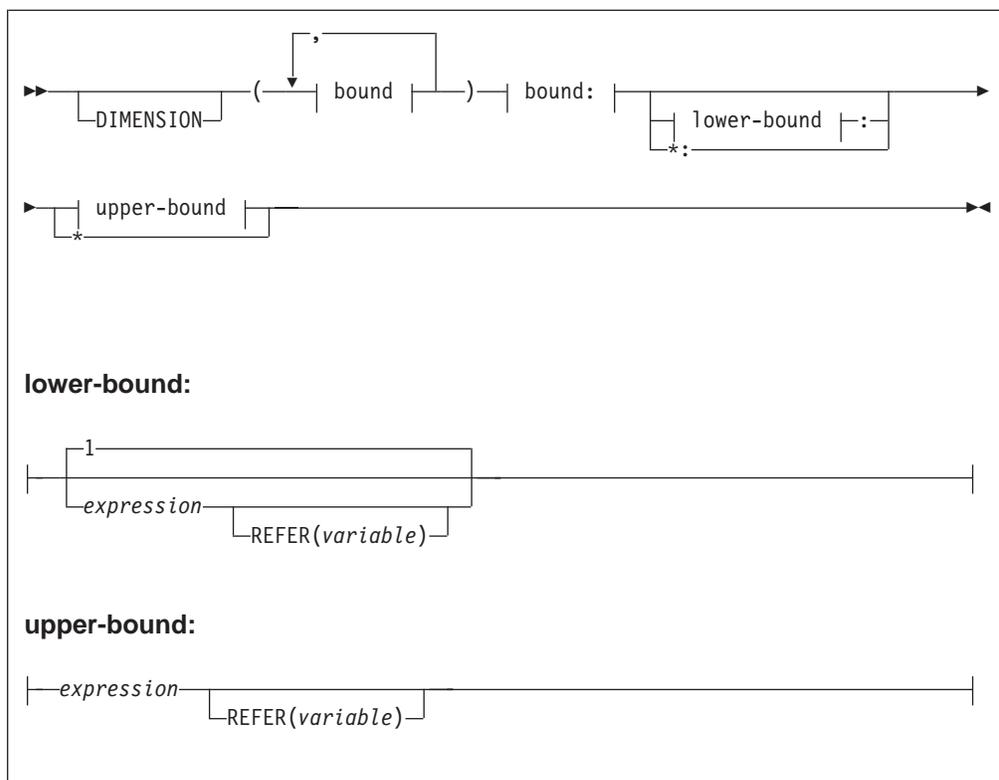
Abbreviation: DIM

If the DIMENSION keyword is omitted, the dimension must immediately follow the name (or the parenthesized list of names) in the declaration.

The number of bounds specifications indicates the number of dimensions in the array, unless the declared variable is in an array of structures or unions. In this case it inherits dimensions from the containing structure or union.

The bounds specification indicates the bounds as follows:
- If only the upper bound is given, the lower bound defaults to 1.
- The lower bound must be less than or equal to the upper bound.
- An asterisk (*) specifies that the lower and/or the upper bound is taken from the argument associated with the parameter.

Examples of arrays

Consider the following declaration:

```
declare List fixed decimal(3) dimension(8);
```

List is declared as a one-dimensional array of eight elements, each one a fixed-point decimal element of three digits. The one dimension of List has bounds of 1 and 8, and its extent is 8.

In the example:

```
declare Table (4,2) fixed dec (3);
```

`Table` is declared as a two-dimensional array of eight fixed-point decimal elements. The two dimensions of `Table` have bounds of 1 and 4 and 1 and 2, and the extents are 4 and 2.

Other examples are:

```
declare List_A dimension(4:11);
declare List_B (-4:3);
```

In the first example, the bounds are 4 and 11; in the second they are -4 and 3. The extents are the same for each, 8 integers from the lower bound through the upper bound.

In the manipulation of array data (discussed in "Array expressions" on page 69) involving more than one array, the bounds—not merely the extents—must be identical. Although `List`, `List_A`, and `List_B` all have the same extent, the bounds are not identical.

Subscripts

The bounds of an array determine the way elements of the array can be referred to. For example, when the following data items:

```
20 5 10 30 630 150 310 70
```

are assigned to the array `List`, as declared above, the different elements are referred to as follows:

Reference	Element
LIST (1)	20
LIST (2)	5
LIST (3)	10
LIST (4)	30
LIST (5)	630
LIST (6)	150
LIST (7)	310
LIST (8)	70

Each of the parenthesized numbers following LIST is a *subscript*. A parenthesized subscript following an array name reference identifies a particular data item within the array. A reference to a subscripted name, such as LIST(4), refers to a single element and is an element variable. The entire array can be referred to by the unsubscripted name of the array—for example, LIST.

The same data can be assigned to `List_A` and `List_B` declared previously. In this case it is referenced as follows:

Reference	Element	Reference
LIST_A (4)	20	LIST_B (-4)
LIST_A (5)	5	LIST_B (-3)
LIST_A (6)	10	LIST_B (-2)
LIST_A (7)	30	LIST_B (-1)
LIST_A (8)	630	LIST_B (0)
LIST_A (9)	150	LIST_B (1)
LIST_A (10)	310	LIST_B (2)
LIST_A (11)	70	LIST_B (3)

Assume that the same data is assigned to TABLE, which is declared as a two-dimensional array. TABLE can be illustrated as a matrix of four rows and two columns:

TABLE(m,n)	(m,1)	(m,2)
(1,n)	20	5
(2,n)	10	30
(3,n)	630	150
(4,n)	310	70

An element of TABLE is referred to by a subscripted name with two parenthesized subscripts, separated by a comma. For example, TABLE (2,1) would specify the first item in the second row, the data item 10.

The use of a matrix to illustrate TABLE is purely conceptual. It has no relationship to the way the items are actually organized in storage. Data items are assigned to an array in row major order. This means that the subscript that represents columns varies most rapidly. For example, assignment to TABLE would be to TABLE(1,1), TABLE(1,2), TABLE(2,1), TABLE(2,2), and so forth.

A subscripted reference to an array must contain as many subscripts as there are dimensions in the array.

Any expression that yields a valid arithmetic value can be used for a subscript. If necessary, the value is converted to FIXED BINARY(31,0). Thus, TABLE(I,J*K) can be used to refer to the different elements of TABLE by varying the values of I, J, and K.

Cross sections of arrays

Cross sections of arrays can be referred to by using an asterisk for a subscript. The asterisk specifies that the entire extent is used. For example, TABLE(*,1) refers to all of the elements in the first column of TABLE. It specifies the cross section consisting of TABLE(1,1), TABLE(2,1), TABLE(3,1), and TABLE(4,1). The subscripted name TABLE(2,*) refers to all of the data items in the second row of TABLE. TABLE(*,*) refers to the entire array, as does TABLE.

A subscripted name containing asterisk subscripts represents not a single data element, but an array with as many dimensions as there are asterisks. Consequently, such a name is not an element expression, but an array expression.

A reference to a cross section of an array can refer to two or more elements that are not adjacent in storage. The storage represented by such a cross section is known as *nonconnected* storage. (See "CONNECTED and NONCONNECTED attributes" on page 245.) The rule is as follows: if a nonasterisk bound appears to the right of the leftmost asterisk bound, the array cross section is in nonconnected storage. Thus A(4,*,*) is in connected storage; A(*,2,*) is not.

Structures

A structure is a collection of member elements that can be structures, unions, elementary variables and arrays.

The *structure variable* is a name that can be used to refer to the entire aggregate of data. Unlike an array, however, each member of a structure also has a name, and

the attributes of each member can differ. An asterisk can be used as the name of a structure or a member when it will not be referred to. For example, reserved or filler items can be named with an asterisk.

A structure has different *levels*. The name at level-1 is called a *major structure*. Names at deeper levels can be *minor structures or unions*. Names at the deepest level are called *elementary* names, which can represent an elementary variable or an array variable. Unions are described in "Unions" on page 172.

A structure is described in a DECLARE statement through the use of level-numbers preceding the associated names. Level-numbers must be integers.

A major structure name is declared with the level-number 1. Minor structures, unions, and elementary names are declared with level-numbers greater than 1. A delimiter must separate the level-number and its associated name. For example, the items of a payroll record could be declared as follows:

```
declare 1 Payroll,                  /*  major structure name */
          2 Name,                   /*  minor structure name */
            3 Last char(20),        /*  elementary name      */
            3 First char(15),
          2 Hours,
            3 Regular fixed dec(5,2),
            3 Overtime fixed dec(5,2),
          2 Rate,
            3 Regular fixed dec(3,2),
            3 Overtime fixed dec(3,2);
```

In the example, `Payroll` is the major structure and all other names are members of this structure. `Name`, `Hours`, and `Rate` are minor structures, and all other members are elementary variables. You can refer to the entire structure by the name `Payroll`, or to portions of the structure by the minor structure names. You can refer to a member by referring to the member name.

Indentation is only for readability. The statement could be written in a continuous string as:

```
Declare 1 Payroll, 2 Name, 3 Last char(20), . . .
```

The level-numbers you choose for successively deeper levels need not be consecutive. A minor structure at level n contains all the names with level-numbers greater than n that lie between that minor structure name and the next name with a level-number less than or equal to n.

For example, the following declaration results in exactly the same structure as the declaration in the previous example.

```
Declare 1 Payroll,
          4 Name,
            5 Last char(20),
            5 First char(15),
          3 Hours,
            6 Regular fixed dec(5,2),
            5 Overtime fixed dec(5,2),
          2 Rate,
            9 Regular fixed dec(3,2),
            9 Overtime fixed dec(3,2);
```

The description of a major structure is usually terminated by a semicolon terminating the DECLARE statement. It can also be terminated by comma, followed by the declaration of another item.

Unions

A union is a collection of member elements that overlay each other, occupying the same storage. The members can be structures, unions, elementary variables, and arrays. They need not have identical attributes.

The entire union is given a name that can be used to refer to the entire aggregate of data. Like a structure, each element of a union also has a name. An asterisk can be used as the name of a union or a member, when it will not be referred to. For example, reserved or filler items can be named asterisk.

Like a structure, a union can be at any level including level 1. All elements of a union at the next deeper level are members of the union and occupy the same storage. The storage occupied by the union is equal to the storage required by the largest member. Normally, only one member is used at any time and the programmer determines which member is used.

A union, like a structure, is declared through the use of level-numbers preceding the associated names.

Unions can be used to declare variant records that would typically contain a common part, a selector part, and variant parts. For example, records in a client file can be declared as follows:

```
Declare 1 Client,
          2 Number pic '999999',
          2 Type bit(1),
          2 * bit(7),
          2 Name union,
            3 Individual,
              5 Last_Name char(20),
              5 First_Name union,
                7 First   char(15),
                7 Initial char(1),
            3 Company char(35),
          2 * char(0);
```

In this example, `Client` is a major structure. The structure `Individual`, and the element `Company` are members of the union `Name`. One of these members is active depending on `Type`. The structure `Individual` contains the union `First_name` and the element `Last_name`. `First_name` union has `First` and `Initial` as its members, both of which are active. The example also shows the use of asterisk as a name. The description of a union is terminated by the semicolon that terminates a DECLARE statement or by a comma, followed by the declaration of another item.

UNION attribute

The UNION attribute allows you to specify that a variable is a union and that its members are those that follow it and are at the next logically higher level. CELL is accepted as a synonym for UNION.

```
►►──UNION─────────────────────────────────────────────────────────────►◄
```
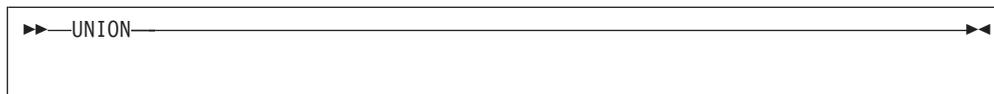
Structure/union qualification

A member of a structure or a union can be referred to by its name alone if it is unique. If another member has the same name, whether at the same or different level, ambiguity occurs. Where ambiguity occurs, a qualified reference is required to uniquely identify the correct member.

A *qualified reference* is a member name that is qualified with one or more names of parent members connected by periods. (See the qualified reference syntax in Chapter 4, "Expressions and references," on page 51.) Blanks can appear surrounding the period.

The qualification must follow the order of levels. That is, the name at the highest level must appear first, with the name at the deepest level appearing last.

While the level-1 structure or union name must be unique within the block scope, member names need not be unique as long as they do not appear at same logical level within their most immediate parent. A qualified name must be used only so far as necessary to make a reference of the same structure unique within the block in which it appears. In the following example, the value of x.y (19) is displayed, not the value (17).

```
dcl Y fixed init(17);


begin;
dcl
1 X,
2 Y fixed init(19);
display( Y );
end;
```

A reference is always taken to apply to the declared name in the innermost block containing the reference.

The following examples illustrate both ambiguous and unambiguous references. In the following example, A.C refers to C in the inner block; D.E refers to E in the outer block.

```
  declare 1 A, 2 C, 2 D, 3 E;
     begin;
     declare 1 A, 2 B, 3 C, 3 E;
  A.C = D.E;
```

In the following example, D has been declared twice. A reference to A.D refers to the second D, because A.D is a complete qualification of only the second D. The first D is referred to as A.C.D.

```
  declare 1 A,
           2 B,
           2 C,
             3 D,
           2 D;
```

In the following example, a reference to A.C is ambiguous because neither C can be completely qualified by this reference.

Structure/union qualification

```
declare 1 A,
          2 B,
            3 C,
          2 D,
            3 C;
```

In the following example, a reference to `A` refers to the first `A`, `A.A` to the second `A`, and `A.A.A` to the third `A`.

```
declare 1 A,
          2 A,
            3 A;
```

In the following example, a reference to `X` refers to the first DECLARE statement. A reference to `Y.Z` is ambiguous. `Y.Y.Z` refers to the second `Z`, and `Y.X.Z` refers to the first `Z`.

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

For more information about name qualification, refer to "Scope of declarations" on page 151.

LIKE attribute

The LIKE attribute specifies that the name being declared has an organization that is logically the same as the referenced structure or union (object of the LIKE attribute). The object variable's member names and their attributes, including the dimension attribute, are effectively copied and become members of the name being declared. If necessary, the level-numbers of the copied members are automatically adjusted. The object variable name and its attributes, including the dimension attribute, are ignored.

```
►►──LIKE──object-variable────────────────────────────────►◄
```
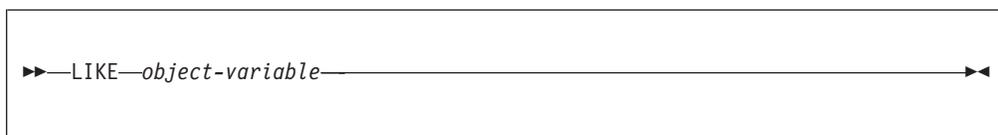
object-variable
Can be a major structure, a minor structure, or a union. It must be known in the block containing the LIKE attribute specification. It can be qualified but must not be subscripted. The object or any of its members must not have the LIKE attribute or the REFER option.

The objects in all LIKE attributes are associated with declared names before any LIKE attributes are expanded.

New members cannot be added to the created structure or union. Any level-number that immediately follows the object variable in the LIKE attribute must be equal to or less than the level-number of the name with the LIKE attribute.

The following declarations yield the same structure for `X`.

```
dcl
  1 A(10) aligned static,
    2 B    bit(4),
    2 C    bit(4),
  1 X like A;
```

```
dcl
  1 X,
    2 B bit(4),
    2 C bit(4);
```

Notice that the dimension (DIM(10)), ALIGNED, and STATIC attributes are not copied as part of the LIKE expansion.

The LIKE attribute is expanded before the defaults are applied and before the ALIGNED and UNALIGNED attributes are applied to the contained elements of the LIKE object variable.

Examples

```
declare 1 A,
        2 C,
          3 E(3) union,
            5 E1,
            5 E2,
          3 F;
declare 1 B(10) union,
        2 C, 3 G, 3 H,
        2 D;
begin;
declare 1 C like B;
declare 1 D(2),
        5 BB like A.C;
end;
```

Declarations `C` and `D` have the results shown in the following example.

```
dcl
    1 C,                /* DIM and UNION not copied. */
      2 C, 3 G, 3 H,
      2 D;
```

```
dcl 1 D(2),
      5 BB,
        6 E(3) union,  /* DIM(3) and UNION copied.     */
          7 E1,        /* Note adjusted level-numbers.  */
          7 E2,
        6 F;
```

The following example is invalid because `C.E` has the LIKE attribute.

```
declare 1 A like C,
        1 B,
          2 C,
            3 D,
            3 E like X,
          2 F,
        1 X,
          2 Y,
          2 Z;
```

The following example is invalid because the LIKE attribute of `A` specifies a substructure, `G.C`, of a structure, `G`, declared with the LIKE attribute.

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

The following example is invalid because the LIKE attribute of A specifies a structure, C, within a structure, B, that contains a substructure, F, having the LIKE attribute.

```
declare 1 A like C,
        1 B,
          2 C,
            3 D,
            3 E,
          2 F like X,
        1 X,
          2 Y,
          2 Z;
```

Combinations of arrays, structures, and unions

Specifying the dimension attribute on a structure or union results in an *array of structures* or an *array of unions*, respectively. The elements of such an array are structures or unions having identical names, levels, and members. For example, if a structure were used to hold meteorological data for each month of the year for the twentieth and the twenty-first centuries, it might be declared as follows:

```
Declare 1 Year(1901:2100),
          3 Month(12),
            5 Temperature,
              7 High decimal fixed(4,1),
              7 Low decimal fixed(4,1),
            5 Wind_velocity,
              7 High decimal fixed(3),
              7 Low decimal fixed(3),
           5 Precipitation,
              7 Total decimal fixed(3,1),
              7 Average decimal fixed(3,1),
         3 * char(0);
```

You could refer to the weather data for July 1991 by specifying `Year(1991,7)`. Portions of this data could be referred to by `Temperature(1991,7)` and `Wind_Velocity(1991,7)`. `Precipitation.Total(1991,7)` or `Total(1991,7)` would both refer to the total precipitation during July 1991.

`Temperature.High(1991,3)`, which would refer to the high temperature in March 1991, is a subscripted qualified reference.

The need for subscripted qualified references becomes apparent when an array of structures or unions contains members that are arrays. In the following example, both A and B are structures:

```
declare 1 A (2,2),
          (2 B (2),
             3 C,
             3 D,
           2 E) fixed bin;
```

To refer to a data item, it might be necessary to use as many as three names and three subscripts. For example:

```
A(1,1).B        refers to B, an array of structures.
A(1,1)          refers to a structure.
A(1,1).B(1)     refers to a structure.
A(1,1).B(2).C   refers to an element.
```

As long as the order of subscripts remains unchanged, subscripts in such references can be moved to names at a lower or higher level. In the previous example, `A.B.C(1,1,2)` and `A(1,1,2).B.C` have the same meaning as `A(1,1).B(2).C` for the above array of structures. Unless all of the subscripts are moved to the lowest level, the reference is said to have *interleaved subscripts*, so `A.B(1,1,2).C` has interleaved subscripts.

Any item declared within an array of structures or unions inherits dimensions declared in the parent. In the previous declaration for the array of structures `A`, the array `B` is a three-dimensional structure, because it inherits the two dimensions declared for `A`. If `B` is unique and requires no qualification, any reference to a particular `B` would require three subscripts, two to identify the specific `A` and one to identify the specific `B` within that `A`.

Cross sections of arrays of structures or unions

A reference to a cross section of an array of structures or unions is not allowed. That is, the asterisk notation cannot be used in a reference unless all of the subscripts are asterisks.

Structure and union operations

Structures can be referenced in most contexts that any elementary variable can be referenced. For example, you can have structure references in assignments, I/O statements, and so on. References to unions or structures that contain unions, however, are limited to the following:
- Parameters and arguments
- Storage control and those built-in functions and subroutines that allow structures.

Structure and union mapping

Individual members of a union are mapped the same way as members of the structure. That is, each of the members, if not a union, is mapped as if it were a member of a structure. This means that the first storage locations for each of the members of a union do not overlay each other if each of the members requires different alignment and therefore different padding before the beginning of the member.

Consider the following union:

```
dcl
   1 A union,
     2 B,
       3 C char(1),
       3 D fixed bin(31),
     2 E,
       3 F char(2),
       3 G fixed bin(31);
```

Three bytes of padding are added between `A` and `B`. Two bytes are added between `A` and `E`.

Structure and union mapping

In order to ensure that the first storage location of each of the members of a union is the same, make sure that the first member of each has the same alignment requirement and it is the same as the highest alignment of any of its members (or its member's members).

The remainder of the discussion applies to members of a structure or union, which can be minor structures or elementary variables.

For any major or minor structure, the length, alignment requirement, and position relative to an 8-byte boundary depend on the lengths, alignment requirements, and relative positions of its members. The process of determining these requirements for each level and for the complete structure is known as *structure mapping*.

You can use structure mapping for determining the record length required for a structure when record-oriented input/output is used, and determining the amount of padding or rearrangement required for correct alignment of a structure for locate-mode input/output.

The structure mapping process minimizes the amount of unused storage (padding) between members of the structure. It completes the entire process before the structure is allocated, according (in effect) to the rules discussed in the following paragraphs.

Structure mapping is not a physical process. Terms such as *shifted* and *offset* are used purely for ease of discussion, and do not imply actual movement in storage. When the structure is allocated, the relative locations are already known as a result of the mapping process.

The mapping for a complete structure reduces to successively combining pairs of items (elements, or minor structures whose individual mappings have already been determined). Once a pair has been combined, it becomes a unit to be paired with another unit, and so on until the complete structure is mapped. The rules for the process are categorized as follows:
- Rules for determining the order of pairing
- Rules for mapping one pair.

These rules are described below, and an example shows an application of the rules in detail. It is necessary to understand the difference between the *logical level* and the *level-number* of structure elements. The logical levels are immediately apparent if the structure declaration is written with consistent level-numbers or suitable indentation (as in the detailed example given after the rules). In any case, you can determine the logical level of each item in the structure by applying the following rule to each item in turn, starting at the beginning of the structure declaration:

Note: The logical level of a given item is always one unit deeper than that of its immediate containing structure.

In the following example, the lower line shows the logical level for each item in the declaration.

```
dcl 1 A, 4 B, 5 C, 5 D, 3 E, 8 F, 7 G;

    1   2   3   3   2   3   3
```

Rules for order of pairing
The steps in determining the order of pairing are as follows:

1. Find the minor structure at the deepest logical level (which we will call logical level n).
2. If more than one minor structure has the logical level n, take the first one that appears in the declaration.
3. Pair the first two elements appearing in this minor structure, thus forming a unit. Use the rules for mapping one pair. (See "Rules for mapping one pair.")
4. Pair this unit with the next element (if any) declared in the minor structure, thus forming a larger unit.
5. Repeat step 4 until all the elements in the minor structure have been combined into one unit. This completes the mapping for this minor structure. its alignment requirement and length, including any padding, are now determined and will not change (unless you change the structure declaration). Its offset from a doubleword boundary is also now determined. This offset is significant during mapping of any containing structure, and it can change as a result of such mapping.
6. Repeat steps 3 through 5 for the next minor structure (if any) appearing at logical level n in the declaration.
7. Repeat step 6 until all minor structures at logical level n have been mapped. Each of these minor structures can now be thought of as an element for structure mapping purposes.
8. Repeat the pairing process for minor structures at the next higher logical level; that is, make n equal to (n-1) and repeat steps 2 through 7.
9. Repeat step 8 until n = 1; then repeat steps 3 through 5 for the major structure.

Rules for mapping one pair

For purposes of this explanation, think of storage as contiguous doublewords, each having 8 bytes, numbered 0 through 7, which indicate the offset from a doubleword boundary. Think of the bytes as numbered continuously from 0 onward, starting at any byte, so that lengths and offsets from the start of the structure can be calculated.

1. Begin the first element of the pair on a doubleword boundary; or, if the element is a minor structure that has already been mapped, offset it from the doubleword boundary by the amount indicated.
2. Begin the second element of the pair at the first valid position following the end of the first element. This position depends on the alignment requirement of the second element. (If the second element is a minor structure, its alignment requirement will have already been determined.)
3. Shift the first element towards the second element as far as the alignment requirement of the first allows. The amount of shift determines the offset of this pair from a doubleword boundary.

After this process has been completed, any padding between the two elements has been minimized and does not change throughout the rest of the operation. The pair is now a unit of fixed length and alignment requirement; its length is the sum of the two lengths plus padding, and its alignment requirement is the higher of the two alignment requirements (if they differ).

Effect of UNALIGNED attribute

The example of structure mapping given below shows the rules applied to a structure declared ALIGNED. Mapping of aligned structures is more complex because of the number of alignment requirements. The effect of the UNALIGNED attribute is to reduce to one byte the alignment requirements for halfwords, fullwords, and doublewords, and to reduce to one bit the alignment requirement for

bit strings. The same structure mapping rules apply, but the reduced alignment requirements are used. The only unused storage will be bit padding within a byte when the structure contains bit strings.

AREA data cannot be unaligned.

If a structure has the UNALIGNED attribute and it contains an element that cannot be unaligned, UNALIGNED is ignored for that element. The element is aligned and an error message is produced. For example, in a program with the following declaration, `C` is given the attribute ALIGNED because the inherited attribute UNALIGNED conflicts with AREA.

```
declare 1 A unaligned,
          2 B,
          2 C area(100);
```

Example of structure mapping

The following example shows the application of the structure mapping rules for a structure with the specified declaration.

```
declare 1 A aligned,
          2 B fixed bin(31),
          2 C,
            3 D float decimal(14),
            3 E,
              4 F entry,
              4 G,
                5 H character(2),
                5 I float decimal(13),
              4 J fixed binary(31,0),
            3 K character(2),
            3 L fixed binary(20,0),
          2 M,
            3 N,
              4 P fixed binary(15),
              4 Q character(5),
              4 R float decimal(2),
            3 S,
              4 T float decimal(15),
              4 U bit(3),
              4 V char(1),
            3 W fixed bin(31),
          2 X picture '$9V99';
```

The minor structure at the deepest logical level is `G`, so this is mapped first. Then `E` is mapped, followed by `N`, `S`, `C`, and `M`, in that order.

For each minor structure, a table in Figure 8 on page 181 shows the steps in the process, and a diagram in Figure 9 on page 182 shows a visual interpretation of the process. Finally, the major structure `A` is mapped as shown in Figure 15 on page 184. At the end of the example, the structure map for `A` is set out in the form of a table (Figure 16 on page 185) showing the offset of each member from the start of `A`.

	Name of Element	Alignment Requirement	Length	Offset from Doubleword Begin	End	Length of Padding	Offset from Minor Structure
Step 1	H	Byte	2	0	1		
	I	Doubleword	8	0	7		
Step 2	*H	Byte	2	6	7		0
	I	Doubleword	8	0	7	0	2
Minor Structure	G	Doubleword	10	6	7		
Step 1	F	Fullword	8	0	7		
	G	Doubleword	10	6	7		
Step 2	*F	Fullword	8	4	3		0
	G	Doubleword	10	6	7	2	10
Step 3	F & G	Doubleword	20	4	7		
	J	Fullword	4	0	3	0	20
Minor Structure	E	Doubleword	24	4	3		
Step 1	P	Halfword	2	0	1		0
	Q	Byte	5	2	6		2
Step 2	P & Q	Halfword	7	0	6		
	R	Fullword	4	0	3	1	8
Minor Structure	N	Fullword	12	0	3		
Step 1	T	Doubleword	8	0	7		0
	U	Byte	1	0	0	0	8
Step 2	T & U	Doubleword	9	0	0		
	V	Byte	1	1	1	0	9
Minor Structure	S	Doubleword	10	0	1		
Step 1	D	Doubleword	8	0	7		0
	E	Doubleword	24	4	3	4	12
Step 2	D . E	Doubleword	36	0	3		
	K	Byte	2	4	5	0	36
Step 3	D, E, & K	Doubleword	38	0	5		
	L	Fullword	4	0	3	2	40
Minor Structure	C	Doubleword	44	0	3		
Step 1	N	Fullword	12	0	3		
	S	Doubleword	10	0	1		
Step 2	*N	Fullword	12	4	7		0
	S	Doubleword	10	0	1	0	12
Step 3	N & S	Doubleword	22	4	1		
	W	Fullword	4	4	7	2	24
Minor Structure	M	Doubleword	28	4	7		

*First item shifted right

Figure 8. Mapping of example structure

Structure mapping example
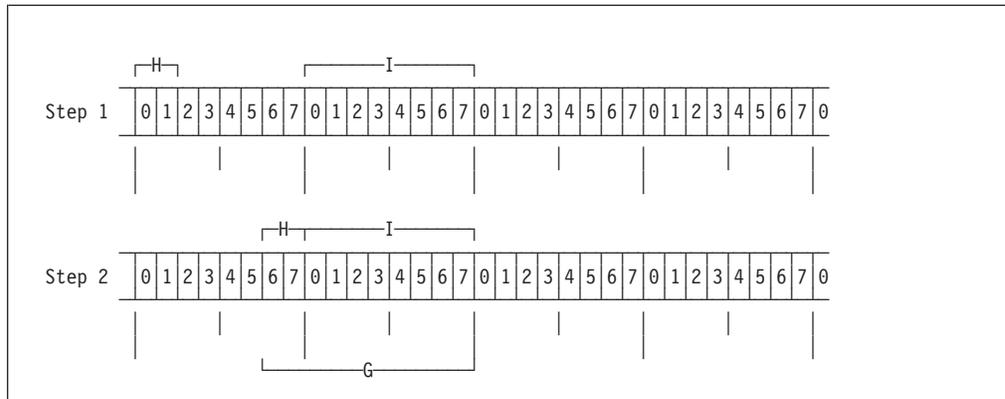
Figure 9. Mapping of minor structure G
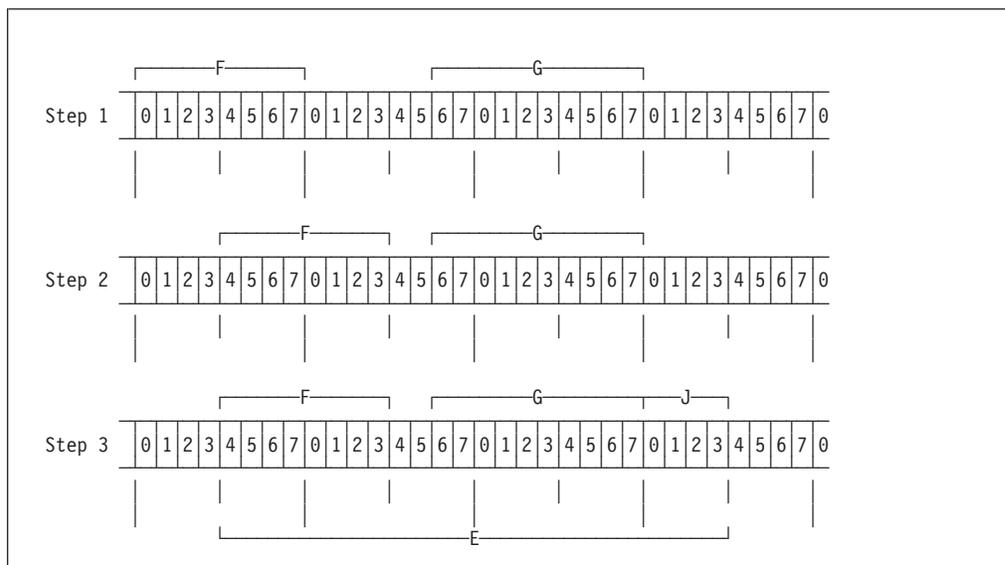
Figure 10. Mapping of minor structure E
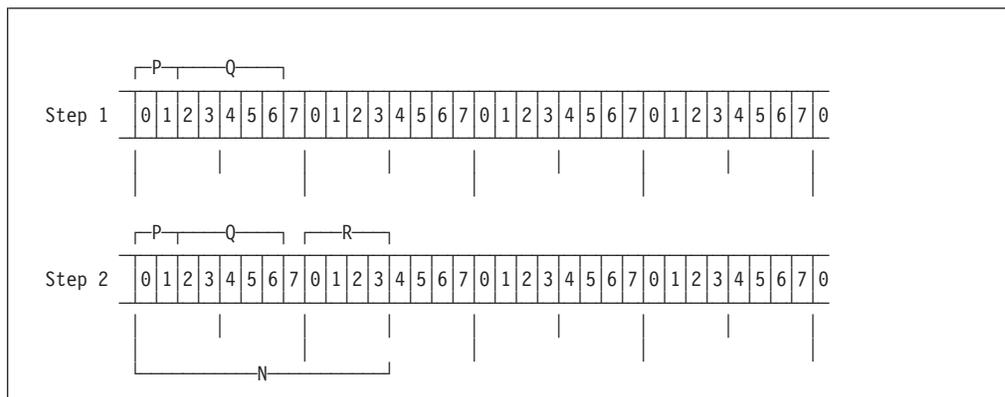
Figure 11. Mapping of minor structure N
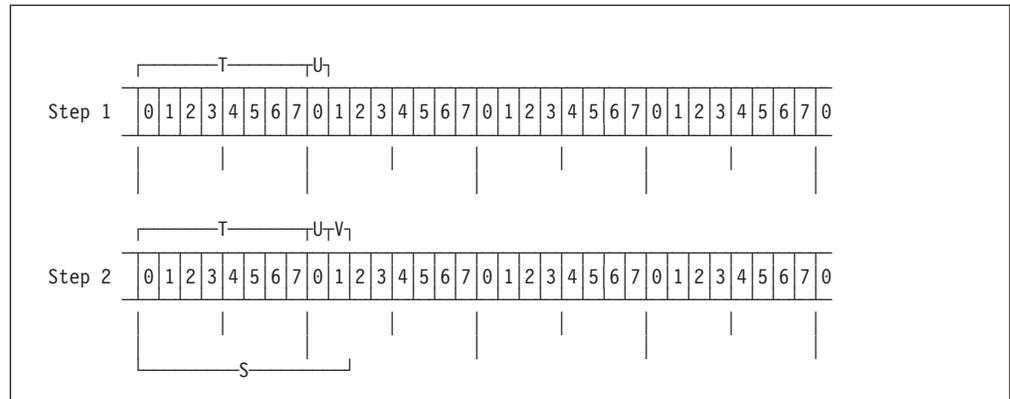
Figure 12. Mapping of minor structure S
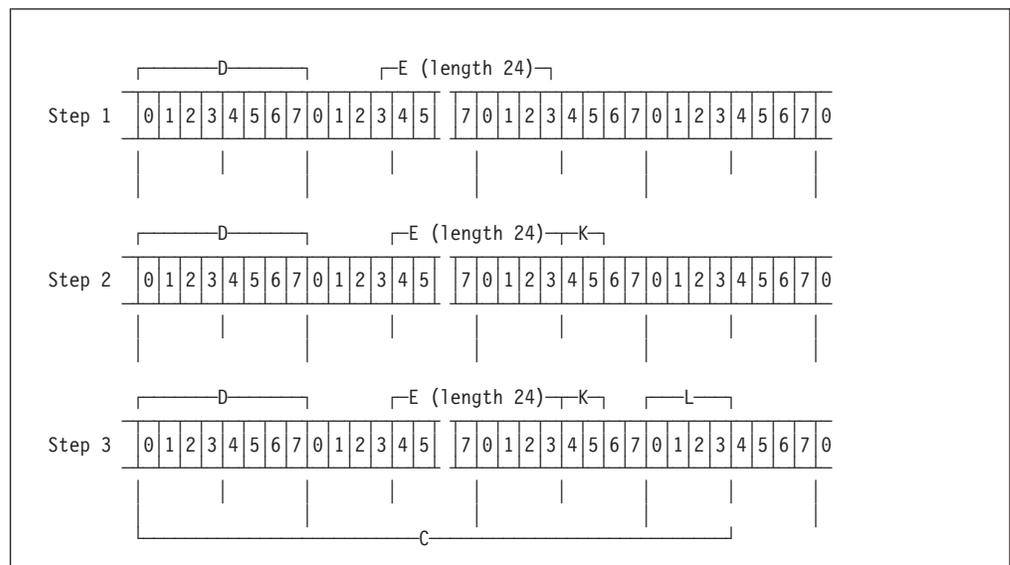
Figure 13. Mapping of minor structure C
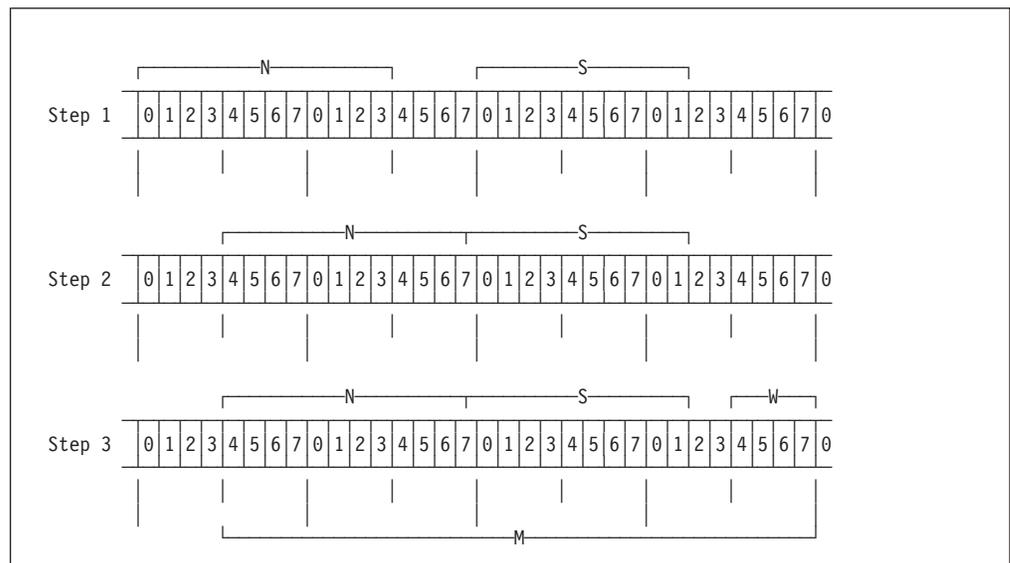
Figure 14. Mapping of minor structure M

Structure mapping example

	Name of Item	Alignment Required	Length	Offset from Doubleword		Length of Padding	Offset from A
				Begin	End		
Step 1	B	Fullword	4	0	3		
	C	Doubleword	44	0	3		
Step 2	B*	Fullword	4	4	7		0
	C	Doubleword	44	0	3	0	4
Step 3	B & C	Doubleword	48	4	3		
	M	Doubleword	28	4	7	0	48
Step 4	B, C, & M	Doubleword	76	4	7		
	X	Byte	4	0	3	0	76
	A	Doubleword	80	4	3		

* First item shifted right
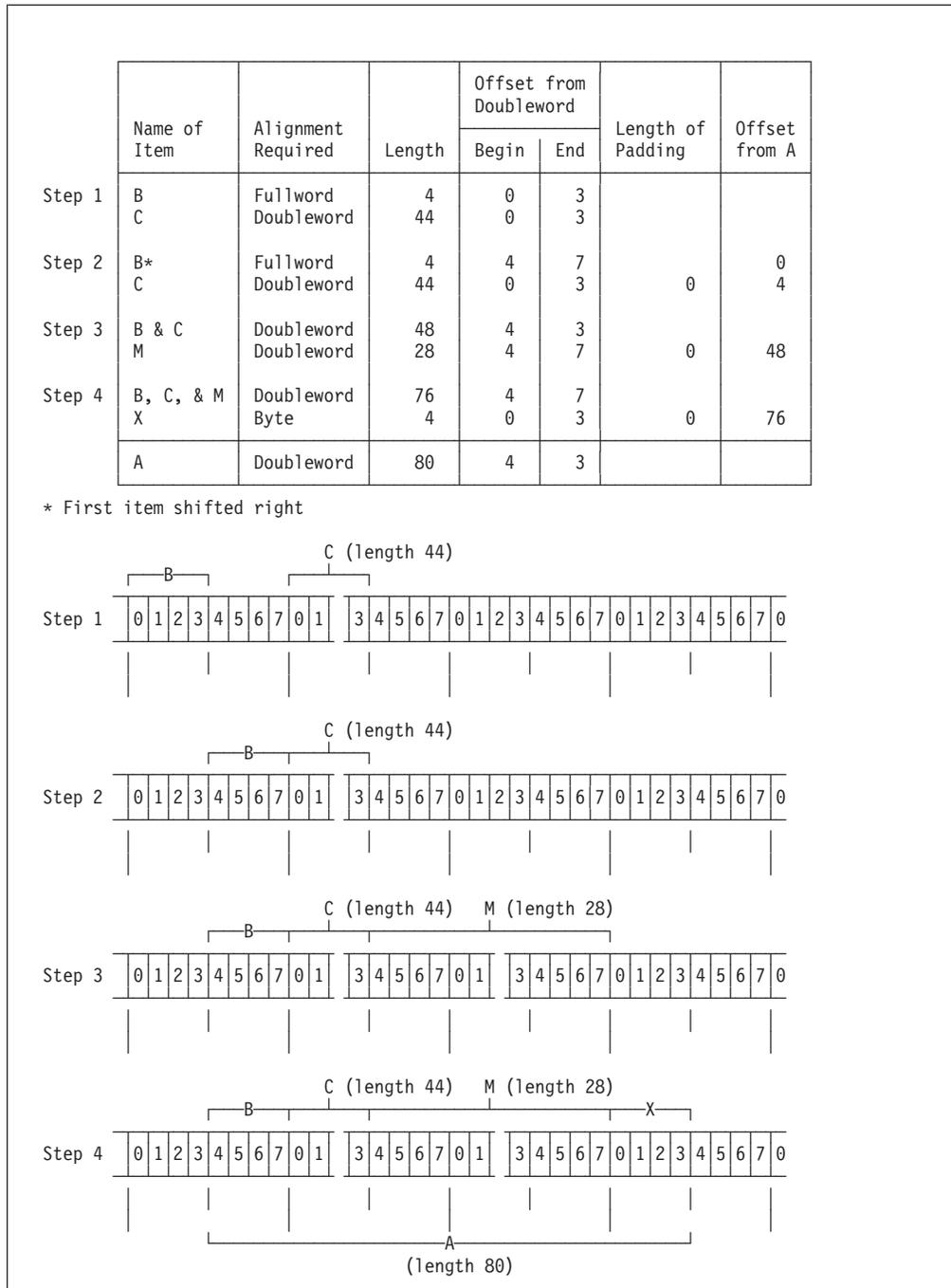
Figure 15. Mapping of major structure A

```
A                                                          From A
  B                                                          0
  C                                         From C           4
    D                                          0             4
    padding (4)                                8            12
    E                           From E        12            16
      F                            0          12            16
      padding (2)                  8          20            24
      G           From G          10          22            26
        H            0            10          22            26
        I            2            12          24            28
      J                          20          32            36
    K                                         36            40
    padding (2)                              38            42
    L                                        40            44
  M                                        From M          48
    N                           From N        0            48
      P                            0          0            48
      Q                            2          2            50
      padding (1)                  7          7            55
      R                            8          8            56
    S                           From S       12            60
      T                            0          12            60
      U                            8          20            68
      V                            9          21            69
    padding (2)                              22            70
    W                                        24            72
  X                                                        76
```

Figure 16. Offsets in final mapping of structure A

Chapter 9. Statements and directives

This chapter lists all of the PL/I statements and %directives. %Statements and macro statements are described in chapters Chapter 21, "Preprocessor Facilities," on page 499.

ALLOCATE statement

The ALLOCATE statement is described in Chapter 10, "Storage control," on page 221

Assignment and compound assignment statements

The assignment statement evaluates an expression and assigns its value to one or more target variables.

These statements are used for internal data movement, as well as for specifying computations. (The GET and PUT statements with the STRING option can also be used for internal data movement. Additionally, the PUT statement can specify computations to be done. See Chapter 13, "Stream-oriented data transmission.")

Because the attributes of the target variable or pseudovariable can differ from the attributes of the source (a variable, a constant, or the result of an expression), the assignment statement might require conversions (see Chapter 5, "Data conversion").

Assignment statement

Use the following syntax for the assignment statement.

```
      ┌──,──────┐
►►─────▼─reference─┴───── = ──expression────────────;──────────────►◄
                                      └─,BY NAME─┘
```
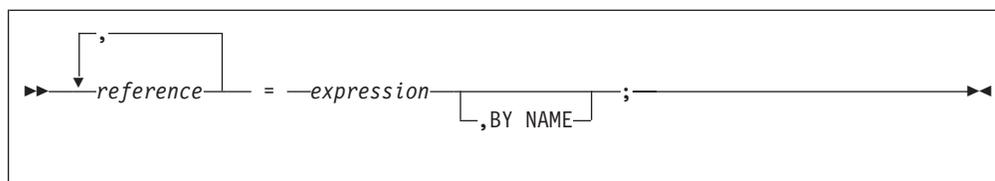
reference
    Specifies the target to be given the assignment

expression
    Specifies an expression that is evaluated and possibly converted

BY NAME
    For structure assignments, the BY NAME option specifies the assignment
    follows the steps outlined on 192.

Compound assignment statement

Use the following syntax for the compound assignment statement.

```
      ┌──,──────┐
►►─────▼─reference─┴──── compound assignment operator ──expression──;───────►◄
```
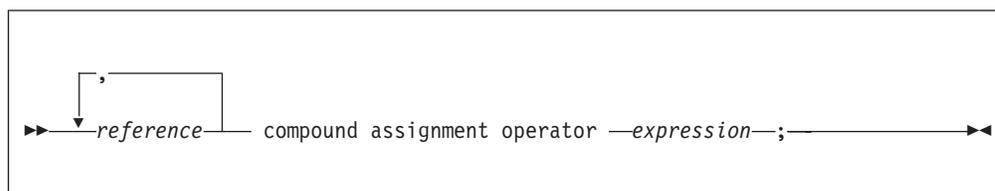
reference
    Specifies the target to be given the assignment

compound assignment operator
    Specifies the operator that is applied to the reference and the evaluated
    expression before the assignment is made. Table 30 lists the compound
    assignment operators allowed in compound assignments.

expression
    Specifies an expression that is evaluated and possibly converted.

Area assignment is described in "Area data and attribute" on page 237.

Table 30. Compound assignment operators

Compound assignment operator	Meaning
+=	Evaluate expression, add and assign
−=	Evaluate expression, subtract and assign
*=	Evaluate expression, multiply and assign
/=	Evaluate expression, divide and assign
|=	Evaluate expression, or and assign
&=	Evaluate expression, and and assign
||=	Evaluate expression, concatenate and assign
**=	Evaluate expression, exponentiate and assign
¬=	Evaluate expression, exclusive-or and assign

The operator is applied to the target and source first, and then what results is assigned to the target.

For example:

```
X += 1                is the same as                X = X+(1)
X *= Y+Z              is the same as                X = X*(Y+Z)
```

but

```
X *= Y+Z              is NOT equivalent to          X = X*Y+Z
```

In a compound assignment, any subscripts or locator expressions specified in the target variable are evaluated only once.

If `f` is a function and `X` is an array, then:

```
X(f()) += 1           is NOT equivalent to          X(f()) = X(f())+1
```

The function `f` is called only once.

The remaining text discusses the following topics: :
- The requirements for target variables
- How element and aggregate assignments are performed
- How BY NAME assignments are performed
- How multiple assignment are performed.

Examples of assignments begin in "Example of moving internal data" on page 193.

Target variables

The target variables can be element, array, or structure variables; or pseudovariables.

Array targets
For array assignments, each target variable must be an array of scalars or structures. The source must be a scalar or an expression with the same number of dimensions and the same bounds for all dimensions as for the target.

Union targets
Union assignments are not allowed.

Structure targets
In BY NAME structure assignments, each target variable must be a structure. The right-hand side must be a structure reference.

In structure assignments not using BY NAME, each target variable must be a structure. The right-hand side must be a scalar or a structure expression with the same structuring as the target structure:
- The structures must have the same minor structuring and the same number of contained elements and arrays.
- The positioning of the elements and arrays within the structure (and within the minor structures, if any) must be the same.
- Arrays in corresponding positions must have identical bounds.

In structure assignments not using BY NAME, the source may be the null bit string (''b) even if the target structure contains non-computational data. In this case, the assignment is performed as if

1. all of the target was filled with '00'x
2. all the numeric target fields were set to 0
3. all the nonvarying character, widechar and graphic fields were filled with blanks

How assignments are performed

Element assignments
Element assignments are performed as follows:

1. First to be evaluated are subscripts, POSITION attribute expressions, locator qualifications of the target variables, and the second and third arguments of SUBSTR pseudovariable references.
2. The expression on the right-hand side is then evaluated.
3. For each target variable (in left to right order), the expression is converted to the characteristics of the target variable according to the rules for data conversion. The converted value is then assigned to the target variable.

Aggregate assignments
Aggregate assignments (array and structure assignments) are expanded into a series of element assignments as follows:

1. The label prefix of the original statement is applied to a null statement preceding the other generated statements.
2. Array and structure assignments, when there are more than one, are done iteratively.
3. Any assignment statement can be generated by a previous array or structure assignment. The first target variable in an aggregate assignment is known as the master variable. (It can also be the first argument of a pseudovariable). If the master variable is an array, an array expansion is performed; otherwise, a structure expansion is performed.
4. If an aggregate assignment meets a certain set of conditions, it can be done as a whole instead of being expanded into a series of element assignments. Two conditions are if the arrays are not interleaved, or if the structures are contiguous and have the same format.

In array assignments, all array operands must have the same number of dimensions and identical bounds. The array assignment is expanded into a loop as follows:

```
  do J1 = lbound(Master-variable,1) to
        hbound(Master-variable,1);
  do J2 = lbound(Master-variable,2) to
        hbound(Master-variable,2);
:
  do Jn = lbound(Master-variable,N) to
        hbound(Master-variable,N);

  generated assignment statement

  end;
```

In this expansion, n is the number of dimensions of the master variable that are to participate in the assignment. In the generated assignment statement, all array operands are fully subscripted, using (from left to right) the dummy variables j1 to jn. If an array operand appears with no subscripts, it has only the subscripts j1 to

Aggregate assignments

jn. If cross-section notation is used, the asterisks are replaced by j1 to jn. If the original assignment statement has a condition prefix, the generated assignment statement is given this condition prefix.

If the generated assignment statement is a structure assignment, it is expanded as described next.

In structure assignments where the BY NAME option is not specified:
- None of the operands can be arrays, although they can be structures that contain arrays.
- All of the structure operands must have the same number, *k*, of immediately contained items.
- The assignment statement is replaced by *k* generated assignment statements.
  - The ith generated assignment statement is derived from the original assignment statement by replacing each structure operand by its ith contained item; such generated assignment statements can require further expansion.
  - All generated assignment statements are given the condition prefix of the original statement.

In structure assignments where the BY NAME option is given, the structure assignment is expanded according to the steps below, which can generate further array and structure assignments. None of the operands can be arrays.
1. The first item immediately contained in the master variable is considered.
2. If each structure operand and target variable has an immediately contained item with the same name, an assignment statement is generated as follows:
   a. The statement is derived by replacing each structure operand and target variable with its immediately contained item that has this name. If any structure contains no such name, no statement is generated.
   b. If the generated assignment is a structure or array-of-structures assignment, BY NAME is appended.
   c. All generated assignment statements are given the condition prefix of the original assignment statement.
   d. A target structure must not contain unions.
3. Step 2 is repeated for each of the items immediately contained in the master variable. The assignments are generated in the order of the items contained in the master variable.

Multiple assignments

Assignments can be made to multiple variables in a single assignment statement, for example:

```
A,X = B + C;
```

The value of B + C is assigned to both A and X. In general, it has the same effect as the following statements:

```
Temporary = B + C;
A = Temporary;
X = Temporary;
```

The source in the assignment statement must be a scalar or an array of scalars, and if the source is an array, all the targets must also be arrays. If the source is a

constant, it is assigned to each of the targets from left to right. If the source is not a constant, it is assigned to a temporary variable, which is then assigned to each of the targets from left to right.

The target can be any reference allowed in a simple assignment.

BY NAME is not allowed in multiple assignments.

Example of moving internal data

The following example of the assignment statement can be used for internal data movement. The value of the expression on the right of the assignment symbol is assigned to the variable on the left.

```
NTOT=TOT;
```

Example of assigning expression values

The following example includes an expression whose value is to be assigned to the variable on the left of the assignment symbol:

```
Av=(Av*Num+Tav*Tnum)/(Num+Tnum);
```

Example of assigning a structure using BY NAME

The following example illustrates structure assignment using the BY NAME option:

```
declare        declare        declare
1 One,         1 Two,         1 Three,
 2 Part1,       2 Part1,       2 Part1,
  3 Red,         3 Blue,        3 Red,
  3 Orange,      3 Green,       3 Blue,
 2 Part2,        3 Red,         3 Brown,
  3 Yellow,     2 Part2,       2 Part2,
  3 Blue,        3 Brown,       3 Yellow,
  3 Green;       3 Yellow;      3 Green;
```

1
2
```
        One = Two, by name;
        One.Part1 = Three.Part1, by name;
```

1   The first assignment statement is the same as the following:

```
One.Part1.Red    = Two.Part1.Red;
One.Part2.Yellow = Two.Part2.Yellow;
```

2   The second assignment statement is the same as the following:

```
One.Part1.Red = Three.Part1.Red;
```

ATTACH statement

The ATTACH statement is described in Chapter 18, "Multithreading facility," on page 359.

BEGIN statement

The BEGIN statement is described in Chapter 6, "Program organization," on page 87.

CALL statement

The CALL statement is described in "CALL statement" on page 124.

CLOSE statement

The CLOSE statement is described in Chapter 11, "Input and output," on page 257.

DECLARE statement

The DECLARE statement is described in "DECLARE statement" on page 148.

DEFINE ALIAS statement

The DEFINE ALIAS statement is described in "User-defined types (aliases)" on page 135.

DEFINE ORDINAL statement

The DEFINE ORDINAL statement is described in "DEFINE ORDINAL statement" on page 136.

DEFINE STRUCTURE statement

The DEFINE STRUCTURE statement is described in "Defining typed structures and unions" on page 138.

DEFAULT statement

The DEFAULT statement is described in "DEFAULT statement" on page 163.

DELAY statement

The DELAY statement suspends the execution of the next statement in the application program for the specified period of time.

```
►►──DELAY──(expression)──;──────────────────────────────────►◄
```

expression
Specifies an expression that is evaluated and converted to FIXED BINARY(31,0). Execution is suspended for the number of milliseconds specified.

The maximum wait time is 23 hours and 59 minutes.

For example:
```
delay (20);
```

suspends execution for 20 milliseconds.
```
delay (10**3);
```

suspends execution for one second.
```
delay (10*10**3);
```

suspends execution for ten seconds.

DELETE statement

The DELETE statement is described in "DELETE statement" on page 274.

DETACH statement

The DETACH statement is described in Chapter 18, "Multithreading facility," on page 359.

DISPLAY statement

The DISPLAY statement displays a message on the user's screen and optionally requests the user to enter a response to the message.

```
►►──DISPLAY──(expression)──┬──────────────────────┬──;──────────►◄
                           └─REPLY──(char-ref)─────┘
```
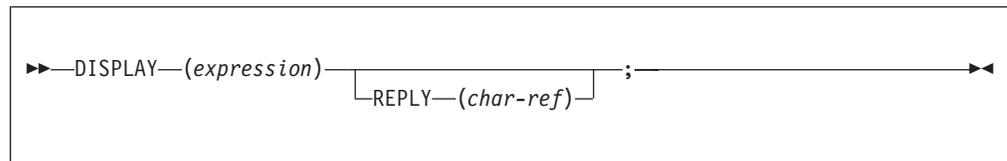
expression
Is converted, where necessary, to a character string. This character string is displayed. It can contain mixed character data. If the expression has the GRAPHIC attribute, it is not converted.

REPLY (char-ref)
Specifies a character reference that receives the user entered response. The response can contain CHARACTER, GRAPHIC, or mixed data.

The REPLY option suspends program execution until the user enters a response.

If GRAPHIC data is entered in the REPLY, it is received as character data that contains mixed data. Such character data can be converted to GRAPHIC data using the GRAPHIC BUILTIN.

Example:

```
display ('Communication link established.');
```

displays the message

```
Communication link established.
```

DO statement

The DO statement and its corresponding END statement, delimit a group of statements collectively called a do-group.

Note: Condition prefixes are invalid on DO statements.

```
┌──────────────────────────────────────────────────────────────────────┐
│ Type 1                                                                 │
│                                                                        │
│ ►►──DO──;──────────────────────────────────────────────────────────►◄ │
│                                                                        │
└──────────────────────────────────────────────────────────────────────┘
```
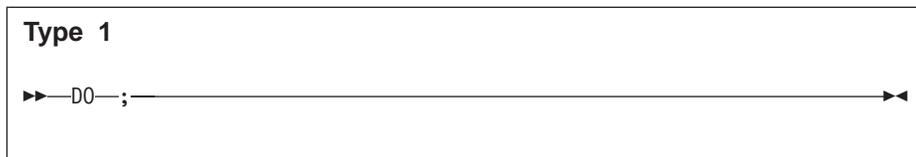
expn

An abbreviation for *expression n*.

Type 1

The type 1 do-group specifies that the statements in the group are executed. It does not provide for the repetitive execution of the statements within the group.

Types 2 and 3

Types 2 and 3 provide for the repetitive execution of the statements within the do-group.

WHILE (exp4)

Specifies that, before each repetition of do-group, *exp4* is evaluated and, if necessary, converted to a bit string. If any bit in the resulting string is 1, the do-group is executed. If all bits are 0, or the string is null, execution of the Type 2 do-group is terminated. For Type 3, only the execution associated with the specification containing the WHILE option is terminated. Execution for the next specification, if one exists, then begins.

UNTIL (exp5)

Specifies that, after each repetition of do-group, *exp5* is evaluated, and, if necessary, converted to a bit string. If all the bits in the resulting string are 0, or the string is null, the next iteration of the do-group is executed. If any bit is 1, execution of the Type 2 do-group is terminated. For Type 3, only the execution associated with the specification containing the UNTIL option is terminated. Execution for the next specification, if one exists, then begins.

reference

The only pseudovariables that can be used as references are SUBSTR, REAL, IMAG and UNSPEC. All data types are allowed.

The generation, g, of a reference is established once at the beginning of the do-group, immediately before the initial value expression (exp1) is evaluated. If the reference generation is changed to h in the do-group, the do-group continues to execute with the reference derived from the generation g. However, any reference to the reference inside the do-group is a reference to generation h. It is an error to free generation g in the do-group.

If a reference is made to a reference after the last iteration is completed, the value of the variable is the value that was out of range of the limit set in the specification. The preceding is true if the following conditions apply to the limit set in the application:
- The BY value is positive and the reference is greater than the TO value.
- The BY value is negative and the reference is less than the TO value.

If reference is a program-control data variable, but is not a locator, the BY and TO options cannot be used in specification.

If reference is a program-control variable, but is not a locator or an ordinal, the UPTHRU and DOWNTHRU options cannot be used in specification.

exp1
> Specifies the initial value of the reference.
>
> If TO, BY, and REPEAT are all omitted from a specification, there is a single execution of the do-group, with the reference having the value of exp1. If WHILE(exp4) is included, the single execution does not take place unless exp4 is true.

TO exp2

> exp2 is evaluated at entry to the specification and saved. This saved value specifies the terminating value of the reference. Execution of the statements in a do-group terminates for a specification as soon as the value of the reference, when tested at the beginning of the do-group, is out of range. Execution of the next specification, if one exists, then begins.
>
> If TO exp2 is omitted from a specification, and if BY exp3 is specified, repetitive execution continues until it is terminated by the WHILE or UNTIL option, or until another statement transfers control out of the do-group.

BY exp3

> exp3 is evaluated at entry to the specification and saved. This saved value specifies the increment to be added to the reference after each execution of the do-group.
>
> If BY exp3 is omitted from a specification, and if TO exp2 is specified, exp3 defaults to 1.
>
> If BY 0 is specified, the execution of the do-group continues indefinitely unless it is halted by a WHILE or UNTIL option, or control is transferred to a point outside the do-group.

UPTHRU exp2
> exp2 is evaluated at entry to the specification and saved. This saved value specifies the terminating value of the reference. Execution of the statements in a do-group terminates for a specification as soon as the value of the reference, when tested at the end of the do-group, is out of range. Execution of the next specification, if one exists, then begins.
>
> If UPTHRU is specified, the reference is compared to exp2 *after* the statements in the loop are executed, but before the reference is updated with the next value it can assume. The loop is executed at least once.
>
> UPTHRU is used primarily when processing ordinals using loops; however, it can also be used for a reference which is an arithmetic or locator control variable. If the reference is not an ordinal, the increment to be added to the reference after each execution of the do-group is assumed to be +1.

DOWNTHRU exp2

> exp2 is evaluated at entry to the specification and saved. This saved value specifies the terminating value of the reference. Execution of the statements in a do-group terminates for a specification as soon as the value of the reference, when tested at the end of the do-group, is out of range. Execution of the next specification, if one exists, then begins.
>
> If DOWNTHRU is specified, the reference is compared to exp2 *after* the statements in the loop are executed, but before the reference is updated with the next value it could assume. The loop is executed at least once.
>
> DOWNTHRU is used primarily when processing ordinals using loops; however, it can also be used for a reference which is an arithmetic or locator control

variable. If the reference is not an ordinal, the increment to be added to the reference after each execution of the do-group is assumed to be –1.

REPEAT exp6

exp6 is evaluated and assigned to the reference after each execution of the do-group. Repetitive execution continues until it is terminated by the WHILE or UNTIL option, or another statement transfers control out of the do-group.

Control can transfer into a do-group from outside the do-group only if the do-group is delimited by the DO statement in Type 1. Consequently, Type 2 and 3 do-groups cannot contain ENTRY statements. Control can also return to a do-group from a procedure or ON-unit invoked from within that do-group.

The following sections give more information about using Type 2 and Type 3 DO groups. Examples of DO groups begin in "Examples of basic repetitions" on page 202.

Using type 2 WHILE and UNTIL

If a Type 2 DO specification includes both the WHILE and UNTIL option, the DO statement provides for repetitive execution as defined by the following:

```
  Label:  do while (Exp4)
until (Exp5)
statement-1
 ⋮
statement-n
end;
  Next:    statement /* Statement following the do-group */
```

The above is equivalent to the following expansion:

```
  Label:  if (Exp4) then;
else
go to Next;
statement-1
 ⋮
statement-n
  Label2: if (Exp5) then;
else
go to Label;
  Next:    statement /* Statement following the do-group */
```

If the WHILE option is omitted, the IF statement at label Label is replaced by a null statement. Note that if the WHILE option is omitted, statements 1 through n are executed at least once.

If the UNTIL option is omitted, the IF statement at label Label2 in the expansion is replaced by the statement GO TO Label.

Using type 3 with one specification

The following sequence of events summarizes the effect of executing a do-group with one specification:

1. If reference is specified and BY, TO, UPTHRU, or DOWNTHRU options are also specified, exp1, exp2, and exp3 will be evaluated prior to the assignment of exp1 to the reference. Then the initial value is assigned to reference, for example:

   ```
     do Reference = Exp1 to Exp2 by Exp3;
   ```

   For a variable that is not a pseudovariable, the action of the do-group definition in the preceding example is equivalent to the following expansion:

```
        E1=Exp1;
        E2=Exp2;
        E3=Exp3;
        V=E1;
```

The variable `V` is a compiler-created based variable with the same attributes as the reference. `E1`, `E2`, and `E3` are compiler-created variables.

2. If the TO option is present, test the value of the control variable against the previously-evaluated expression (`E2`) in the TO option.
3. If the WHILE option is specified, evaluate the expression in the WHILE option. If it is *false*, leave the do-group.
4. Execute the statements in the do-group.
5. If the UNTIL option is specified, evaluate the expression in the UNTIL option. If it is *true*, leave the do-group.
6. If the UPTHRU option is specified, test the value of the control variable against the previously evaluated expression in the UPTHRU expression.
7. If the DOWNTHRU option is specified, test the value of the control variable against the previously evaluated expression in the DOWNTHRU expression.
8. If there is a reference:
   a. If the TO or BY option is specified, add the previously-evaluated *exp3* (`E3`) to the reference.
   b. If the REPEAT option is specified, evaluate the *exp6* and assign it to the reference.
   c. If the TO, BY, and REPEAT options are all absent, leave the do-group.
   d. If the UPTHRU option is specified and the reference is an ordinal, assign the reference the successor of its current value. Otherwise, add 1 to the reference.
   e. If the DOWNTHRU option is specified and the reference is an ordinal, assign it the predecessor of its current value. Otherwise, subtract 1 from the reference.
   f. If the TO, BY, UPTHRU, DOWNTHRU, and REPEAT options are all absent, leave the do-group.
9. Go to 2.

Using type 3 with two or more specifications

If the DO statement contains more than one specification, the second expansion is analogous to the first expansion in every respect. However, the statements in the do-group are not actually duplicated in the program. A succeeding specification is executed only after the preceding specification has been terminated.

When execution of the last specification terminates, control passes to the statement following the do-group.

Using type 3 with TO, BY, REPEAT

The TO and BY options let you vary the reference in fixed positive or negative increments. In contrast, the REPEAT option, which is an alternative to the TO and BY options, lets you vary the control variable nonlinearly. The REPEAT option can also be used for nonarithmetic control variables (such as pointer).

If the Type 3 DO specification includes the TO and BY options, the action of the do-group is defined by the following:

```
  Label:  do Variable=
Exp1
to Exp2
```

```
by Exp3
while (Exp4)
until(Exp5);
statement-1
  .
  .
  .
statement-m
  Label1:  end;
  Next:    statement
```

The action of the previous do-group definition is equivalent to the following expansion. In this expansion, `V` is a compiler-created variable with the same attributes as `Variable`; and `E1`, `E2`, and `E3` are compiler-created variables:

```
  Label:  E1=Exp1;
  E2=Exp2;
  E3=Exp3;
  V=E1;
  Label2: if (E3>=0)&(V>E2)|(E3<0)&(V<E2) then
go to Next;
if (Exp4) then;
else
 go to Next;
statement-1
  .
  .
  .
statement-m
  Label1: if (Exp5) then
go to Next;
  Label3: V=V+E3;
go to Label2;
  Next:    statement
```

If the specification includes the REPEAT option, the action of the do-group is defined by the following:

```
  Label:  do Variable=
Exp1
repeat Exp6
while (Exp4)
until(Exp5);
statement-1
  .
  .
  .
statement-m
  Label1:  end;
  Next:    statement
```

The action of the previous do-group definition is equivalent to the following expansion:

```
  Label:  E1=Exp1;
V=E1;
  Label2: ;
if (Exp4) then;
else
  go to Next;
statement-1
  .
  .
  .
statement-m
  Label1: if (Exp5) then
   go to Next;
  Label3: V=Exp6;
   go to Label2;
  Next:    statement
```

Additional rules for the sample expansions are as follows:

1. The previous expansion shows only the result of one specification. If the DO statement contains more than one specification, the statement labeled NEXT is the first statement in the expansion for the next specification. The second expansion is analogous to the first expansion in every respect. Statements 1 through m, however, are not actually duplicated in the program.

2. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in each of the expansions is also omitted.

3. If the UNTIL clause is omitted, the IF statement immediately following statement-m in each of the expansions is also omitted.

Using type 3 with UPTHRU

If the Type 3 DO specification includes the UPTHRU option, the action of the do-group is defined by the following:

```
  Label:  do Variable = Exp1 upthru Exp2 while (Exp4) until (Exp5);
statement₁
:
:
statementₙ
  Label1: end;
  Next: statement
```

The action of the previous do-group is equivalent to the following expansion. In this expansion, *V* is a compiler-generated variable with the same attributes as *Variable*; and *E1* and *E2* are compiler-generated variables:

```
  Label:  E1 = Exp1;
E2 = Exp2;
V = E1;
  Label2: if (Exp4) then;
else
go to next;
statement₁
:
:
statementₙ


  Label1: if (Exp5) then
go to Next;
if V ≥ E2 then
go to Next;
V = V + 1;
go to Label2;
  Next: statement
```

If the reference is an ordinal, the statement `V = V + 1` is replaced by `V = ordinalsucc(V)`.

Using type 3 with DOWNTHRU

If the Type 3 DO specification includes the DOWNTHRU option, the action of the do-group is defined by the following:

```
  Label:  do Variable = Exp1 downthru Exp2 while (Exp4) until (Exp5);
statement₁
:
:
statementₙ
  Label1: end;
  Next: statement
```

The action of the previous do-group is equivalent to the following expansion. In this expansion, *V* is a compiler-generated variable with the same attributes as *Variable*; and *E1* and *E2* are compiler-generated variables:

```
  Label:  E1 = Exp1;
E2 = Exp2;
V = E1;
  Label2: if (Exp4) then;
else
go to Next;
statement₁
:
statementₙ


  Label1: if (Exp5) then
go to Next;
if V ≤ E2 then
go to Next;
V = V − 1;
go to Label2;
  Next: statement
```

If the reference is an ordinal, the statement $V = V - 1$ is replaced by $V =$ ordinalpred(V).

Type 4

LOOP

Specifies infinite iteration. FOREVER is a synonym of LOOP.

For example:
```
dcl Payroll file;
dcl 1 Payrec,
     2 Type char,
     2 Subtype char,
     2 * char(100);

Readfile:
  do loop;

    read file(Payroll) into(Payrec);

    If Payrec.type = 'E'
      then leave; /* like goto After_ReadFile */

    If Payrec.type = '1' then
      do;
           /* process first part of record */

        If Payrec.subtype = 'S'
          then iterate Readfile; /* like goto End_ReadFile */

           /* process remainder of record */
      end;

End_ReadFile:
    end;
  After_ReadFile:;
```

The only way to exit this loop is by a LEAVE or GO TO, or by terminating a procedure or the program.

Examples of basic repetitions

In the following example, the do-group is executed ten times, while the value of the reference I progresses from 1 through 10.

```
  do I = 1 to 10;
:
  end;
```

The effect of this DO and END statement is equivalent to the following:

```
    I = 1;
  A: if I > 10 then go to B;
:
    I = I +1;
    go to A;
  B: next statement
```

The following DO statement executes the do-group three times—once for each assignment of `'Tom'`, `'Dick'`, and `'Harry'` to `Name`.

```
  do Name = 'Tom', 'Dick', 'Harry';
```

The following statement specifies that the do-group executes thirteen times—ten times with the value of `I` equal to 1 through 10, and three times with the value of `I` equal to 13 through 15:

```
  do I = 1 to 10, 13 to 15;
```

Repetition using the reference as a subscript

The reference of a DO statement can be used as a subscript in statements within the do-group, so that each execution deals with successive elements of a table or array.

In the following example, the first ten elements of `A` are set to 1 through 10 in sequence:

```
  do I = 1 to 10;
    A(I) = I;
  end;
```

Repetition with TO and BY

The following example specifies that the do-group is executed five times, with the value of `I` equal to 2, 4, 6, 8, and 10:

```
  do I = 2 to 10 by 2;
```

If negative increments of the reference are required, the BY option must be used. For example, the following is executed with `I` equal to 10, 8, 6, 4, 2, 0, and -2:

```
  do I = 10 to -2 by -2;
```

In the following example, the do-group is executed with `I` equal to 1, 3, 5:

```
  I=2;
  do I=1 to I+3 by I;
:
end;
```

The preceding example is equivalent to:

```
  do I=1 to 5 by 2;
:
end;
```

Example of DO with WHILE, UNTIL

The WHILE and UNTIL options make successive executions of the do-group dependent upon a specified condition, for example:

```
  do while (A=B);
    ⋮
end;
```

is equivalent to the following:

```
  S:  if A=B then;
else goto R;
    ⋮
goto S;
  R:  next statement
```

The example:

```
  do until (A=B);
    ⋮
end;
```

is equivalent to the following:

```
  S:
    ⋮
if (A=B) then goto R;
goto S;
  R:  next statement
```

In the absence of other options, a do-group headed by a DO UNTIL statement is executed at least once, but a do-group headed by a DO WHILE statement might not be executed at all. That is, the statements `DO WHILE (A=B)` and `DO UNTIL (A¬=B)` are not equivalent.

In the following example, if `A¬=B` when the DO statement is first encountered, the do-group is not executed at all.

```
  do while(A=B) until(X=10);
```

However, if `A=B`, the do-group is executed. If `X=10` after an execution of the do-group, no further executions are performed. Otherwise, a further execution is performed provided that `A` is still equal to `B`.

In the following example, the do-group is executed at least once, with `I` equal to 1:

```
  do I=1 to 10 until(Y=1);
```

If `Y=1` after an execution of the do-group, no further executions are performed. Otherwise, the default increment (BY 1) is added to `I`, and the new value of `I` is compared with 10. If `I` is greater than 10, no further executions are performed. Otherwise, a new execution commences.

The following statement specifies that the do-group executes ten times while `C(I)` is less than zero, and then (provided that `A` is equal to `B`) once more:

```
  do I = 1 to 10 while (C(I)<0),
        11 while (A = B);
```

Example of DO with UPTHRU and DOWNTHRU

In the following example, the do-group executes 5 times and at the end of the loop *i* has the value 5:

```
  do i = 1 upthru 5;
    ⋮
    end;
```

When the UPTHRU option is used, the reference is compared to the terminating value before being updated; this can be very useful when there is no value after the terminating value. For instance, the FIXEDOVERFLOW condition would not be raised by the following loop:

```
  do i = 2147483641 upthru 2147483647;
:
  end;
```

Similarly, the following loop avoids the problem of decrementing an unsigned value equal to zero:

```
  dcl U unsigned fixed bin;
  do U = 17 downthru 0;
:
  end;
```

UPTHRU and DOWNTHRU are particularly useful with ordinals. Consider the following example:

```
  define ordinal Color ( Red value (1),
Orange,
Yellow,
Green,
Blue,
Indigo,
Violet);
  dcl C ordinal Color;


  do C = Red upthru Violet;
:
  end;


  do C = Violet downthru Red;
:
  end;
```

In the first loop, *c* assumes each successive color in ascending order from red to violet. In the second loop, *c* assumes each successive color in descending order from violet to red.

Example of REPEAT

In the following example, the do-group is executed with I equal to 1, 2, 4, 8, 16, and so on:

```
  do I = 1 repeat 2*I;
:
end;
```

The preceding example is equivalent to:

```
    I=1;
  A:
:
I=2*I;
goto A;
```

In the following example, the first execution of the do-group is performed with I=1.

```
  do I=1 repeat 2*I until(I=256);
```

After this and each subsequent execution of the do-group, the UNTIL expression is tested. If `I=256`, no further executions are performed. Otherwise, the REPEAT expression is evaluated and assigned to `I`, and a new execution commences.

The following example shows a DO statement used to locate a specific item in a chained list:

```
do P=Phead repeat P -> Fwd
    while(P¬=null())
    until(P->Id=Id_to_be_found);
end;
```

The value `Phead` is assigned to `P` for the first execution of the do-group. Before each subsequent execution, the value `P -> Fwd` is assigned to `P`. The value of `P` is tested before the first and each subsequent execution of the do-group. If it is null, no further executions are performed.

The following statement specifies that the do-group is to be executed nine times, with the value of `I` equal to 1 through 9; then successively with the value of `I` equal to 10, 20, 40, and so on. Execution ceases when the do-group has been executed with a value of `I` greater than 10000.

```
do I = 1 to 9, 10 repeat 2*I
               until (I>10000);
⋮
end;
```

END statement

The END statement ends one or more blocks or groups. Every block or group must have an END statement.

```
►►──END─────────────────────;──────────────────►◄
        └─statement-label─┘
```
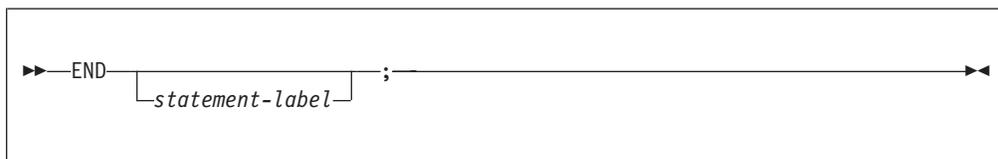
statement-label
Cannot be subscripted. If a statement-label follows END, the END statement closes the unclosed group or block headed by the nearest preceding DO, SELECT, PACKAGE, BEGIN, or PROCEDURE statement having that statement-label. Every block with a DO, SELECT, PACKAGE, BEGIN or PROCEDURE statement must have a corresponding END statement.

If a statement-label does not follow END, the END statement closes the one group or block headed by the nearest preceding DO, SELECT, PACKAGE, BEGIN, or PROCEDURE statement for which there is no other corresponding END statement.

Execution of a block terminates when control reaches the END statement for the block. However, it is not the only means of terminating a block's execution, even though each block must have an END statement. (See "Procedures" on page 92 and "Begin-blocks" on page 110 for more details.)

If control reaches an END statement for a procedure, it is treated as a RETURN statement.

Normal termination of a program occurs when control reaches the END statement of the main procedure.

ENTRY statement

The ENTRY statement is described in "ENTRY statement" on page 94.

EXIT statement

The EXIT statement stops the current thread.

```
►►──EXIT──;─────────────────────────────────────────────────►◄
```

FETCH statement

The FETCH statement is described in "FETCH statement" on page 102.

FLUSH statement

The FLUSH statement is described in Chapter 11, "Input and output," on page 257.

FORMAT statement

The FORMAT statement is described in Chapter 13, "Stream-oriented data transmission," on page 281.

FREE statement

The FREE statement is described in Chapter 10, "Storage control," on page 221.

GET statement

The GET statement is described in Chapter 13, "Stream-oriented data transmission," on page 281.

GO TO statement

The GO TO statement transfers control to the statement identified by the specified label reference. The GO TO statement is an unconditional branch.

```
►►──GO TO──label──;──────────────────────────────────────────►◄
```

Abbreviation: GOTO

label
> Specifies a label constant, a label variable, or a function reference that returns

a label value. Since a label variable can have different values at each execution of the GO TO statement, control might not always transfer to the same statement.

If a GO TO statement transfers control from within a block to a point not contained within that block, the block is terminated. If the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are also terminated (see "Procedure termination" on page 99).

When a GO TO statement specifies a label constant contained in a block that has more than one activation, control is transferred to the activation current when the GO TO is executed (see "Recursive procedures" on page 100).

A GO TO statement cannot transfer control:
- To an inactive block. Detection of such an error is not guaranteed.
- From outside a do-group to a statement inside a Type 2 or Type 3 do-group, unless the GO TO terminates a procedure or ON-unit invoked from within the do-group.
- To a FORMAT statement.

If the destination of the GO TO is specified by a label variable, it can then be used as a switch by assigning label constants to the label variable. If the label variable is subscripted, the switch can be controlled by varying the subscript. By using label variables or function references, quite complex switching can be effected. It is usually true, however, that simple control statements are the most efficient. GOTO operations from one block to another block hinder many optimizations in the target block, unless the target label is the last statement in its block.

IF statement

The IF statement evaluates an expression and controls the flow of execution according to the result of that evaluation. The IF statement thus provides a conditional branch.

Note: Condition prefixes are invalid on ELSE statements.

```
►►──IF──expression──THEN──unit1─────────────────────────►◄
                              └─ELSE──unit2─┘
```
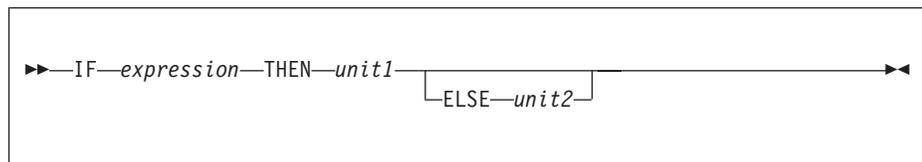
expression
> The expression must evaluate to a bit, not a bit string, unless RULES(LAXIF) is used.
>
> When expressions involve the use of the & and/or | operators, they are evaluated as described in "Combinations of operations" on page 67.

unit
> Either a valid single statement, a group, or a begin-block. All single statements are considered valid and executable except DECLARE, DEFAULT, END, ENTRY FORMAT, PROCEDURE, or a %statement. If a nonexecutable statement is used, the result can be unpredictable. Each unit can contain statements that specify a transfer of control (for example, GO TO). Hence, the normal sequence of the IF statement can be overridden.

Each unit can be labeled and can have condition prefixes.

IF is a compound statement. The semicolon terminating the last unit also terminates the IF statement.

If any bit in the string expression has the value '1'B, *unit1* is executed and *unit2*, if present, is ignored. If all bits are '0'B, or the string is null, *unit1* is ignored and *unit2*, if present, is executed.

IF statements can be nested. That is, either unit can itself be an IF statement, or both can be. Since each ELSE is always associated with the innermost unmatched IF in the same block or do-group, an ELSE with a null statement might be required to specify a desired sequence of control. For example, if `B` and `C` are constants, the following example:

```
if A = B then
  :
  :
else
if A = C then
  :
  :
else
  :
  :
```

is equivalent to and would be better coded as:

```
select( A );
when ( B )
  :
  :
when ( C )
  :
  :
  :
end;
```

Examples

In the following example, if the comparison is true (if `A` is equal to `B`), the value of `D` is assigned to `C`, and the ELSE unit is not executed.

```
  if A = B then
    C = D;
  else
    C = E;
```

If the comparison is false (`A` is not equal to `B`), the THEN unit is not executed, and the value of `E` is assigned to `C`.

Either the THEN unit or the ELSE unit can contain a statement that transfers control, either conditionally or unconditionally. If the THEN unit ends with a GO TO statement there is no need to specify an ELSE unit, for example:

```
  if all(Array1 = Array2) then
    go to LABEL_1;
  next-statement
```

If the expression is true, the GO TO statement of the THEN unit transfers control to `LABEL_1`. If the expression is not true, the THEN unit is not executed and control passes to the next statement.

%INCLUDE directive

The %INCLUDE directive is used to incorporate external text into the source
program.

```
►►──%INCLUDE──┬─member──────────┬──;──────────────►◄
              └─ddname (member)─┘
                    ↑─────,─────┐
```
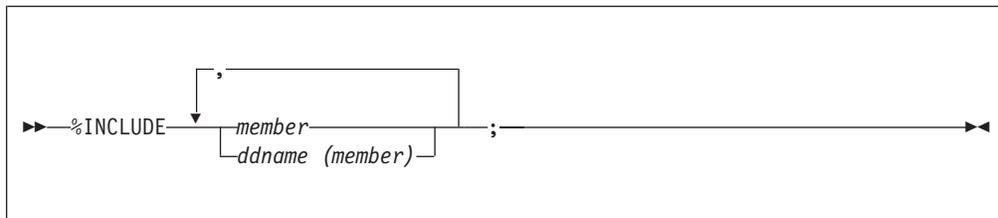
The included member can specify an absolute file name. Enclose the absolute file
name in single or double quotes. For example, the following is valid:

INTEL

```
%include "\ibmpli\include\sqlcodes.inc"
```

AIX and z/OS USS

```
%include "/ibmpli/include/sqlcodes.inc"
```

ITERATE statement

The ITERATE statement transfers control to the END statement that delimits its
containing iterative do-group. The current iteration completes and the next iteration,
if needed, is started. The ITERATE statement can be specified inside a non-iterative
do-group as long as that do-group is contained by an iterative do-group.
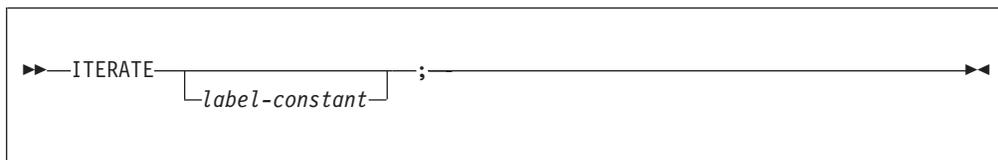
```
►►──ITERATE──┬────────────────┬──;──────────────────►◄
             └─label-constant─┘
```

label-constant
> Must be the label of a containing do-group. If omitted, control transfers to the
> END statement of the most recent iterative do-group containing the ITERATE
> statement.

For an example, see "Type 4" on page 202.

LEAVE statement

When contained in or specifying a simple do-group, the LEAVE statement
terminates the group. When contained in or specifying an iterative do-group, the
LEAVE statement terminates all iterations of the group, including the current
iteration. The flow of control goes to the same point it would normally go to if the
do-group had terminated by reaching its END statement. This point is not
necessarily the statement following the END statement of the do-group (see
"Example" on page 211).

```
►►──LEAVE──────────────────;──────────────────────────────►◄
          └─label-constant─┘
```

label-constant

> Must be a label of a containing do-group. The do-group that is left is the do-group that has the specified label. If *label-constant* is omitted, the do-group that is left is the group that contains the LEAVE statement.

The LEAVE statement and the referenced or implied DO statement must not be in different blocks.

In addition to the following examples, see the example in "Type 4" on page 202.

Example

In the following example, the statement `leave A;` transfers control to `C`.

```
  If Time_to_process_X then


A:  do I = lbound(X,1) to hbound(X,1);
       do J = lbound(X,2) to hbound(X,2);
          If X(I,J)=0 then
             leave A;          /* control goes to C, not B */
          else
             do;
.
.
.
             end;
       end;
    end;


   Else


 B:  do;
.
.
.
     end;


 C:  statement after group A;
```

%LINE directive

The %LINE directive specifies that the next line should be treated in messages and in information generated for debugging as if it came from the specified line and file.

```
►►──%LINE──(─line-number,─file-specification─)──;──────────────►◄
```

The characters '%LINE' must be in columns 1 through 5 of the input line for the directive to be recognized (and conversely, any line starting with these five characters is treated as a %LINE directive). The `line-number` must be an integral value of seven digits or less and the `file-specification` must not be enclosed in quotes. Any characters specified after the semicolon are ignored.

LOCATE statement

The LOCATE statement is described in Chapter 12, "Record-oriented data transmission," on page 271.

%NOPRINT directive

The %NOPRINT directive causes printing of the source listings to be suspended until a %PRINT directive is encountered or until a %POP directive is encountered that restores the previous %PRINT directive.

```
►►──%NOPRINT──;──────────────────────────────────────────►◄
```

For an example of the %NOPRINT directive, refer to "%PUSH directive" on page 215.

%NOTE directive

The %NOTE directive generates a diagnostic message of specified text and severity.

```
►►──%NOTE──(──message──────────)──;──────────────────►◄
                     └─,code─┘
```

message
A character expression whose value is the required diagnostic message.

code
A fixed expression whose value indicates the severity of the message, as follows:

Code	Severity
0	I
4	W
8	E
12	S
16	U

If code is omitted, the default is 0.

If code has a value other than those listed above, a diagnostic message is produced; the resulting system action is undefined.

Generated messages are filed together with other messages. Whether or not a particular message is subsequently printed depends upon its severity level and the setting of the compiler FLAG option (as described in the Programming Guide).

Generated messages of severity U cause immediate termination of preprocessing and compilation. Generated messages of severity S, E, or W might cause termination of compilation, depending upon the setting of various compiler options.

null statement

The null statement causes no operation to be performed and does not modify sequential statement execution. It is often used to denote null action for THEN and ELSE clauses and for WHEN and OTHERWISE statements.

```
►►──;─────────────────────────────────────────────────────────►◄
```

ON statement

The ON statement is described in Chapter 16, "Condition handling," on page 331.

OPEN statement

The OPEN statement is described in Chapter 11, "Input and output," on page 257.

%OPTION directive

The %OPTION directive is used to specify one of a selected subset of compiler options for a segment of source code. The specified option is then in effect until:

- Another %OPTION directive specifies a complementary compiler option (thus overriding the first), or
- A saved (via a %PUSH directive) compiler option is restored via a %POP directive

```
►►──%OPTION──compiler-option──;───────────────────────────────►◄
```

compiler-option
Specifies the compiler option to be in effect

For the allowed compiler options, see the Programming Guide.

For an example of %OPTION, see "%PUSH directive" on page 215.

OTHERWISE statement

The OTHERWISE statement is described in "SELECT statement" on page 217.

PACKAGE statement

The PACKAGE statement is described in "Packages" on page 90.

%PAGE directive

The %PAGE directive allows you to start a new page in the compiler source listings.

```
►►──%PAGE──;─────────────────────────────────────────────────►◄
```

%POP directive

The %POP directive allows you to restore the status of the %PRINT, %NOPRINT, and %OPTION directives saved by the most recent %PUSH directive.

The most common use of the %PUSH and %POP directives is in included files and macros.

```
►►──%POP──;───────────────────────────────────────────────────►◄
```

For an example, see "%PUSH directive" on page 215.

%PRINT directive

The %PRINT directive causes printing of the source listings to be resumed.

```
►►──%PRINT──;─────────────────────────────────────────────────►◄
```

%PRINT is in effect, provided that the relevant compiler options are specified. For an example of the %PRINT directive, refer to "%PUSH directive" on page 215.

PROCEDURE statement

The PROCEDURE statement is described in "PROCEDURE and ENTRY statements" on page 93.

%PROCESS directive

The %PROCESS directive is used to override compiler options.

```
►►──%PROCESS─┬──────────────────┬──;────────────────────────────────►◄
             │  ┌──────────────┐ │
             └──compiler-option─┘
```
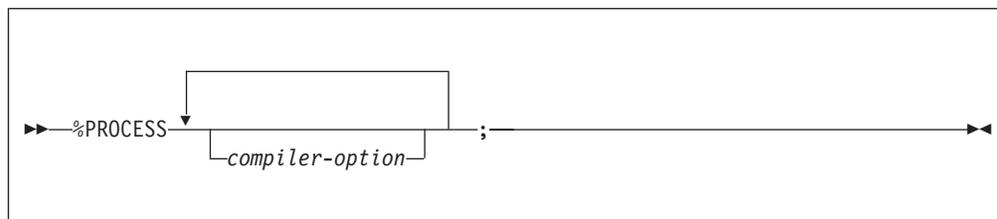
The % or * must be the first data byte of a source record. Any number of %PROCESS and *PROCESS directives can be specified, but they must all appear before the first language element appears. Refer to the Programming Guide for more information.

*PROCESS directive

The *PROCESS directive is a synonym for the %PROCESS directive. For information on the %PROCESS directive, refer to "%PROCESS directive."

%PUSH directive

The %PUSH directive allows you to save the current status of the %PRINT, %NOPRINT, and %OPTION directives in a "push down" stack on a last-in, first-out basis. You can restore this saved status later, also on a last-in, first-out basis, by using the %POP directive.

A common use of %PUSH and %POP directives is in included files and macros.

```
►►──%PUSH──;──────────────────────────────────────────────────────►◄
```
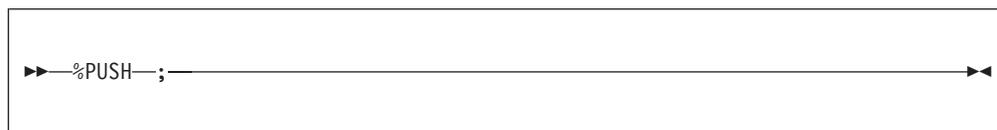
In the following example, statements 1, 2, 3, S4, S5, and 4 are printed in the listings. All others are not printed. Initially, LANGLVL(SAA) is in effect; then, LANGLVL(SAA2) is in effect for the entire included file Second.

```
             ┌──────────Source Program──────────┐
             │                                    │
             │ *process langlvl(saa);             │
             │                                    │
             │  statement 1;                      │
             │  statement 2;                      │
             │ %include First;   /* statement 3 */│
             │  ┌───────────First───────────┐     │
             │  │ %push;              /* F1 */│    │
             │  │ %noprint;           /* F2 */│    │
             │  │ dcl A entry (ptr,fixed bin); /* F3 */│
             │  │ statement F4;               │    │
             │  │ %include Second; /* stmt F5 */│  │
             │  │  ┌────────Second────────┐   │    │
             │  │  │ %push;         /* S1 */│  │    │
             │  │  │ %print;        /* S2 */│  │    │
             │  │  │ %option langlvl(saa2);   /* S3 */│
             │  │  │ dcl B  entry (ptr,fixed bin)│  │
             │  │  │    options(byvalue);   /* S4 */│
             │  │  │ statement S5;          │  │    │
             │  │  │ %pop;                  │  │    │
             │  │  └────────────────────────┘   │  │
             │  │                             │    │
             │  │ statement F6;               │    │
             │  │ %pop;                       │    │
             │  └─────────────────────────────┘    │
             │                                    │
             │  statement 4;                      │
             │                                    │
             └────────────────────────────────────┘
```
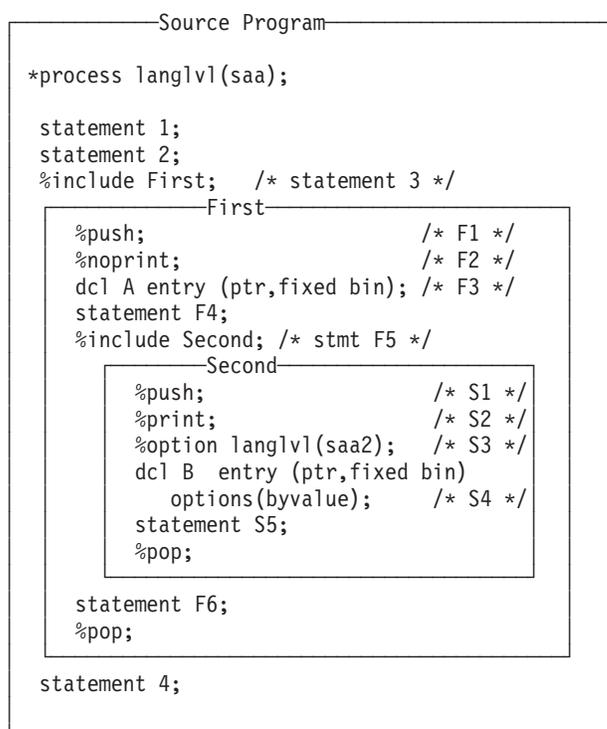
The original setting is restored following the %POP directive in Second.

PUT statement

The PUT statement is described in Chapter 13, "Stream-oriented data transmission," on page 281.

READ statement

The READ statement is described in Chapter 12, "Record-oriented data transmission," on page 271.

RELEASE statement

The RELEASE statement is described in "RELEASE statement" on page 103.

RESIGNAL statement

The RESIGNAL statement is described in "RESIGNAL statement" on page 337.

RETURN statement

The RETURN statement is described in "RETURN statement" on page 125.

REVERT statement

The REVERT statement is described in Chapter 16, "Condition handling," on page 331

REWRITE statement

The REWRITE statement is described in Chapter 12, "Record-oriented data transmission," on page 271.

SELECT statement

A select-group provides a multiple path conditional branch. A select-group contains a SELECT statement, optionally one or more WHEN statements, optionally an OTHERWISE statement, and an END statement.

Note: Condition prefixes are invalid on OTHERWISE statements.
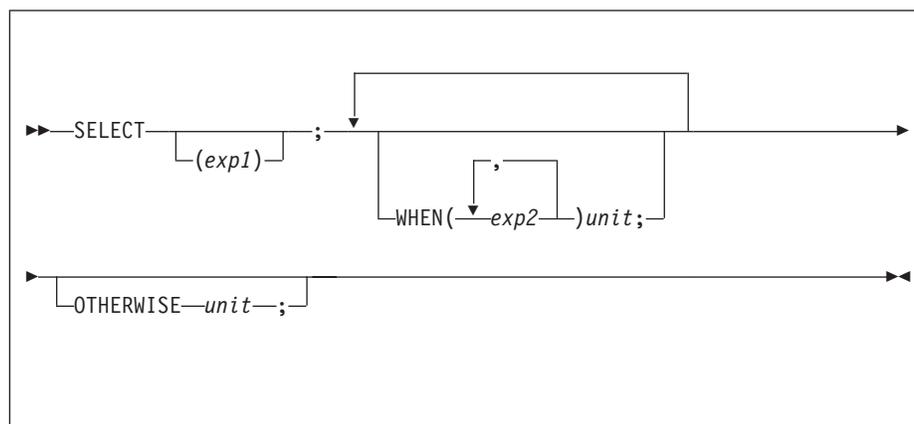
Abbreviation: OTHER for OTHERWISE

SELECT (exp1)

The SELECT statement and its corresponding END statement, delimit a group of statements collectively called a select-group. The expression in the SELECT statement is evaluated and its value is saved.

WHEN (exp2, exp2, ...) unit

Specifies one or more expressions that are evaluated and compared with the saved value from the SELECT statement.

If an expression is found that is equal to the saved value, the evaluation of expressions in WHEN statements is terminated, and the unit of the associated WHEN statement is executed. If no such expression is found, the unit of the OTHERWISE statement is executed.

The WHEN statement must not have a label.

OTHERWISE unit

Specifies the unit to be executed when every test of the preceding WHEN statements fails.

If the OTHERWISE statement is omitted and execution of the select-group does not result in the selection of a unit, the ERROR condition is raised.

The OTHERWISE statement must not have a label or condition prefix.

unit

Each unit is either a valid single statement, a group, or a begin-block. DECLARE, DEFAULT, END, ENTRY FORMAT, PROCEDURE, and %statement

statements are not valid. Each unit can contain statements that specify a transfer of control (for example, GO TO). Hence, the normal sequence of the SELECT statement can be overridden.

If *exp1* is omitted, each *exp2* is evaluated and converted, if necessary, to a bit string. If any bit in the resulting string is '1'B, the unit of the associated WHEN statement is executed. If all bits are 0 or the string is null, the unit of the OTHERWISE statement is executed.

After execution of a unit of a WHEN or OTHERWISE statement, control passes to the statement following the select-group, unless the normal flow of control is altered within the unit.

If *exp1* is specified, each *exp2* must be such that the following comparison expression has a scalar bit value:

```
(exp1) = (exp2)
```

Array, structure, and union operands cannot be used in either *exp1* or *exp2*.

Examples

In the following example, E, E1, and so on, are expressions. When control reaches the SELECT statement, the expression E is evaluated and its value is saved. The expressions in the WHEN statements are then evaluated in turn (in the order in which they appear), and each value is compared with the value of E.

If a value is found that is equal to the value of E, the action following the corresponding THEN statement is performed; no further WHEN statement expressions are evaluated.

If none of the expressions in the WHEN statements are equal to the expression in the SELECT statement, the action specified after the OTHERWISE statement is executed.

```
select (E);
  when (E1,E2,E3) action-1;
  when (E4,E5) action-2;
  otherwise action-n;
end;
Nl: next statement;
```

An example of *exp1* being omitted is:

```
select;
  when (A>B) call Bigger;
  when (A=B) call Same;
  otherwise call Smaller;
end;
```

If a select-group contains no WHEN statements, the action in the OTHERWISE statement is executed unconditionally. If the OTHERWISE statement is omitted, and execution of the select-group does not result in the selection of a WHEN statement, the ERROR condition is raised.

SIGNAL statement

The SIGNAL statement is described in Chapter 16, "Condition handling," on page 331.

%SKIP directive

The %SKIP directive causes the specified number of lines to be left blank in the compiler source listings.

```
►►──%SKIP────────────;────────────────────────────────►◄
           └─(n)─┘
```
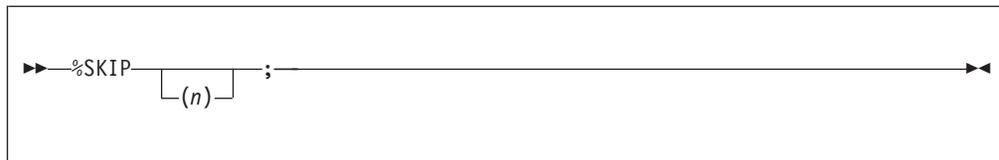
n   Specifies the number of lines to be skipped. It must be an integer in the range 1 through 999. If *n* is omitted, the default is 1. If *n* is greater than the number of lines remaining on the page, the equivalent of a %PAGE directive is executed in place of the %SKIP directive.

STOP statement

The STOP statement stops the current application.

```
►►──STOP──;────────────────────────────────────────────►◄
```
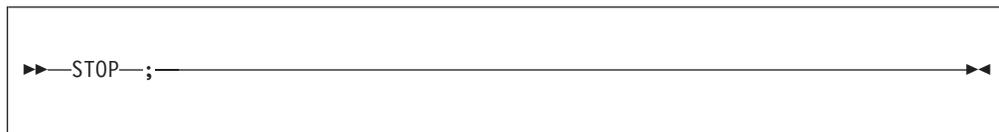
UNLOCK Statement

The UNLOCK statement makes the specified locked record available to other MVS tasks. The syntax for the UNLOCK statement is:

```
►►──UNLOCK──FILE──(──file-reference──)──KEY──(──expression──)──;──────────►◄
```
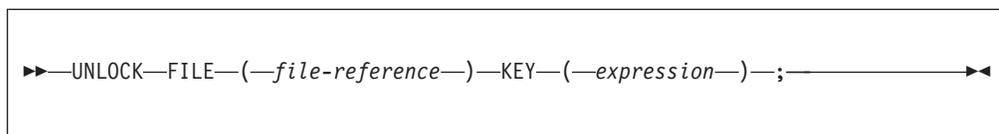
The keywords can appear in any order.

WAIT statement

WHEN statement

WRITE statement

Chapter 10. Storage control

All variables require storage. The attributes specified for a variable describe the amount of storage required and how it is interpreted. In the following example a reference to X is a reference to a piece of storage that contains a value to be interpreted as fixed-point binary.

```
dcl X fixed binary(31,0) automatic;
```

Since X is automatic, the storage for it is allocated when its declaring block is activated, and the storage remains allocated until the block is deactivated.

Storage classes, allocation, and deallocation

Storage allocation is the process of associating an area of storage with a variable so that the data item(s) represented by the variable can be recorded internally. When storage is associated with a variable, the variable is *allocated*.

Allocation for a given variable can take place *statically*, (before the execution of the program) or *dynamically* (during execution). A variable that is allocated statically remains allocated for the duration of the application program. A variable that is allocated dynamically relinquishes its storage either upon the termination of the block containing that variable, or at an explicit request from the application.

The storage class assigned to a variable determines the degree of storage control applied to it and the manner in which the variable's storage is allocated and freed. There are four storage classes: automatic, static, controlled, and based. You assign the storage class using its corresponding attribute in an explicit, implicit, or contextual declaration:

- AUTOMATIC specifies that storage is allocated upon each entry to the block that contains the storage declaration. The storage is released when the block is exited. If the block is a procedure that is invoked recursively, the previously allocated storage is pushed down upon entry; the latest allocation of storage is popped up in a recursive procedure when each generation terminates. (For a discussion of push-down and pop-up stacking, see "Recursive procedures" on page 100.)
- STATIC specifies that storage is allocated when the program is loaded. The storage is not freed until program execution is completed. The storage for a fetched procedure is not freed until the procedure is released.
- CONTROLLED specifies that you use the ALLOCATE and FREE statements to control the allocation and freeing of storage. Multiple allocations of the same controlled variable in the same program, without intervening freeing, stacks generations of the variable. You can access earlier generations only by freeing the later ones.
- BASED, like CONTROLLED, specifies that you control storage allocation and freeing. One difference is that multiple allocations are not stacked but are available at any time. Each allocation can be identified by the value of a pointer variable. Another difference is that based variables can be associated with an area of storage and identified by the value of an offset variable.

  Based variables outside of areas can be allocated and freed using the ALLOCATE built-in function and PLIFREE built-in subroutine respectively. They can also be allocated using the AUTOMATIC built-in function; such allocated variables are freed automatically when the block in which they are allocated terminates.

Storage class attributes can be declared explicitly for element, array, and major structure and union variables. For array and major structure and union variables, the storage class declared for the variable applies to all of the elements in the array or structure or union.

Storage class attributes cannot be specified for:
- CONDITION conditions
- Defined data items
- Entry constants
- File constants
- Format constants
- Identifiers defined in the DEFINE statement

- Label constants
- Members of structures and unions
- Named constants

Allocation of storage for variables is managed by PL/I. You do not specify where in storage the allocation is to be made. You can, however, specify that a variable be allocated in an existing AREA. For more information, refer to "Area data and attribute" on page 237.

Static storage and attribute

Variables declared with the STATIC attribute are allocated prior to running a program. They remain allocated until the program terminates. The program has no control over the allocation of static variables during execution.

```
►►──STATIC──────────────────────────────────────────────►◄
```

STATIC is the default for external variables, but internal variables can also be static. It is also the default for variables declared in a package, outside of any procedure. Static variables follow the normal scope rules for the validity of references to them. In the following example, the variable X is allocated for the life of the program, but it can be referenced only within procedure B or any block contained in B. The variable Y gets the STATIC attribute and is also allocated for the life of the program.

```
Package: Package exports (*);
  dcl Y char(10);

  A: proc options(main);
    B: proc;
      declare X static internal;
    end B;
  end A;

  C: proc;
    Y = 'hello';
  end C;

end Package;
```

If static variables are initialized using the INITIAL attribute, the initial values must be restricted expressions. Extent specifications must also be restricted expressions.

Automatic storage and attribute

Automatic variables are allocated on entry to the block in which they are declared. They can be reallocated many times during the execution of a program. You control their allocation by your design of the block structure.

```
►►──AUTOMATIC────────────────────────────────────────────►◄
```

Abbreviation: AUTO

Automatic storage and attribute

AUTOMATIC is the default. Automatic variables are always internal.

In the following example, each time procedure `B` is invoked, the variables `X` and `Y` are allocated storage. When `B` terminates, the storage is released, and the values `X` and `Y` contain are lost.

```
A: proc;
  :
call B;
B: proc;
declare X,Y auto;
  :
end B;
  :
call B;
```

The storage that is freed is available for allocation to other variables. Thus, whenever a block (procedure or begin) is active, storage is allocated for all variables declared automatic within that block. Whenever a block is inactive, no storage is allocated for the automatic variables in that block. Only one allocation of a particular automatic variable can exist, except for those procedures that are called recursively or by more than one program.

Extents for automatic variables can be specified as expressions. This means that you can allocate a specific amount of storage when you need it. In the following example, the character string `STR` has a length defined by the value of the variable `N` when procedure `B` is invoked.

```
A: proc;
declare N fixed bin;
  :
B: proc;
declare STR char(N);
```

If the declare statements are located in the same block, PL/I requires that the variable `N` be initialized either to a restricted expression or to an initialized static variable. In the following example, the length allocated is correct for `Str1`, but not for `Str2`. PL/I does not resolve this type of declaration dependency.

```
dcl N fixed bin (15) init(10),
M fixed bin (15) init(N),
Str1 char(N),
Str2 char(M);
```

Controlled storage and attribute

Variables declared as CONTROLLED are allocated only when you specify them in an ALLOCATE statement. A controlled variable remains allocated until a FREE statement that names the variable is encountered or until the end of the program.

Controlled variables are partially independent of the program block structure, but not completely. The scope of a controlled variable can be internal or external. When it is declared as internal, the scope of the variable is the block in which the variable is declared and any contained blocks. Any reference to a controlled variable that is not allocated is in error.

```
►►──CONTROLLED─────────────────────────────────────────────────►◄
```

Abbreviation: CTL

In the following example, the variable X can be validly referred to within procedure B and that part of procedure A that follows execution of the CALL statement.

```
A: proc;
dcl X controlled;
call B;
  .
  .
  .
B: proc;
allocate X;
  .
  .
  .
end B;
end A;
```

Generally, controlled variables are useful when a program requires large data aggregates with adjustable extents. Statements in the following example allocate the exact storage required depending on the input data and free the storage when it is no longer required.

```
  dcl A(M,N) ctl;
  get list(M,N);
  allocate A;
  get list(A);
  .
  .
  free A;
```

This method is more efficient than the alternative of setting up a begin-block, because block activation and termination are not required.

ALLOCATE statement for controlled variables

The ALLOCATE statement allocates storage for controlled variables, independent of procedure block boundaries. Controlled parameters can also be allocated. The bounds of controlled arrays, the lengths of controlled strings, and the size of controlled areas, as well as their initial values, can be specified in the ALLOCATE statement.

ALLOCATE for controlled variables

```
          ,
   ┌───────────────────────────────────────────┐
►►─ALLOCATE─┬──────────┬─┬─controlled-variable─┬─────────────────┬─;──►
           └──level───┘                       └──│ attribute │──┘

►─┬─ attribute: ──┬─dimension──────────────────────────────────────►◄
                  ├─CHARACTER(length)────────────┤
                  ├─BIT(length)──────────────────┤
                  ├─GRAPHIC(length)──────────────┤
                  ├─WIDECHAR(length)─────────────┤
                  ├─AREA(size)───────────────────┤
                  │          ,                   │
                  │       ┌─────┐                │
                  ├─INITIAL(─▼─item─┬─)──────────┤
                  └─INITIAL CALL─entry-reference──┘
                                        ,
                                     ┌─────────┐
                              └─(─▼─argument─┬─)─┘
```
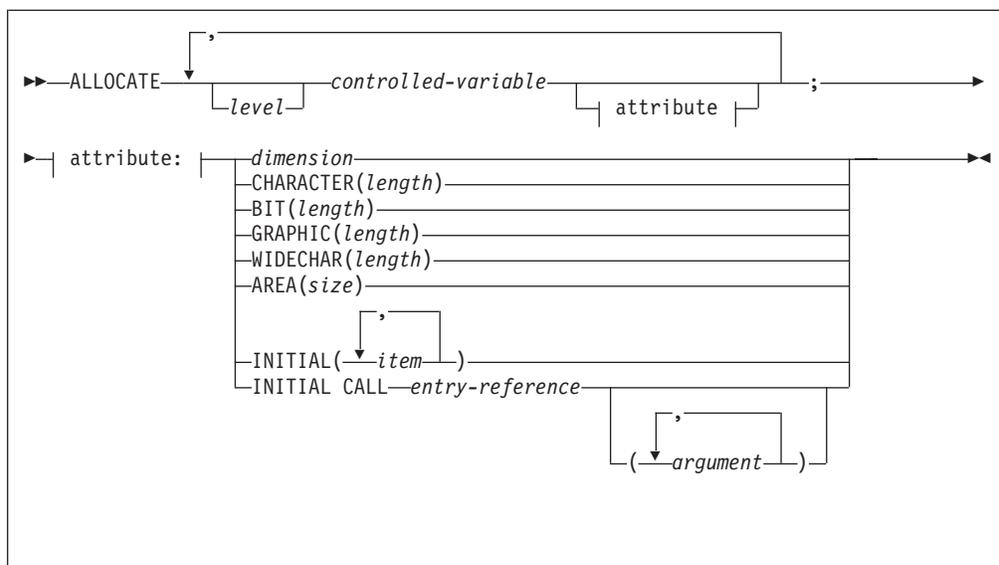
Abbreviation: ALLOC

level
> Indicates a level number. If no level number is specified, the controlled-variable named must be a level-1 variable.

controlled-variable
> Specifies a controlled variable or an element of a controlled major structure. A structure element, other than the major structure, can appear only if the relative structuring of the entire major structure containing the element appears as it is in the DECLARE statement for that structure. In this case, dimension attributes must be specified for all names that are declared with the dimension attribute.

Both controlled and based variables can be allocated in the same statement. For the syntax of based variables, refer to "ALLOCATE statement for based variables" on page 234.

Bounds for arrays, lengths of strings, and sizes of areas (extents) are evaluated at the execution of an ALLOCATE statement:

- Either the ALLOCATE statement or a DECLARE or DEFAULT statement must specify any necessary dimension, size, or length attributes (extents) for a variable. Any expression taken from a DECLARE statement is evaluated at the point of allocation using the conditions enabled at the ALLOCATE statement. However, names in the expression refer to those variables whose scope includes the DECLARE or DEFAULT statement.

- If a bound, length, or size is explicitly specified in an ALLOCATE statement, it overrides that given in the DECLARE statement for that variable.

- If a bound, length, or size is specified by an asterisk in an ALLOCATE statement, that extent is taken from the current generation. If no generation of the variable exists, the extent is undefined and the program is in error.

- If, in either an ALLOCATE or a DECLARE statement, bounds, lengths, or sizes are specified by expressions that contain references to the variable being allocated, the expressions are evaluated using the value of the most recent generation of the variable. For example:

```
declare X(N) fixed bin ctl;
N = 20;
allocate X;
allocate X(X(1));
```

> In the first allocation of X, the upper bound is specified by the declare statement and N = 20;. In the second allocation, the upper bound is specified by the value of the first element of the first generation of X.

The dimension attribute must specify the same number of dimensions as declared. The dimension attribute can appear with any of the other attributes and must be the first attribute specified. For example:

```
declare X(M) char(N) controlled;
M = 20;
N = 5;
allocate X(25) char(6);
```

The BIT, CHARACTER, GRAPHIC, WIDECHAR and AREA attributes can appear only for variables having the same attributes, respectively.

Initial values are assigned to a variable upon allocation, if the variable has an INITIAL attribute in either the DECLARE or ALLOCATE statement. Expressions or the CALL option in the INITIAL attribute are evaluated at the point of allocation, using the conditions enabled at the ALLOCATE statement. However, the names are interpreted in the environment of the declaration. If an INITIAL attribute appears in both DECLARE and ALLOCATE statements, the INITIAL attribute in the ALLOCATE statement is used. If initialization involves reference to the variable being allocated, the reference is to the new generation of the variable. For more information on initialization, refer to "INITIAL attribute" on page 251.

Any evaluations performed at the time the ALLOCATE statement is executed (for example, evaluation of expressions in an INITIAL attribute) must not be interdependent.

If storage for the controlled variable is not available, the STORAGE condition is raised.

FREE statement for controlled variables

The FREE statement frees the storage allocated for controlled variables. The freed storage is then available for other allocations. The previously allocated controlled variable is made available, and subsequent references refer to that allocation.
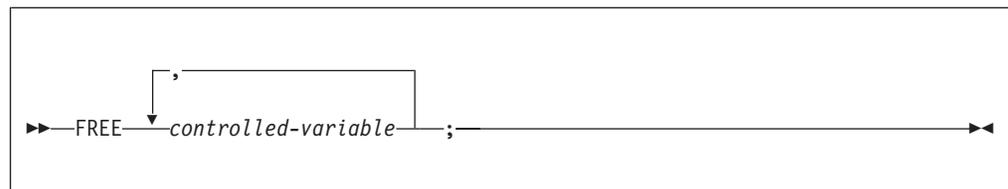
controlled-variable
> A level-1, unsubscripted variable.

Both based and controlled variables can be freed in the same statement. For the syntax of based variables, refer to "FREE statement for based variables" on page 235.

Implicit freeing

A controlled variable need not be explicitly freed by a FREE statement. However, it is a good practice to explicitly FREE controlled variables.

All controlled storage is freed at the termination of the program.

Multiple generations of controlled variables

An ALLOCATE statement for a variable for which storage was previously allocated and not freed pushes down or stacks storage for the variable. This stacking creates a new generation of data for the variable. The new generation becomes the current generation. The previous generation cannot be directly accessed until the current generation has been freed. When storage for this variable is freed, using the FREE statement or at termination of the program in which the storage was allocated, storage is popped up or removed from the stack.

Asterisk notation

In an ALLOCATE statement, values are inherited from the most recent previous generation when dimensions, lengths, or sizes are indicated by asterisks. For arrays, the asterisk must be used for every dimension of the array, not just one of them. For example:

```
dcl X(M,N) char(A) ctl;
  M=10;
  N=20;
  A=5;

allocate X;
allocate X(10,10);
allocate X(*,*);
```

The first generation of X has bounds (10,20); the second and third generations have bounds (10,10). The elements of each generation of X are all character strings of length 5.

The asterisk notation can also be used in a DECLARE statement, but has a different meaning there. For example:

```
dcl Y char(*) ctl,
N fixed bin;

N=20;
allocate Y char(N);
allocate Y;
```

The length of the character string Y is taken from the previous generation unless it is specified in an ALLOCATE statement. In that case, Y is given the specified length. This allows you to defer the specification of the string length until the actual allocation of storage.

Adjustable extents

Controlled scalars, arrays, and members of structures and unions can have adjustable array extents, string lengths, and area sizes. In the following example, when the structure is allocated, A.B has the extent 1 to 10 and A.C is a varying character string with maximum length 5.

```
dcl 1 A ctl,
2 B(N:M),
2 C char(*) varying;
N = -10;
M = 10;
```

```
alloc 1 A,
2 B(1:10),
2 C char(5);
free A;
```

Built-in functions for controlled variables

The ALLOCATION built-in function can be used to determine the number of generations that have been allocated for a given controlled variable. If the variable is not allocated, the function returns the value zero.

Based storage and attribute

A declaration of a based variable is a description of the generation: the amount of storage required and its attributes. (A based variable does not identify the location of a generation in main storage.) A locator value identifies the location of the generation. Any reference to a based variable that is not allocated is in error.

```
►►─BASED─────────────────────────────────────────────►◄
         └─(locator-reference)─┘
```
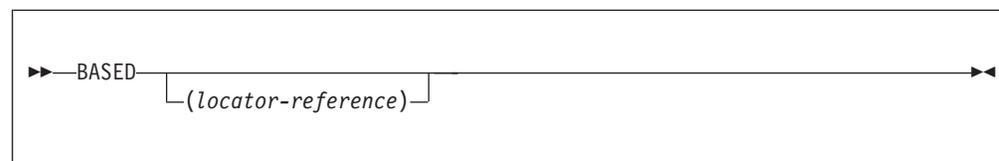
locator-reference
    Identifies the location of the data.

When reference is made to a based variable, the data and alignment attributes used are those of the based variable, while the qualifying locator variable identifies the location of data.

A based variable cannot have the EXTERNAL attribute, but a locator reference for a based variable can have any storage class, including based.

A based structure or union can be declared to contain adjustable area sizes, array-bounds, and string-length specifications, by using the REFER option. See "REFER option (self-defining data)" on page 236. The maximum length of a based VARYING or VARYINGZ string must be equal to the maximum length of any string upon which the based VARYING or VARYINGZ string is overlaid. For example:

```
declare A char(50) varying based(Q),
        B char(50) varying;
        Q=addr(B);
```

A based VARYING string can only be overlaid on a VARYING string; a based VARYINGZ string can only be overlaid on a VARYINGZ string.

Storage for a based variable can be allocated by using the ALLOCATE statement, the ALLOCATE built-in function, the AUTOMATIC built-in function, or the LOCATE statement. A based variable can also be used to access existing data by using the READ statement (with SET option), or the FETCH statement (with SET option), or the ADDR built-in function.

Based AREA variables can be allocated using the ALLOCATE statement; PL/I automatically initializes the area to EMPTY upon allocation. However, if you obtain storage for the area variable using the ALLOCATE or the AUTOMATIC built-in function, you must assign EMPTY to the variable after obtaining the storage.

Based storage and attribute

Because a locator variable identifies the location of any generation, you can refer at any point in a program to any generation of a based variable by using an appropriate locator value. The following example declares that references to X, except when the reference is explicitly qualified, use the locator variable P to locate the storage for X.

```
dcl X fixed bin based(P);
```

The association of a locator reference in this way is not permanent. The locator reference can be used to identify locations of other based variables and other locator references can be used to identify other generations of the variable X. When a based variable is declared without a locator reference, any reference to the based variable must always be explicitly locator-qualified.

In the following example, the arrays A and C refer to the same storage. The elements B and C(2,1) also refer to the same storage.

```
dcl A(3,2) character(5) based(P),
    B char(5) based(Q),
    C(3,2) character(5);
P = addr(C);
Q = addr(A(2,1));
```

Note: When a based variable is overlaid in this way, no new storage is allocated. The based variable uses the same storage as the variable on which it is overlaid (C(3,2) in the example).

You can also use the DEFINED and UNION attributes to overlay variable storage, but DEFINED and UNION overlay the storage permanently. When based variables are overlaid with a locator reference, the association can be changed at any time in the program by assigning a new value to the locator variable.

For more information on the DEFINED and UNION attributes, refer to "DEFINED and POSITION attributes" on page 245, and "Unions" on page 172.

The INITIAL attribute can be specified for a based variable. The initial values are assigned only upon explicit allocation of the based variable with an ALLOCATE or LOCATE statement.

Locator data

There are two types of locator data: pointer and offset.

The value of a *pointer variable* is an address of a location in storage. It can be used to qualify a reference to a variable with allocated storage in several different locations.

The value of an *offset variable* specifies a location within an area variable and remains valid when the area is assigned to a different part of storage.

A locator value can be assigned only to a locator variable. When an offset value is assigned to an offset variable, the area variables named in the OFFSET attributes are ignored.

Locator conversion

Locator data cannot be converted to other data types, except as follows:

- To and from REAL FIXED BINARY (p,0) by using the BINARYVALUE, POINTERVALUE, and OFFSETVALUE built-in functions

- Between pointer and offset implicitly or explicitly using the POINTER and OFFSET built-in functions.

When an offset variable is used in a reference, it is implicitly converted to a pointer value by using the address of the area variable designated in the OFFSET attribute and the offset variable. Explicit conversion of an offset to a pointer value is accomplished using the POINTER built-in function. For example, the following statement assigns a pointer value to P, giving the location of a based variable, identified by offset O in area B.

```
dcl P pointer, O offset(A),B area;
P = pointer(O,B);
```

Because the area variable is different from that associated with the offset variable, you must ensure that the offset value is valid for the different area. It is valid, for example, if area A is assigned to area B prior to the invocation of the function.

The OFFSET built-in function, in contrast to the POINTER built-in function, returns an offset value derived from a given pointer and area. The given pointer value must identify the location of a based variable in the given area.

A pointer value is converted to offset by using the pointer value and the address of the area. This conversion is limited to pointer values that relate to addresses within the area named in the OFFSET attribute.

Except when assigning the NULL or the SYSNULL built-in function value, it is an error to attempt to convert from or to an offset variable that is not associated with an area.

There is no implicit locator conversion in multiple assignments.

Locator reference

A locator reference is either a locator variable that can be qualified or subscripted, or a function reference that returns a locator value.

A locator reference can be used in the following ways:
- As a locator qualifier, in association with a declaration of a based variable
- In a comparison operation, as in an IF statement
- As an argument in a procedure reference.

Because PL/I implicitly converts an offset to a pointer value, offset references can be used interchangeably with pointer references.

Locator qualification

Locator qualification is the association of one or more locator references with a based reference to identify a particular generation of a based variable. This is called a locator-qualified reference. The composite symbol –> represents "qualified by" or "points to". The following syntax diagram is for an explicit qualified reference.

Locator Data

```
 ►►──locator-reference ─► ──┬─────────────────────────────┬──────────────►
                            │  ┌───────────────────────┐  │
                            └──┴─based-locator-reference ─►─┘

 ►──based-variable─────────────────────────────────────────────►◄
```
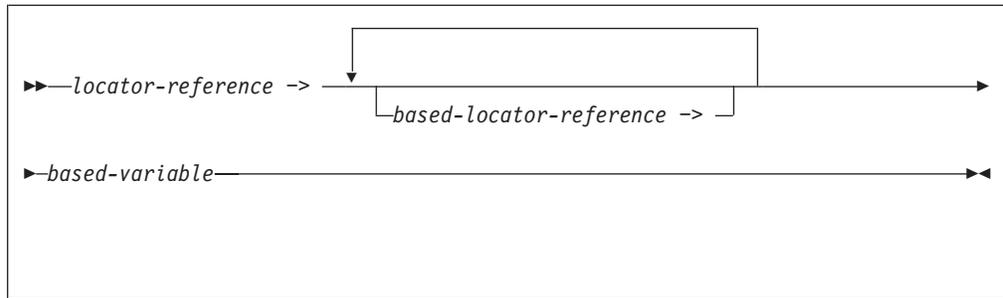
locator-reference
based-locator-reference
>   Identify the location of the data.

In the following example, X is a based variable, P is a locator variable, and Q is a based locator variable.

```
P -> Q -> X
```

The reference means that it is that generation of X that is identified by the based locator Q that is also identified by the value of the locator P. X and Q are said to be *explicitly locator-qualified*.

When more than one locator qualifier is used, they are evaluated from left to right.

Reference to a based variable can also be *implicitly qualified*. The locator reference used to determine the generation of a based variable that is implicitly qualified is the one declared with the based variable. In the following example, the ALLOCATE statement sets the pointer variable P so that the reference X applies to allocated storage.

```
dcl X fixed bin based(P) init(0);
allocate X;
X = X + 1;
```

The references to X in the assignment statement are implicitly locator-qualified by P. References to X can also be explicitly locator-qualified as shown in the following example.

```
P->X = P->X + 1;
```

The following assignment statements have the same effect as the previous example:

```
Q = P;
Q->X = Q->X + 1;
```

Because the locator declared with a based variable can also be based, a chain of locator qualifiers can be implied. For example, the following pointer and based variables can be used:

```
declare (P(10),Q) pointer,
        R pointer based (Q),
        V based (P(3)),
        W based (R),
        Y based;
allocate R,V,W;
```

Given the previous declaration and allocation, the following references are valid:

```
P(3) -> V
V
Q -> R -> W
R -> W
W
```

The first two references are equivalent, and the last three are equivalent. Any reference to `Y` must include a qualifying locator variable.

Levels of locator qualification

A pointer that qualifies a based variable represents one level of locator qualification. An offset represents two levels because it is implicitly qualified within an area. The number of levels is not affected by a locator being subscripted and/or an element of a structure or union. In the following example, the references `X`, `P -> X`, and `Q -> P -> X` represent three levels of locator qualification.

```
declare X based (P),
        P pointer based (Q),
        Q offset (A);
```

POINTER variable and attribute

A pointer variable is declared contextually if it appears in the declaration of a based variable, as a locator qualifier, in a BASED attribute, or in the SET option of an ALLOCATE, LOCATE, READ, or FETCH statement. It can also be declared explicitly.

```
►►──POINTER───────────────────────────────────────────────►◄
```
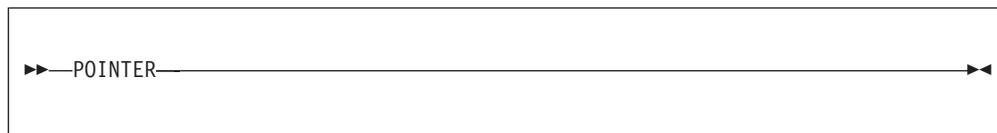
Abbreviation: PTR

The value of a pointer variable that no longer identifies a generation of a based variable is undefined (for example, when a based variable has been freed). Before a reference is made to a pointer-qualified variable, the pointer must have a value.

Built-in functions for based variables

The ALLOCATE built-in function can be used to obtain storage for a based variable, and the PLIFREE built-in subroutine can be used to free such storage. The AUTOMATIC built-in function can also be used to obtain storage for a based variable, but such storage must not be explicitly freed. Storage allocated with the AUTOMATIC built-in function is automatically freed when the block in which it is allocated terminates.

The ADDR built-in function returns a pointer value that identifies the first byte of a variable. The ENTRYADDR built-in function returns a pointer value that is the address of the first executed instruction if the entry were to be invoked. The NULL and SYSNULL built-in functions return the PL/I null pointer and the system null pointer respectively.

Note: The NULL and SYSNULL built-in functions can, but do not necessarily, compare equally. Your application program must **not** depend on the functions' equality.

ALLOCATE statement for based variables

The ALLOCATE statement allocates storage for based variables and sets a locator variable that can be used to identify the location, independent of procedure block boundaries.
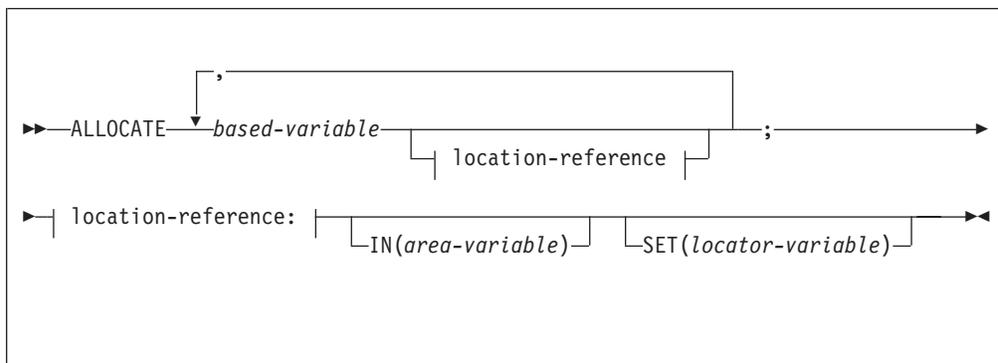
Abbreviation: ALLOC

based variable

> Is a level-1 unsubscripted variable.

IN

> Specifies the area variable in which the storage is allocated. For more information on areas, refer to "Area data and attribute" on page 237.

SET

> Specifies a locator variable that is set to the location of the storage allocated. If the SET option is not specified, the locator used is the one specified in the declaration of the based variable. For syntax information about declaring based variables, refer to "Based storage and attribute" on page 229 and "Locator data" on page 230.

Both based and controlled variables can be allocated in the same statement. For the syntax of controlled variables, see "ALLOCATE statement for controlled variables" on page 225.

Storage is allocated in an area when the IN option is specified or the SET option specifies an offset variable. These options can appear in any order. For allocations in areas:

- If sufficient storage for the based variable does not exist within the area, the AREA condition is raised.
- If the IN option is not used when using an offset variable, the declaration of the offset variable must specify an area reference.

When an area is not used, the locator variable must be a pointer variable. If storage for the based variable is not available, the STORAGE condition is raised.

The amount of storage allocated for a based variable depends on its attributes, and on its dimensions, length, or size specifications if these are applicable at the time of allocation. These attributes are determined from the declaration of the based variable.

A based structure or union can contain adjustable array bounds or string lengths or area sizes (see "REFER option (self-defining data)" on page 236). The asterisk notation for extents is not allowed for based variables.

FREE statement for based variables

The FREE statement frees the storage allocated for based and controlled variables.

```
>>-FREE--+-option-+--;--+-option:-+--------------------------->
         '-,------'              |
                                 +-locator-reference ->-+

>--based-variable--+----------------+--------------------------><
                   '-IN(area-variable)-'
```
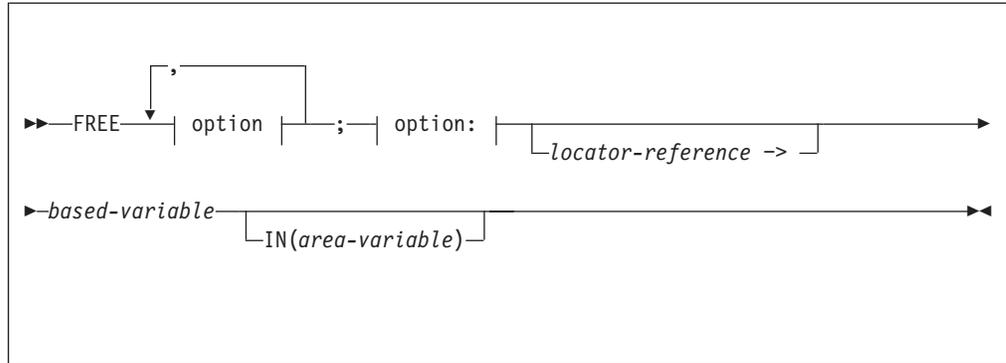
locator-reference ->
Frees a particular generation of a based variable. The composite symbol -> means "qualified by" or "points to." If the based variable is not explicitly locator-qualified, the locator variable declared in the BASED attribute is used to identify the generation of data to be freed. If no locator has been declared, the statement is in error.

based variable
Must be a level-1 unsubscripted based variable.

IN Must be specified or the based variable must be qualified by an offset declared with an associated area, if the storage to be freed was allocated in an area. The IN option cannot appear if the based variable was not allocated in an area. Area assignment allocates based storage in the target area. These allocations can be freed by the IN option naming the target area.

Both based and controlled variables can be freed in the same statement. For the syntax of controlled variables, see "FREE statement for controlled variables" on page 227.

A based variable can be used to free storage only if that storage has been allocated for a based variable having identical data attributes.

The amount of storage freed depends upon the attributes of the based variable, including bounds and/or lengths at the time the storage is freed. The user is responsible for determining that this amount coincides with the amount allocated. If the variable has not been allocated, the results are unpredictable.

Implicit freeing
A based variable need not be explicitly freed by a FREE statement, but it is a good practice to do so.

All based storage is freed at the termination of the program.

REFER option (self-defining data)

A self-defining structure or union contains information about its own fields, such as the length of a string. A based structure or union can be declared to manipulate this data. String lengths, array bounds, and area sizes can all be defined by variables, known as the *refer object*, declared within the structure or union. When the structure or union is allocated (by either an ALLOCATE statement or a LOCATE statement), the value of an expression is assigned to the refer object variable. For any other reference to the structure or union, the value of the refer object is used.

The REFER option is used in the declaration of a based structure or union to specify that, on allocation of the structure or union, the value of an expression is assigned to the refer object and represents the length, bound, or size of another variable in the structure or union. The syntax for a length, bound, or size with a REFER option is shown in the following diagram.

```
►►──expression──REFER──(member-variable)──────────────────────►◄
```
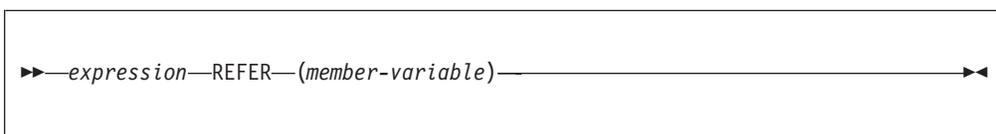
expression

The value of this expression defines the length, bound, or size of the member when the structure or union is allocated (using ALLOCATE or LOCATE). The expression is evaluated and converted to FIXED BINARY (31,0). Any variables used as operands in the expression must not belong to the structure or union containing the REFER option.

Subsequent references to the structure or union obtain the REFER option member's length, bound, or size from the current value of *member-variable* (refer object).

member-variable

The refer object must conform to the following rules:

- It must be a member of the same level-1 structure or union, and it must appear before any member that names it in a REFER option.
- It must be computational.
- It cannot be locator-qualified (see "Locator qualification" on page 231) or subscripted.
- It cannot be part of an array.

In the following example, the declaration specifies that the based structure STR consists of an array Y and an element X.

```
declare 1 STR based(P),
          2 X fixed binary(31,0),
          2 Y (L refer (X)),
        L fixed binary(31,0) init(1000);
```

When STR is allocated, the upper bound is set to the current value of L which is assigned to X. For any other reference to Y, such as a READ statement that sets P, the bound value is taken from X.

If the INITIAL attribute is specified for the member with the REFER option, initialization of the member occurs after the refer object has been assigned its value.

Any number of REFER options can be used in the declaration of a structure or union.

The value of the refer object should not be changed during program execution. It is an error to free such an aggregate if the value of the refer object has changed.

Note also that any variables used in the expression defining the REFER extent should be declared in the block (or one of its parent blocks) containing the DECLARE using that REFER. If one of the variables is not declared, it will be implicitly declared following the usual rules for implicit declaration, i.e. a DECLARE for it will be added to the outermost block containing the DECLARE.

This means that in the following code, the declaration of and assignment to the variable `m` in the subroutine `inner_proc` will have no effect on the ALLOCATE statement: the ALLOCATE statement will use the implicitly declared and uninitialized `m` from the main block!

```
 refertst: proc options(main);

    dcl
      1 a based,
        2 n fixed bin(31),
        2 c char(m refer(n));

    call inner_proc;

    inner_proc: proc;

      dcl m fixed bin(31);
      dcl p pointer;

      m = 15;
      allocate a set(p);
    end;
  end;
```

Area data and attribute

Area variables describe areas of storage that are reserved for the allocation of based variables. This reserved storage can be allocated to, and freed from, based variables by the ALLOCATE and FREE statements. Area variables can have any storage class and must be aligned.

When a based variable is allocated and an area is not specified, the storage is obtained from wherever it is available. Consequently, allocated based variables can be scattered widely throughout main storage. This is not significant for internal operations because items are readily accessed using the pointers. However, if these allocations are transmitted to a data set, the items have to be collected together. Items allocated within an area variable are already collected and can be transmitted or assigned as a unit while still retaining their separate identities.

You might want to identify the locations of based variables within an area variable relative to the start of the area variable. Offset variables are provided for this purpose.

An area can be assigned or transmitted complete with its contained allocations; thus, a set of based allocations can be treated as one unit for assignment and input/output while each allocation retains its individual identity.

Area data and attribute

The size of an area is adjustable in the same way as a string length or an array bound and therefore it can be specified by an expression or an asterisk (for a controlled area parameter) or by a REFER option (for a based area).

A variable is given the AREA attribute contextually by its appearance in the OFFSET attribute or an IN option, or by explicit declaration.

```
►►──AREA─┬──────────────────────────────────┬──►◄
         ├─(*)──────────────────────────┬───┤
         └─(expression─┬──────────────┬─)──┘
                       └─REFER(variable)─┘
```
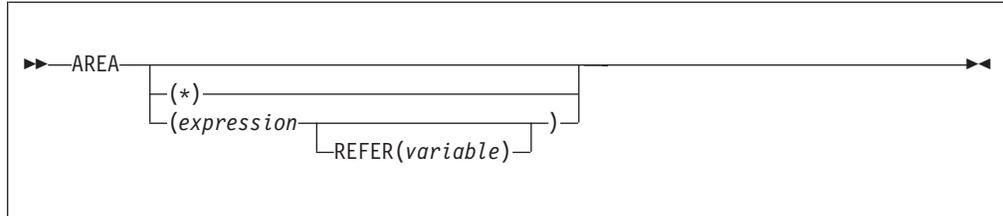
expression
> Specifies the size of the area. If *expression*, or an asterisk is not specified, the default is 1000.

* An asterisk can be used to specify the size if the area variable is declared is a parameter.

REFER
> For a description of the REFER option, refer to "REFER option (self-defining data)" on page 236.

The area size for areas that have the storage classes AUTOMATIC or CONTROLLED is given by an expression whose value specifies the number of reserved bytes.

If an area has the BASED attribute, the area size must be a constant unless the area is a member of a based structure or union and the REFER option is used.

The size for areas of static storage class must be specified as a restricted expression.

Examples of AREA declarations are:

```
  declare area1 area(2000),
         area2 area;
```

In addition to the declared size, an extra 16 bytes of control information precedes the reserved size of an area. The 16 bytes contain such details as the amount of storage in use.

The amount of reserved storage that is actually in use is known as the *extent* of the area. When an area variable is allocated, it is empty, that is, the area extent is zero. The maximum extent is represented by the area size. Based variables can be allocated and freed within an area at any time during execution, thus varying the extent of an area.

When a based variable is freed, the storage it occupied is available for other allocations. A chain of available storage within an area is maintained; the head of the chain is held within the control information. Inevitably, as based variables with different storage requirements are allocated and freed, gaps occur in the area when allocations do not fit available spaces. These gaps are included in the extent of the area.

No operators, including comparison, can be applied to area variables.

Offset data and attribute

Offset data is used exclusively with area variables. The value of an offset variable indicates the location of a based variable within an area variable relative to the start of the area. Because the based variables are located relatively, if the area variable is assigned to a different part of main storage, the offset values remain valid.

Offset variables do not preclude the use of pointer variables within an area.

```
►►──OFFSET──────────────────────────────────────►◄
            └─(area-variable)─┘
```
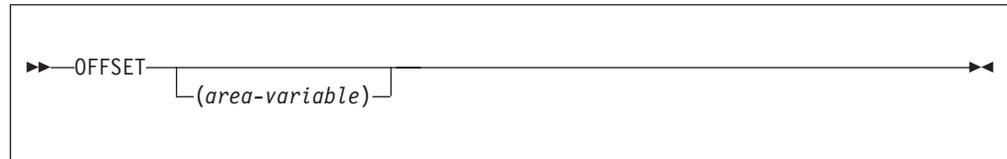
The association of an area variable with an offset variable is not permanent. An offset variable can be associated with any area variable by means of the POINTER built-in function (see "Locator conversion" on page 230). The advantage of making such an association in a declaration is that a reference to the offset variable implies reference to the associated area variable. If no area variable is specified, the offset can be used as a locator qualifier only through use of the POINTER built-in function.

Setting offset variables

The value of an offset variable can be set in any one of the following ways:

- By an ALLOCATE statement
- By assignment of the value of another locator variable, or a locator value returned by a user-defined function
- The NULL, SYSNULL, ADDR, ENTRYADDR, OFFSETADD, OFFSETSUBTRACT, OFFSETVALUE, or OFFSET built-in function

If no area variable is specified, the offset can be used only as a locator qualifier through use of the POINTER built-in function.

Examples of offset variables

Consider the following example:

```
dcl X based(O),
    Y based(P),
    A area,
    O offset(A);

    allocate X;
    allocate Y in(A);
```

The storage class of area A and offset O is AUTOMATIC by default. The first ALLOCATE statement is equivalent to:

```
allocate x in(A) set(O);
```

The second ALLOCATE statement is equivalent to:

```
allocate Y in(A) set(P);
```

The following example shows how a list can be built in an area variable using offset variables:

```
dcl A area,
    (T,H) offset(A),
    1 STR based(H),
      2 P offset(A),
      2 data;

    allocate STR in(A);
    T=H;

  do loop;
    allocate STR set(T->P);
    T=T->P;
 ⋮
  end;
```

Built-in functions for area variables

The EMPTY built-in function initializes the area variable to empty, freeing all allocations it might have. This is the initial state of an area variable in which no allocations have yet been made. The AVAILABLEAREA built-in function returns the size of the largest allocation that can be made in the area.

Area assignment

The value of an area reference can be assigned to one or more area variables by an assignment statement. Area-to-area assignment has the effect of freeing all allocations in the target area and then assigning the extent of the source area to the target area, so that all offsets for the source area are valid for the target area.

In the following example:

```
declare X based (O(1)),
        O(2) offset (A),
        (A,B) area;

        alloc X in (A);
        X = 1;
        alloc X in (A) set (O(2));
        O(2) -> X = 2;
        B = A;
```

Using the POINTER built-in function, the references `POINTER (O(2),B)->X` and `O(2)->X` represent the same value allocated in areas B and A, respectively.

If an area containing no allocations is assigned to a target area, the effect is to free all allocations in the target area.

Area assignment can be used to expand a list of based variables beyond the bounds of the original area. Attempting to allocate a based variable within an area that contains insufficient free storage to accommodate it, or attempting to assign an area to another area that is not large enough raises the AREA condition. The ON-unit for this condition can be used to change the value of a pointer qualifying the reference to the inadequate area, so that it points to a different area; on return from the ON-unit, the allocation is attempted again, within the new area. Alternatively, you can use the AVAILABLEAREA built-in function to determine whether the allocation you are about to make can be done in the area without raising the AREA condition. Also, the ON-unit can write out the area and reset it to EMPTY.

Input/output of areas

Areas allow input and output of complete lists of based variables as one unit, to and from RECORD files. On output, the area extent, together with the 16 bytes of control information, is transmitted, except when the area is in a structure or union and is not the last item in it—then, the declared size is transmitted. Thus the unused part of an area does not take up space on the data set.

Because the extents of areas can vary, varying length records should be used. The maximum record length required is governed by the area length (area size + 16).

List processing

List processing is the name for a number of techniques to help manipulate collections of data. Although arrays, structures, and unions are also used for manipulating collections of data, list processing techniques are more flexible since they allow collections of data to be indefinitely reordered and extended during program execution. The purpose here is not to illustrate these techniques but is to show how based variables and locator variables serve as a basis for this type of processing.

In list processing, a number of based variables with many generations can be included in a list. Members of the list are linked together by one or more pointers in one member identifying the location of other members or lists. The allocation of a based variable cannot specify where in main storage the variable is to be allocated (except that you can specify the area in which you want it allocated). In practice, a chain of items can be scattered throughout main storage, but by accessing each pointer the next member is found. A member of a list is usually a structure or union that includes a pointer variable. The following example creates a list of structures:

```
dcl 1 STR based(H),
      2 P pointer,
      2 data,
    T  pointer;

    allocate STR;
    T=H;

    do loop;
      allocate STR set(T->P);
        T=T->P;
        T->P=null;
⋮
    end;
```

The structures are generations of `STR` and are linked by the pointer variable `P` in each generation. The pointer variable `T` identifies the previous generation during the creation of the list. The first ALLOCATE statement sets the pointer `H` to identify it. The pointer `H` identifies the start, or head, of the list. The second ALLOCATE statement sets the pointer `P` in the previous generation to identify the location of this new generation. The assignment statement `T=T->P;` updates pointer `T` to identify the location of the new generation. The assignment statement `T->P=NULL;` sets the pointer in the last generation to NULL, giving a positive indication of the end of the list.

Figure 17 on page 242 shows a diagrammatic representation of a one-directional chain.

```
              ITEM 1                    ITEM 2                    ITEM 3

 ┌─────────┐   ┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐
 │         │   │                  │   │                  │   │                  │
 │  Head   │──▶│  Forward Pointer │──▶│  Forward Pointer │──▶│      NULL        │
 │         │   │                  │   │                  │   │                  │
 └─────────┘   ├──────────────────┤   ├──────────────────┤   ├──────────────────┤
              │                  │   │                  │   │                  │
              │     Data 1       │   │     Data 2       │   │     Data 3       │
              │                  │   │                  │   │                  │
              └──────────────────┘   └──────────────────┘   └──────────────────┘
```
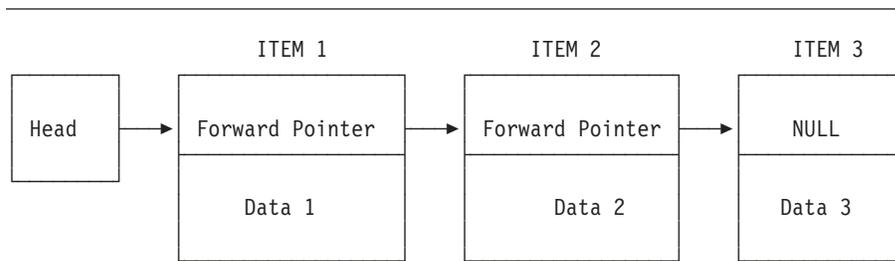
Figure 17. Example of one-directional chain

Unless the value of P in each generation is assigned to a separate pointer variable for each generation, the generations of STR can be accessed only in the order in which the list was created. For the above example, the following statements can be used to access each generation in turn:

```
  do T=H
     repeat(T->P)
     while (T¬=null);
⋮
       T->data;
⋮
end;
```

The foregoing examples show a simple list processing technique, the creation of a unidirectional list. More complex lists can be formed by adding other pointer variables into the structure or union. If a second pointer is added, it can be made to point to the previous generation. The list is then bidirectional; from any item in the list, the previous and next items can be accessed by using the appropriate pointer value. Instead of setting the last pointer value to the value of NULL, it can be set to point to the first item in the list, creating a ring or circular list.

A list need not consist only of generations of a single based variable. Generations of different based structure or unions can be included in a list by setting the appropriate pointer values. Items can be added and deleted from a list by manipulating the values of pointers. A list can be restructured by manipulating the pointers so that the processing of data in the list can be simplified.

ASSIGNABLE and NONASSIGNABLE attributes

The ASSIGNABLE and NONASSIGNABLE attributes specify whether the associated variable can be the target of an assignment.

```
              ┌─ASSIGNABLE──────┐
►►────────────┼─────────────────┼──────────────────────────────►◄
              └─NONASSIGNABLE───┘
```
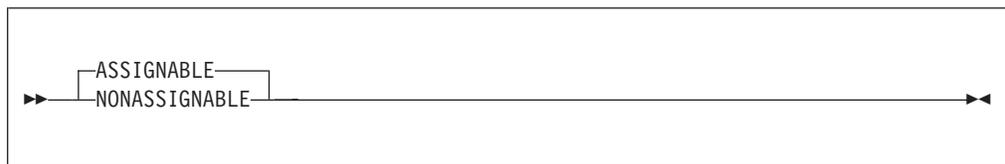
Abbreviations: ASGN, NONASGN

Default: ASSIGNABLE

If a variable has the NONASSIGNABLE attribute, the variable cannot be assigned.

If an entry descriptor has the NONASSIGNABLE attribute, the argument is assumed not to change when the associated ENTRY is invoked. If the argument is a constant, no dummy argument is created.

The ASSIGNABLE and NONASSIGNABLE attributes are propagated to members of structures or unions.

NORMAL and ABNORMAL attributes

The NORMAL and ABNORMAL attributes specify whether the associated variable is subject to change at any time.

The ABNORMAL attribute specifies that the value of the variable can change between statements or within a statement. An abnormal variable is fetched from or stored in storage each time it is needed or each time it is changed. All optimization is inhibited for an abnormal variable.

```
         ┌─NORMAL───┐
►►───────┴─ABNORMAL─┴──────────────────────────────────────►◄
```
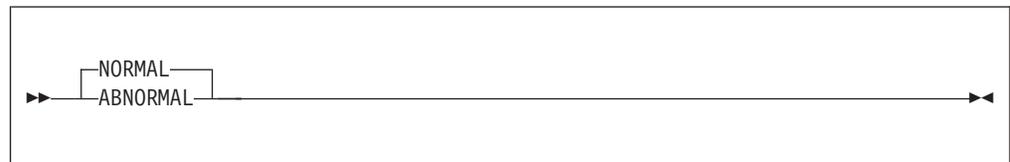
Default: NORMAL

The NORMAL and ABNORMAL attributes are propagated to members of structures or unions.

If the ABNORMAL attribute applies to an INTERNAL STATIC variable with an INITIAL value, the variable (with its initial value) will appear in the generated object code even if the variable is otherwise unused.

BIGENDIAN and LITTLEENDIAN attributes

The BIGENDIAN and LITTLEENDIAN attributes specify whether the associated variable is stored with the most or least significant digits first. The BIGENDIAN and LITTLEENDIAN attributes are ignored except for FIXED BINARY, ORDINAL, OFFSET, POINTER, and AREA variables and VARYING string variables.

```
         ┌─BIGENDIAN────┐
►►───────┴─LITTLEENDIAN─┴──────────────────────────────────►◄
```
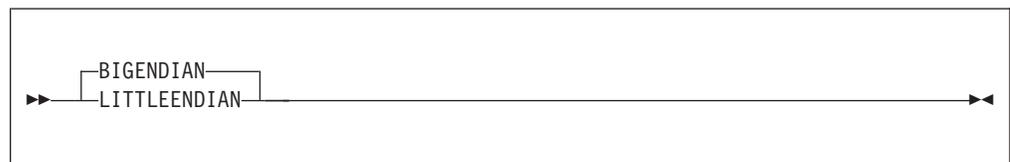
Default: BIGENDIAN

BIGENDIAN indicates that the variable (for varying strings, the length prefix part of the variable) is stored with its most significant bytes first. This format is the native style for S/370 and RS/6000.

LITTLEENDIAN indicates that the variable is stored in the opposite format: with its least significant bytes first. This format is the native style for OS/2 and Windows.

BIGENDIAN and LITTLEENDIAN

When the LITTLEENDIAN or BIGENDIAN attribute is applied to an AREA, it affects only the format in which the control values managed by the compiler and library are held. It has no effect on user variables stored in the AREA or on user offset variables used to point to the user variables in the AREA.

The following example illustrates how BIGENDIAN and LITTLEENDIAN variables are stored. The built-in function HEXIMAGE shows how X and Y are actually stored.

```
dcl X fixed bin(15) bigendian;
dcl Y fixed bin(15) littleendian;


X = 258;
Y = 258;


display( heximage( addr(X), stg(X) ) );    /* displays 0102 */
display( heximage( addr(Y), stg(Y) ) );    /* displays 0201 */
```

In contrast, the HEX built-in function would show for X and Y as given above:

```
display (hex(X));                                /* displays 0102   */
display (hex(Y));                                /* displays 0102   */
```

BIGENDIAN and LITTLEENDIAN have no effect on the semantics of any operations, or on the storage requirements for any variables.

The BIGENDIAN and LITTLEENDIAN attributes are propagated to members of structures or unions.

For more information on using BIGENDIAN and LITTLEENDIAN, refer to the Programming Guide.

The NATIVE and NONNATIVE attributes are synonyms for BIGENDIAN and LITTLEENDIAN, but their meanings can vary across different systems:
- On z/OS and RS/600, NATIVE means BIGENDIAN
- On OS/2 and Windows, NATIVE means LITTLEENDIAN

HEXADEC and IEEE attributes

HEXADEC and IEEE specify whether the associated variable is stored using the same format as on S/370 or using the OS/2, Windows, or AIX format. The HEXADEC and IEEE attributes are ignored except for floating-point variables.

```
        ┌─IEEE───┐
►►──┼─HEXADEC─┼──────────────────────────────────────────►◄
```
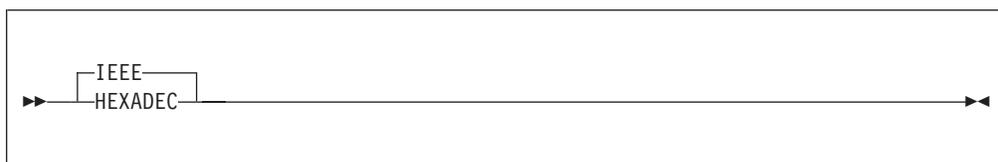
Default: IEEE

HEXADEC indicates that the variable is stored in hexadecimal (S/370) format.

IEEE indicates that the variable is stored using the IEEE format.

All computations are done using IEEE floating-point; variables declared HEXADEC will be converted as necessary.

CONNECTED and NONCONNECTED attributes

Elements, arrays, and major structure or unions are always allocated in connected storage. References to unconnected storage arise only when you refer to an aggregate that is made up of noncontiguous items from a larger aggregate. (See "Cross sections of arrays" on page 170.) For example, in the following structure the interleaved arrays A.B and A.C are both in unconnected storage.

```
 1 A(10),
   2 B,
   2 C;
```

```
                ┌─NONCONNECTED─┐
►►──────────────┴─CONNECTED────┴────────────────────────────────►◄
```

Abbreviations: CONN, NONCONN

Default: NONCONNECTED

The CONNECTED attribute is applicable only to noncontrolled aggregate parameters and can be specified only on level-1 names. It specifies that the parameter is a reference to connected storage only, and therefore, allows the parameter to be used as a target or source in record-oriented I/O, or as a base in string overlay defining. When the parameter is connected and the CONNECTED attribute is used, more efficient object code is produced for references to the connected parameter.

NONCONNECTED should be specified if a parameter occupies noncontiguous storage. In the following example the NONCONNECTED attribute specifies that the sum_Slice routine handles 1-dimensional arrays in which the elements may not be contiguous. In the first invocation, sum_Slice is passed the first row, which is in connected storage. In the second invocation, however, sum_Slice is passed the first column, which is in nonconnected storage.

```
  dcl A(10,10) fixed bin(31);

  display( sum_Slice( A(1,*) ) );    /* first row    */
  display( sum_Slice( A(*,1) ) );    /* first column */

  sum_Slice:proc(X) returns(fixed bin(31));

    dcl X (*) fixed bin(31) nonconnected; /* default */
  return(sum(X) );
  end;
```

DEFINED and POSITION attributes

The DEFINED attribute specifies that the declared variable is associated with some or all of the storage associated with the designated base variable.

The UNION attribute allows you to achieve the same end in a much cleaner manner and also allows variables with different attributes and precisions to be overlaid. Also, while the DEFINED attribute guarantees that access through defined or base variables is reflected in all defined variables, in a union only one member of

the union is valid at any given time. For syntax information on the UNION attribute, refer to "UNION attribute" on page 172.

```
►►──DEFINED──┬──reference──────┬──┬──────────────────────────┬──►◄
             └──(reference)──┘  └──POSITION(expression)──┘
```
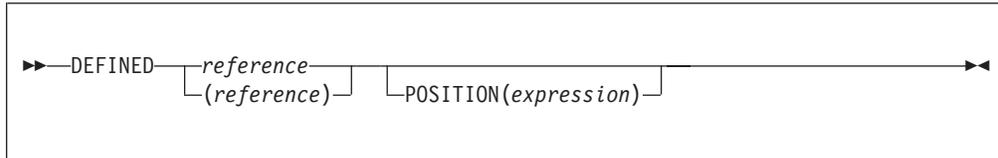
Abbreviations: DEF for DEFINED, POS for POSITION

reference
> To the variable (the *base variable*) whose storage is associated with the declared variable; the latter is the *defined variable*. The base variable can be EXTERNAL or INTERNAL. It can be a parameter (in string overlay defining, the parameter must refer to connected storage). It cannot be BASED or DEFINED. A change to the base variable's value is a corresponding change to the value of the defined variable, and vice versa.

If the base variable is a data aggregate, a defined variable can comprise all the data or only a specified part of it.

The defined variable does not inherit any attributes from the base variable. The defined variable must be INTERNAL and a level-1 identifier. It can have the dimension attribute. It cannot be INITIAL, AUTOMATIC, BASED, CONTROLLED, STATIC, or a parameter.

There are three types of defining: simple, iSUB and string overlay.

The type of defining in effect is determined as follows:
1. If the POSITION attribute is specified, string overlay defining is in effect.
2. If the subscripts specified in the base variable contain references to iSUB variables, iSUB defining is in effect.
3. If neither an iSUB variable nor the POSITION attribute is present and if the base variable and defined variable match according to the criteria given below. simple defining is in effect.
4. Otherwise, string overlay defining is in effect.

If the POSITION attribute is specified, the base variable must not contain iSUB references.

A base variable and a defined variable *match* if the base variable when passed as an argument matches a parameter which has the attributes of the defined variable (except for the DEFINED attribute). For this purpose, the parameter is assumed to have all array bounds, string lengths, and area sizes specified by asterisks.

For simple and iSUB defining, a PICTURE attribute can only be matched by a PICTURE attribute that is identical except for repetition factors. For a reference to specify a valid base variable in string overlay defining, the reference must be in connected storage. You can override the matching rule completely, but this can cause unwanted side effects within your program.

The values specified or derived for any array bounds, string lengths, or area sizes in a defined variable do not always have to match those of the base variable. However, the defined variable must be able to fit into the corresponding base array, string, or area.

In references to defined data, the STRINGRANGE, SUBSCRIPTRANGE, and STRINGSIZE conditions are raised for the array bounds and string lengths of the defined variable, not the base variable.

The determination of values and the interpretation of names occurs in the following sequence:

1. The array bounds, string lengths, and area sizes of a defined variable are evaluated on entry to the block that declares the variable.

2. A reference to a defined variable is a reference to the current generation of the base variable. When a defined variable is passed as an argument without creation of a dummy, the corresponding parameter refers to the generation of the base variable that is current when the argument is passed. This remains true even if the base variable is reallocated within the invoked procedure.

3. When a reference is made to the defined variable, the order of evaluation of the subscripts of the base and defined variable is undefined.

If the defined variable has the BIT attribute, unpredictable results can occur under the following conditions:

- If the base variable is not on a byte boundary
- If the defined variable is not defined on the first position of the base variable and the defined variable is used as:
  - A parameter in a subroutine call (that is, referenced as internally stored data)
  - An argument in a PUT statement
  - An argument in a built-in function (library call)
  - If the base variable is controlled, and the defined variable is dimensioned and is declared with variable array bounds

Unconnected Storage

The DEFINED attribute can overlay arrays. This allows array expressions to refer to array elements in unconnected storage (array elements that are not adjacent in storage). It is possible for an array expression involving consecutive elements to refer to unconnected storage in the following case:

- Where a string array is defined on a string array that has elements of greater length. Consecutive elements in the defined array are separated by the difference between the lengths of the elements of the base and defined arrays, and are held in unconnected storage.

An array overlay-defined on another array is always assumed to be in unconnected storage.

Simple Defining

Simple defining allows you to refer to an element, array, or structure variable by another name.

DEFINED and POSITION

The defined and base variables can comprise any data type, but they must match, as described earlier. The ALIGNED and UNALIGNED attributes must match for each element in the defined variable and the corresponding element in the base variable.

The defined variable can have the dimension attribute.

In simple defining of an array:
- The base variable can be a cross-section of an array.
- The number of dimensions specified for the defined variable must be equal to the number of dimensions specified for the base variable.
- The range specified by a bound pair of the defined array must equal or be contained within the range specified by the corresponding bound pair of the base array.

In simple defining of a string, the length of the defined string must be less than or equal to the length of the base string.

In simple defining of an area, the size of the defined area must be equal to the size of the base area.

A base variable can be, or can contain, a varying string, provided that the corresponding part of the defined variable is a varying string of the same maximum length.

Examples:

```
DCL A(10,10,10),
    X1(2,2,2) DEF A,
    X2(10,10) DEF A(*,*,5),
    X3 DEF A(L,M,N);
```

X1 is a three-dimensional array that consists of the first two elements of each row, column and plane of A. X2 is a two-dimensional array that consists of the fifth plane of A. X3 is an element that consists of the element identified by the subscript expressions L, M, and N.

```
DCL B CHAR(10),
    Y CHAR(5) DEF B;
```

Y is a character string that consists of the first 5 characters of B.

```
DCL C AREA(500),
    Z AREA(500) DEF C;
```

Z is an area defined on C.

```
DCL 1 D UNALIGNED,
        2 E,
        2 F,
            3 G CHAR(10) VAR,
            3 H,
    1 S UNALIGNED DEF D,
        2 T,
        2 U,
            3 V CHAR(10) VAR,
            3 W;
```

S is a structure defined on D. For simple defining, the organization of the two structures must be identical. A reference to T is a reference to E, V to G, and so on.

iSUB Defining

With iSUB defining, you can create a defined array that consists of designated elements from a base array. The defined and base arrays must be arrays of scalars, may comprise any data types, and must have identical attributes (apart from the dimension attribute).

The defined variable must have the dimension attribute. In the declaration of the defined array, the base array must be subscripted, and the subscript positions cannot be specified as asterisks.

A iSUB variable is a reference, in the subscript list for the base array, to the dimension of the defined array. At least one subscript in the base array's subscript-list must be an iSUB expression which, on evaluation, gives the required subscript in the base array. The value of i ranges from 1 to n, where n is the number of dimensions in the defined array. The number of subscripts for the base array must be equal to the number of dimensions for the base array.

If a reference to a defined array does not specify a subscript expression, subscript evaluation occurs during the evaluation of the expression or assignment in which the reference occurs.

The value of i is specified as an integer. Within an iSUB expression, an iSUB variable uis treated as REAL FIXED BINARY(31,0) variable.

A subscript in a reference to a defined variable is evaluated even if there is no corresponding iSUB in the base variable's subscript list.

An iSUB-defined variable may not appear in the data-list of a GET DATA or PUT DATA statement.

Examples:
```
DCL A(10,10) FIXED BIN
    X(10) FIXED BIN DEF( A(1SUB,1SUB) );
```

X is a one-dimensional array that consists of the diagonal of A: X(i) refers to the same storage as A(i,i).
```
DCL B(5,10) FIXED BIN
    Y(10,5) FIXED BIN DEF( A(2SUB,1SUB) );
```

Y is a two-dimensional array that consists of the elements of B with the bounds transposed: Y(i,j) refers to the same storage as X(j,i).

String Overlay Defining

String overlay defining allows you to associate a defined variable with the storage for a base variable. Both the defined and the base variable must be string or picture data.

Neither the defined nor the base variable can have the ALIGNED or the VARYING attributes.

Both the defined and the base variables must belong to:
* The bit class, consisting of:
    Fixed-length bit variables
    Aggregates of fixed-length bit variables

DEFINED and POSITION

- The character class, consisting of:
    Fixed-length character variables
    Character pictured and numeric pictured variables
    Aggregates of the two above
- The graphic class, consisting of:
    Fixed-length graphic variables
    Aggregates of fixed-length graphic variables
- The widechar class, consisting of:
    Fixed-length widechar variables
    Aggregates of fixed-length widechar variables

Examples:

```
DCL A CHAR(100),
    V(10,10) CHAR(1) DEF A;
```

V is a two-dimensional array that consists of all the elements in the character string A.

```
DCL B(10) CHAR(1),
    W CHAR(10) DEF B;
```

W is a character string that consists of all the elements in the array B.

POSITION attribute

The POSITION attribute can be used only with string-overlay defining and specifies the bit, character, graphic or widechar within the base variable at which the defined variable is to begin.

The expression in the POSITION attribute specifies the position relative to the start of the base variable. The value specified in the expression can range from 1 to *n*, where *n* is defined as

$n = N(b) - N(d) + 1$

where N(b) is the number of bits, characters, graphics or widechars in the base variable, and N(d) is the number of bits, characters, graphics or widechars in the defined variable.

The expression is evaluated and converted to an integer value at each reference to the defined item.

If the POSITION attribute is omitted, POSITION(1) is the default.

When the defined variable is a bit class aggregate:
- The POSITION attribute can contain only an integer.
- The base variable must not be subscripted.

The base variable must refer to data in connected storage.

Examples:

```
DCL C(10,10) BIT(1),
    X BIT(40) DEF C POS(20);
```

X is a bit string that consists of 40 elements of C, starting at the 20th element.

```
DCL E PIC'99V.999',
    Z1(6) CHAR(1) DEF (E),
    Z2 CHAR(3) DEF (E) POS(4),
    Z3(4) CHAR(1) DEF (E) POS(2);
```

Z1 is a character string array that consists of all the elements of the decimal numeric picture E. Z2 is a character string that consists of the elements '999' of the picture E. Z3 is a character-string array that consists of the elements '9.99' of the picture E.

```
DCL A(20) CHAR(10),
    B(10) CHAR(5) DEF (A) POSITION(1);
```

The first 50 characters of B consist of the first 50 characters of A. POSITION(1) must be explicitly specified. Otherwise, simple defining is used and gives different results.

INITIAL attribute

The INITIAL attribute specifies an initial value or values assigned to a variable at the time storage is allocated for it. Only one initial value can be specified for an element variable. More than one can be specified for an array variable. A structure or union variable can be initialized only by separate initialization of its elementary names, whether they are element or array variables. The INITIAL attribute cannot be given to constants, defined data, noncontrolled parameters, and non-LIMITED static entry variables.

The INITIAL attribute has three forms.

1. The first form, INITIAL, specifies an initial constant, expression, or function reference, for which the value is assigned to a variable when storage is allocated to it.
2. The second form, INITIAL CALL, specifies (with the CALL option) that a procedure is invoked to perform initialization. The variable is initialized by assignment during the execution of the called routine. (The routine is not invoked as a function that returns a value to the point of invocation.)
3. The third form, INITIAL TO, specifies that the pointer (or array of pointers) is initialized with the address of the character string specified in the INITIAL LIST. The string also has the attributes indicated by the TO keyword.

INITIAL

```
>>--INITIAL--(----,----)--| item |--)--| item: |----*----------------------->◄
              |  item  |              |----| initial-constant |--|
                                      |----reference-----------|
                                      |----(expression)--------|
                                      |----| iteration-specification |--|
```

iteration-specification:

```
|--(--iteration-factor--)--| iteration-item |------------------------------|
    |---*------------|
```

iteration-item:

```
|--------------*----------------------------------------------------------|
  |--| initial-constant |--|
  |----reference----------|
  |        ,              |
  |  (----item----)       |
```

initial-constant:

```
|----------arithmetic-constant----------------------------------------------|
  |--+--|
  |--−--|
  |----bit-constant---------------------|
  |----character-constant---------------|
  |----graphic-constant-----------------|
  |----entry-constant-------------------|
  |----file-constant--------------------|
  |----label-constant-------------------|
  |--------real-constant--+--imaginary-constant--|
  |--+--|         |--−--|
  |--−--|
```

and

```
>>--INITIAL CALL----entry-reference----------------------------------------->◄
                  |--generic-name------|
                  |--built-in-name-----|
                                      |        ,        |
                                      |  (----argument----)  |
```

```
               ┌─────────────────────┐
               │           ,         │
               │           ┌──────┐  │
►►──INITIAL TO──(──┬──varying───┬──)──(──▼──┤ item ├──)───────────────────►
                   ├──varyingz──┤
                   └──nonvarying─┘

►──┤ item: ├────(see description under INITIAL)───────────────────────►◄
```

Abbreviations: INIT, INIT CALL, INIT TO

* Specifies that the element is to be left uninitialized, except when the element is used as an iteration factor.

iteration factor
Specifies the number of times the iteration item is to be repeated in the initialization of elements of an array.

The iteration factor can be an expression or an asterisk.

- An expression is converted to FIXED BINARY(31). For static variables, it must be a constant.

- An asterisk indicates that the remaining elements should be initialized to the specified value.

A negative or zero iteration factor specifies no initialization.

constant
reference
expression
These specify an initial value to be assigned to the initialized variable.

INITIAL CALL
For INITIAL CALL, the entry reference and argument list passed must satisfy the condition stated for block activation as discussed under "Block activation" on page 89.

INITIAL CALL cannot be used to initialize static data.

The following example initializes all of the elements of A to X'00' without the need for the INITIAL attribute on each element:

```
dcl  1  A automatic,
        2  ...,
        2  ...,
        2  * char(0) initial call plifill( addr(A), '00'X, stg(A) );
```

An AUTOMATIC variable that has an INITIAL CALL attribute will be retained even if otherwise unused (in case the logic of your program requires that the call to be executed).

If the procedure invoked by the INITIAL CALL statement has been specified in a FETCH or RELEASE statement and it is not present in main storage, the INITIAL CALL statement initiates dynamic loading of the procedure. (For more information on dynamic loading, refer to "Dynamic loading of an external procedure" on page 101.)

INITIAL TO
Use only with static native pointers. Specifies that the pointer (or array of

pointers) is initialized with the address of the character string specified in the INITIAL LIST. Also specifies that the string has the attributes indicated by the TO keyword.

In the following example, `pdays` is initialized with the addresses of character varyingz strings containing the names of the weekdays.

```
dcl pdays(7) static ptr init to(varyingz)
                ('Sunday',
                 'Monday',
                 'Tuesday',
                 'Wednesday',
                 'Thursday',
                 'Friday',
                 'Saturday'  );
```

You should not change a value identified by a pointer initialized with INITIAL TO. The value can be placed in read-only storage and an attempt to change it could result in a protection exception. Given the array `pdays` in the preceding example, then, the following assignment is illegal:

```
dcl x char(30) varz based;

pdays(1)->x = 'Sonntag';
```

Initializing array variables

Initial values specified for an array are assigned to successive elements of the array in row-major order (final subscript varying most rapidly). If too many initial values are specified, the excess values are ignored; if not enough are specified, the remainder of the array is not initialized.

The initialization of an array of strings can include both string repetition and iteration factors. Where only one of these is given, it is taken to be a string repetition factor unless the string constant is placed in parentheses.

The iteration factor can be specified as *, which means that all of the remaining elements will be initialized with the given value.

The following examples illustrate the use of (and difference between) string repetition and iteration factors:

```
((2)'A') is equivalent to ('AA')
((2)('A')) is equivalent to ('A','A')
((2)(1)'A') is equivalent to ('A','A')
((*)(1)'A') is equivalent to ('A','A'...'A')
```

An area variable is initialized with the value of the EMPTY built-in function, on allocation. Any INITIAL clause for an area variable will be ignored.

If the attributes of an item in the INITIAL attribute differ from those of the data item itself, conversion is performed, provided the attributes are compatible.

INITIAL is not allowed on objects of REFER clauses.

Initializing unions

The members of a union can have initial values. However, if the union is static, only one member of the union can have the initial attribute. For nonstatic unions, initial attributes are applied in order of appearance. Subsequent initial values overwrite previous ones.

In the following example, the declaration for NT1 would be invalid if it had the static storage attribute.

```
dcl
  1 NT1 union automatic,
    2 Numeric_translate_table1 char(256)
                 init( (256)'00'X),
    2 *,
      3 * char(240),
      3 * char(10) init('0123456789'),
    2 * char(0);

dcl
  1 NT2 union static,
    2 Numeric_translate_table2 char(256),
    2 *,
      3 * char(        rank('0')    )
          init((1)(low(rank('0'))) ),
      3 * char(10) init('0123456789'),
      3 * char(        (256-(rank('0'))-10)    )
          init((1)(low( (256-(rank('0'))-10) )) ),
```

The declaration for NT2 is valid even though it has static storage class. Furthermore, the NT2 declaration is portable between EBCDIC and ASCII modes of execution.

Initializing static variables

For a variable that is allocated when the program is loaded, that is, a static variable, which remains allocated throughout execution of the program, any value specified in an INITIAL attribute is assigned only once. (Static storage for fetched procedures is allocated and initialized each time the procedure is loaded.)

If static variables are initialized using the INITIAL attribute, the initial values must be specified as restricted expressions. Extent specifications must be restricted expressions.

The restrictions on initializing static variables are as follows:
* STATIC ENTRY variables must have the LIMITED attribute (see "LIMITED attribute" on page 120).
* INITIAL is not allowed for static format variables.
* INITIAL is allowed for label variables that are not part of structures or unions. In this case, the label variable gets the CONSTANT attribute.
* INITIAL is not valid for AREA variables.
* Only one member of a static union can specify INITIAL.
* If a STATIC EXTERNAL item without the RESERVED attribute is given the INITIAL attribute in more than one declaration, the value specified must be the same in every case.

Initializing automatic variables

For automatic variables, which are allocated at each activation of the declaring block, any specified initial value is assigned with each allocation.

Initializing based and controlled variables

For based and controlled variables which are allocated at the execution of ALLOCATE statements (also LOCATE statements for based variables), any specified initial value is assigned with each allocation.

When storage for based variables is allocated using the ALLOCATE or the AUTOMATIC built-in functions, the initial values are not assigned; for area variables, the area is not implicitly initialized to EMPTY.

Examples

In the following example, when storage is allocated for Name, the character constant 'John Doe' (padded on the right to 10 characters) is assigned to it.

```
dcl Name char(10) init('John Doe');
```

In the following example, when Pi is allocated, it is initialized to the value 3.1416.

```
dcl Pi fixed dec(5,4) init(3.1416);
```

The following example specifies that A is to be initialized with the value of the expression B*C:

```
declare A init((B*C));
```

The following example results in each of the first 920 elements of A being set to 0. The next 80 elements consist of 20 repetitions of the sequence 5,5,5,9.

```
declare A (100,10) initial
       ((920)0, (20) ((3)5,9));
```

In the following example, only the first, third, and fourth elements of A are initialized; the rest of the array is not initialized. The array B is fully initialized, with the first 25 elements initialized to 0, the next 25 to 1, and the remaining elements to 0. In the structure C, where the dimension (8) has been inherited by D and E, only the first element of D is initialized. All the elements of E are initialized.

```
declare A(15) character(13) initial
        ('John Doe',
         *,
         'Richard Row',
         'Mary Smith'),

       B (10,10) decimal fixed(5)
         init((25)0,(25)1,(*)0),

       1 C(8),
         2 D initial (0),
         2 E initial((*)0);
```

When an array of structures or unions is declared with the LIKE attribute to obtain the same structuring as a structure or union whose elements have been initialized, only the first structure or union is initialized.

In the following example only J(1).H and J(1).I are initialized in the array of structures.

```
declare 1 G,
         2 H initial(0),
         2 I initial(0),
       1 J(8) like G;
```

Chapter 11. Input and output

PL/I input and output statements (such as READ, WRITE, GET, PUT) let you transmit data between the main and auxiliary storage of a computer. A collection of data external to a program is called a *data set*. Transmission of data from a data set to a program is called *input*. Transmission of data from a program to a data set is called *output*. (If you are using a terminal, "data set" can also mean your terminal.)

PL/I input and output statements are concerned with the logical organization of a data set and not with its physical characteristics. A program can be designed without specific knowledge of the input/output devices that is used when the program is executed. To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs models of data sets, called *files*. A file can be associated with different data sets at different times during the execution of a program.

PL/I uses two types of data transmission: stream and record.

In stream-oriented data transmission, the organization of the data in the data set is ignored within the program, and the data is treated as though it were a continuous stream of individual data values in character form. Data is converted from character form to internal form on input, and from internal form to character form on output.

For more information on stream-oriented data transmission, refer to Chapter 13, "Stream-oriented data transmission," on page 281.

Stream-oriented data transmission can be used for processing input data prepared in character form and for producing readable output, where editing is required. Stream-oriented data transmission allows synchronized communication with the program at run time from a terminal, if the program is interactive.

Stream-oriented data transmission is more versatile than record-oriented data transmission in its data-formatting abilities, but is less efficient in terms of run time.

In record-oriented data transmission, the data set is a collection of discrete records. The record on the external medium is generally an exact copy of the record as it exists in internal storage. No data conversion takes place during record-oriented data transmission. On input the data is transmitted exactly as it is recorded in the data set, and on output it is transmitted exactly as it is recorded internally.

For more information on record-oriented data transmission, refer to Chapter 12, "Record-oriented data transmission," on page 271.

Record-oriented data transmission can be used for processing files that contain data in any representation, such as binary, decimal, or character.

Record-oriented data transmission is more versatile than stream-oriented data transmission, in both the manner in which data can be processed and the types of data sets that it can process. Since data is recorded in a data set exactly as it appears in main storage, any data type is acceptable. No conversions occur, but you must have a greater awareness of the data structure.

It is possible for the same data set to be processed at different times by either stream or record data transmission. However, all items in the data set must be in character form.

The following sections in this chapter discuss the kinds of data sets, the attributes for describing files, and how you open and close files in order to transmit data. For more information about the types of data set organizations that PL/I recognizes, refer to the Programming Guide.

Data sets

In addition to being used as input from and output to your terminal, data sets are stored on a variety of auxiliary storage media, including magnetic tape and direct-access storage devices (DASDs). Despite their variety, these media have characteristics that allow common methods of collecting, storing, and transmitting data. The organization of a data set determines how data is recorded in a data set and how the data is subsequently retrieved so that it can be transmitted to the program. Records are stored in and retrieved from a data set either sequentially on the basis of successive physical or logical positions, or directly by the use of keys specified in data transmission statements.

PL/I supports the following types of data set organizations:
    Consecutive
    Indexed
    Relative
    Regional

The data set organizations differ in the way they store data and in the means they use to access data.

Consecutive

In the consecutive data set organization, records are organized solely on the basis of their successive physical positions. When the data set is created, records are written consecutively in the order in which they are presented. The records can be retrieved only in the order in which they were written.

Indexed

In the indexed data set organization, records are placed in a logical sequence based on the key of each record. An indexed data set must reside on a direct-access device. A character string key identifies the record and allows direct retrieval, replacement, addition, and deletion of records. Sequential processing is also allowed.

Relative

In the relative data set organization, numbered records are placed in a position relative to each other. The records are numbered in succession, beginning with one. A relative data set must reside on a direct-access device. A key that specifies the record number identifies the record and allows direct retrieval, replacement, addition, and deletion of records. Sequential processing is also allowed.

Regional

The regional data set organization is divided into numbered regions, each of which can contain one record. The regions are numbered in succession, beginning with zero. A region can be accessed by specifying its region number, and perhaps a key, in a data transmission statement. The key specifies the region number and identifies the region to allow optimized direct retrieval, replacement, addition, and deletion of records.

Files

To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs models of data sets, called *files*. These models determine how input and output statements access and process the associated data set. Unlike a data set, a file data item has significance only within the source program and does not exist as a physical entity external to the program.

A name that represents a file has the FILE attribute.

FILE attribute

The FILE attribute specifies that the associated name is a file constant or file variable.
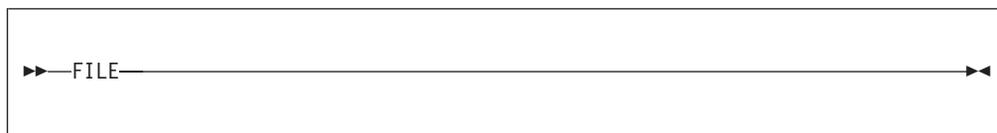
```
►►──FILE──────────────────────────────────────────────────►◄
```

The FILE attribute can be implied for a file constant by any of the file description attributes. A name can be contextually declared as a file constant through its appearance in the FILE option of any input or output statement, or in an ON statement for any input/output condition.

File constant

Each data set processed by a PL/I program must be associated with a file constant.

The individual characteristics of each file constant are described with file description attributes. These attributes fall into two categories: alternative attributes and additive attributes.

File constant

An *alternative* attribute is one that is chosen from a group of attributes. If no explicit or implied attribute is given for one of the alternatives in a group and if one of the alternatives is required, a default attribute is used.

Table 31 lists the PL/I alternative file attributes.

Table 31. Alternative file attributes

Group Type	Alternative Attributes	Default Attribute
Usage	STREAM or RECORD	STREAM
Function	INPUT or OUTPUT or UPDATE	INPUT
Access	SEQUENTIAL or DIRECT	SEQUENTIAL
Buffering	BUFFERED or UNBUFFERED	BUFFERED (for SEQUENTIAL files); UNBUFFERED (for DIRECT files)
Scope	EXTERNAL or INTERNAL	EXTERNAL

An *additive* attribute is one that must be stated explicitly or is implied by another explicitly stated attribute. The additive attributes are ENVIRONMENT, KEYED and PRINT. The additive attribute KEYED is implied by the DIRECT attribute. The additive attribute PRINT can be implied by the output file name SYSPRINT.

Table 32 shows the attributes that apply to each type of data transmission.

Table 32. Attributes by data transmission type

Type of transmission	Attribute
Stream-oriented	ENVIRONMENT INPUT and OUTPUT PRINT STREAM
Record-oriented	BUFFERED and UNBUFFERED DIRECT and SEQUENTIAL ENVIRONMENT INPUT, OUTPUT, and UPDATE KEYED RECORD

Table 33 on page 261 shows the valid combinations of file attributes.

Table 33. Attributes of PL/I file declarations

File Type Data Set Organization	STREAM Consecutive	RECORD SEQUENTIAL Consecutive	RECORD SEQUENTIAL Relative	RECORD SEQUENTIAL Indexed	RECORD DIRECT Relative	RECORD DIRECT Indexed	Legend:
							I Must be specified or implied **D** Default **O** Optional **S** Must be specified **-** Invalid
File Attributes							Attributes Implied
FILE	I	I	I	I	I	I	
INPUT¹	D	D	D	D	D	D	FILE
OUTPUT	O	O	O	O	O	O	FILE
ENVIRONMENT	O	O	O	O	O	O	FILE
STREAM	D	-	-	-	-	-	FILE
PRINT¹	O	-	-	-	-	-	FILE STREAM OUTPUT
RECORD	-	I	I	I	I	I	FILE
UPDATE	-	O	O	O	O	O	FILE RECORD
SEQUENTIAL	-	D	D	D	-	-	FILE RECORD
KEYED²	-	-	O	O	I	I	FILE RECORD
DIRECT	-	-	-	-	S	S	FILE RECORD KEYED

Notes:
 ¹ A file with the INPUT attribute cannot have the PRINT attribute
 ² KEYED is required for *indexed* and *relative* output

Scope is discussed in "Scope of declarations" on page 151.

The FILE attribute can be implied for a file constant by any of the file description attributes discussed in this chapter. A name can be contextually declared as a file constant through its appearance in the FILE option of any input or output statement, or in an ON statement for any input/output condition.

In the following example, the name Master is declared as a file constant:

```
declare Master file;
```

File variable

A file variable has the attributes FILE and VARIABLE. It cannot have any of the file constant description attributes. File constants can be assigned to file variables. After assignment, a reference to the file variable has the same significance as a reference to the assigned file constant.

The value of a file variable can be transmitted by record-oriented transmission statements. The value of the file variable on the data set might not be valid after transmission.

The VARIABLE attribute is implied under the circumstances described in "VARIABLE attribute" on page 49.

In the following declaration `Account` is declared as a file variable, and `Acct1` and `Acct2` are declared as file constants. The file constants can subsequently be assigned to the file variable.

```
declare Account file variable,
    Acct1 file,
    Acc2 file;
```

For syntax information, refer to "VARIABLE attribute" on page 49.

Specifying a file reference

A file reference can be a file constant, a file variable, or a function reference which returns a value with the FILE attribute. It can be used in the following ways:

*   In a FILE or COPY option
*   As an argument to be passed to a function or subroutine
*   To qualify an input/output condition for ON, SIGNAL, and REVERT statements
*   As the expression in a RETURN statement.

On-units can be established for a file constant through a file variable that represents its value (see "ON-units for file variables" on page 336). In the following example, the statements labelled `L1` and `L2` both specify null ON-units for the same file.

```
dcl F file,
    G file variable;
    G=F;
L1:  on endfile(G);
L2:  on endfile(F);
```

RECORD and STREAM attributes

The RECORD and STREAM usage attributes specify the kind of data transmission used for the file.
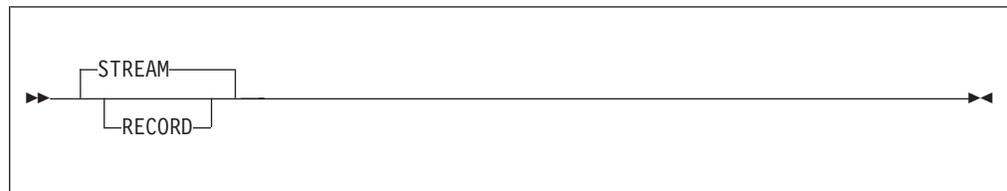
Default: STREAM

RECORD indicates that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable.

STREAM indicates that the data of the file is a continuous stream of data items, in character form, assigned from the stream to variables, or from expressions into the stream.

A file with the STREAM attribute can be specified only in the FILE option of the OPEN, CLOSE, GET, and PUT input/output statements.

A file with the RECORD attribute can be specified only in the FILE option of the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE, and DELETE input/output statements.

INPUT, OUTPUT, and UPDATE attributes

The INPUT, OUTPUT, and UPDATE function attributes specify the direction of data transmission allowed for a file. INPUT specifies that data is transmitted from a data set to the program. OUTPUT specifies that data is transmitted from the program to a data set, either to create a new data set or to extend an existing one. UPDATE, which applies to RECORD files only, specifies that the data can be transmitted in either direction. A declaration of UPDATE for a SEQUENTIAL file indicates the update-in-place mode.

```
            ┌─INPUT──┐
►►─┬────────┼────────┼─────────────────────────────────────────►◄
   │        └─OUTPUT─┘
   └─UPDATE─┘
```
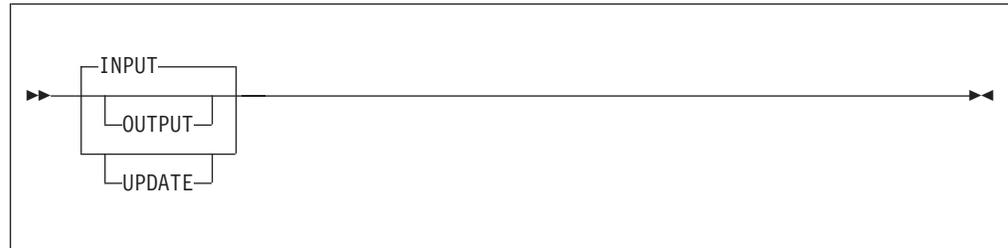
Default: INPUT

SEQUENTIAL and DIRECT attributes

The SEQUENTIAL and DIRECT access attributes apply only to RECORD files, and specify how the records in the file are accessed.

```
   ┌─SEQUENTIAL─┐
►►─┼────────────┼─────────────────────────────────────────────►◄
   └─DIRECT─────┘
```

Abbreviation: SEQL for SEQUENTIAL

Default: SEQUENTIAL

The DIRECT attribute specifies that records in a data set are directly accessed. The location of the record in the data set is determined by a character-string key. Therefore, the DIRECT attribute implies the KEYED attribute. The associated data set must be on a direct-access storage device.

The SEQUENTIAL attribute specifies that records in a consecutive or relative data set are accessed in physical sequence, and that records in an indexed data set are accessed in key sequence order. For certain data set organizations, a file with the SEQUENTIAL attribute can also be used for direct access or for a mixture of random and sequential access. In this case, the file must have the additive attribute KEYED. Existing records of a data set in a SEQUENTIAL UPDATE file can be modified, ignored, or, if the data set is indexed, deleted.

BUFFERED and UNBUFFERED attributes

The buffering attributes apply only to RECORD files.

```
>>─┬──────────────┬───────────────────────────────><
   └─BUFFERED─┘
   └─UNBUFFERED─┘
```
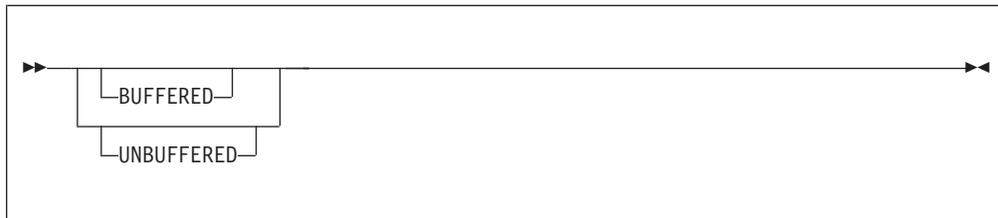
Abbreviations: BUF for BUFFERED, and UNBUF for UNBUFFERED

Defaults: BUFFERED is the default for SEQUENTIAL files. UNBUFFERED is the default for DIRECT files.

The BUFFERED attribute specifies that during transmission to and from a data set, each record of a RECORD file must pass through intermediate storage buffers. This allows both move and locate mode processing.

The UNBUFFERED attribute indicates that a record in a data set need not pass through a buffer but can be transmitted directly to and from the main storage associated with a variable. This allows only move mode processing.

ENVIRONMENT attribute

The characteristic list of the ENVIRONMENT attribute specifies various data set characteristics that are not part of PL/I. For a full description of the characteristics and their uses, refer to the Programming Guide.

Note: Because the characteristics are not part of the PL/I language, using them in a file declaration can limit the portability of your application program.

The following characteristics can be specified on the ENVIRONMENT attribute. For descriptions of the characteristics, refer to the Programming Guide.

BKWD	GRAPHIC	RECSIZE
CONSECUTIVE	KEYLENGTH	REGIONAL(1)
CTLASA	KEYLOC	SCALARVARYING
GENKEY	ORGANIZATION	VSAM

KEYED attribute

The KEYED attribute applies only to RECORD files, and must be associated with indexed and relative data sets. It specifies that records in the file can be accessed using one of the key options (KEY, KEYTO, or KEYFROM) of record I/O statements.

```
►►──KEYED─────────────────────────────────────────────────────►◄
```

The KEYED attribute need not be specified unless one of the key options is used.

PRINT attribute

The PRINT attribute is described in "PRINT attribute" on page 300.

Opening and closing files

Before a file can be used for data transmission, by input or output statements, it must be associated with a data set. Opening a file associates the file with a data set and involves checking for the availability of external media, positioning the media, and allocating required operating system support. When processing is completed, the file must be closed. Closing a file dissociates the file from the data set.

PL/I provides two statements, OPEN and CLOSE, to perform these functions. However, use of these statements is optional. If an OPEN statement is not executed for a file, the file is opened implicitly during the execution of first data transmission statement for that file. In this case, the file opening uses information about the file as specified in a DECLARE statement (and defaults derived from the transmission statement). Similarly, if a file has not been closed before PL/I termination, PL/I will close it during the termination process.

When a file for stream input, sequential input, or sequential update is opened, the associated data set is positioned at the first record.

OPEN statement

The OPEN statement associates a file with a data set. It merges attributes specified on the OPEN statement with those specified on the DECLARE statement. It also completes the specification of attributes for the file, if a complete set of attributes has not been declared for the file being opened.

OPEN
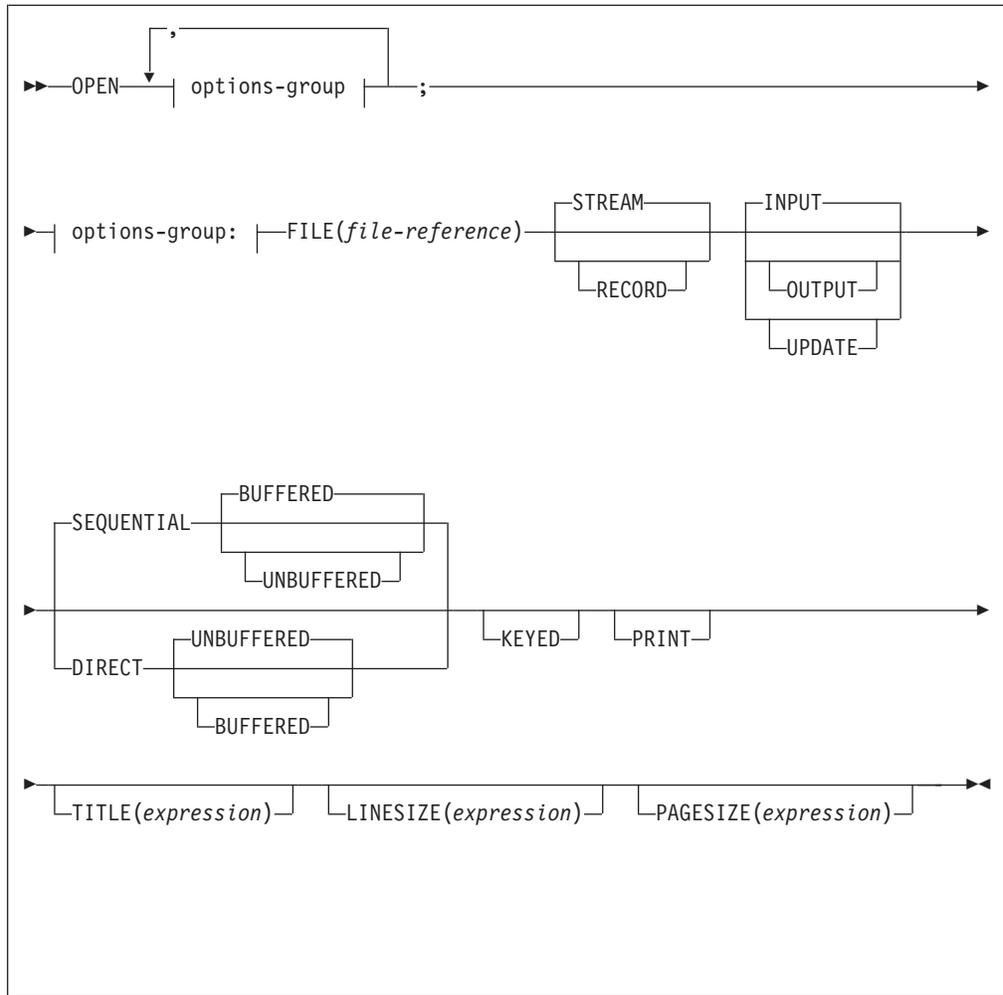
The options of the OPEN statement can appear in any order.

FILE
   Specifies the name of the file that is associated with a data set.

STREAM, RECORD,
INPUT, OUTPUT, UPDATE,
DIRECT, SEQUENTIAL,
BUFFERED, UNBUFFERED,
KEYED, and PRINT
   These options specify attributes that augment the attributes specified in the file
   declaration. The same attributes need not be listed in both OPEN and
   DECLARE statements for the same file. For a list of attributes for record and
   stream input and output, see Table 32 on page 260.

   When a STREAM file is opened, the first GET or PUT statement can specify,
   with a statement option or format item, the first record to be accessed. The
   statement option or format item indicates that *n* lines are skipped before a
   record is accessed. The file is then positioned at the start of the *n*th record. If
   no statement option or format item is encountered, the initial file position is the
   start of the first line or record. If the file has the PRINT attribute, it is physically
   positioned at column 1 of the first line or record.

   Opening a file that is already open does not affect the file.

TITLE

> The content of *expression* determines what is being designated. For more information on the TITLE attribute refer to the Programming Guide.

LINESIZE

> Converted to an integer value, specifies the length in bytes of a line during subsequent operations on the file. New lines can be started by use of the printing and control format items or by options in a GET or PUT statement. If an attempt is made to position a file past the end of a line before explicit action to start a new line is taken, a new line is started, and the file is positioned to the start of this new line. The default line size for PRINT file is 120.

> The LINESIZE option can be specified only for a STREAM OUTPUT file. The line size taken into consideration whenever a SKIP option appears in a GET statement is the line size that was used to create the data set. Otherwise, the line size is taken as the current length of the logical record minus control bytes.

PAGESIZE

> Is evaluated and converted to an integer value, and specifies the number of lines per page. The first attempt to exceed this limit raises the ENDPAGE condition. During subsequent transmission to the PRINT file, a new page can be started by use of the PAGE format item or by the PAGE option in the PUT statement. The default page size is 60.

> The PAGESIZE option can be specified only for a file having the PRINT attribute.

Implicit opening

An implicit opening of a file occurs when a GET, PUT, READ, WRITE, LOCATE, REWRITE, or DELETE statement is executed for a file for which an OPEN statement has not already been executed.

If a GET statement contains a COPY option, execution of the GET statement can cause implicit opening of either the file specified in the COPY option or, if no file was specified, of the output file SYSPRINT. Implicit opening of the file specified in the COPY option implies the STREAM and OUTPUT attributes.

Table 34 shows the attributes that are implied when a given statement causes the file to be implicitly opened:

Table 34. Attributes implied by implicit open

Statement	Implied Attributes
GET	STREAM, INPUT
PUT	STREAM, OUTPUT
READ	RECORD, INPUT[Note]
WRITE	RECORD, OUTPUT[Note]
LOCATE	RECORD, OUTPUT, SEQUENTIAL
REWRITE	RECORD, UPDATE
DELETE	RECORD, UPDATE

Note:

INPUT and OUTPUT are default attributes for READ and WRITE statements only if UPDATE has not been explicitly declared.

When one of the statements listed in Table 34 opens a file implicitly, it is functionally equivalent to using an explicit OPEN statement for the file with the same attributes specified.

There must be no conflict between the attributes specified in a file declaration and the attributes implied as the result of opening the file. For example, the attributes INPUT and UPDATE are in conflict, as are the attributes UPDATE and STREAM.

The implied attributes discussed earlier are applied before the default attributes listed in Table 34 on page 267 are applied. Implied attributes can also cause a conflict. If a conflict in attributes exists after the application of default attributes, the UNDEFINEDFILE condition is raised.

Table 35. Merged and implied attributes

Merged Attributes	Implied Attributes
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD, KEYED
PRINT	OUTPUT, STREAM
KEYED	RECORD

The following two examples illustrate attribute merging for an explicit opening using a file constant and a file variable:

Example of file constant

```
declare Listing file stream;
open file(Listing) print;
```

Attributes after merge caused by execution of the OPEN statement are STREAM and PRINT. Attributes after implication are STREAM, PRINT, and OUTPUT. Attributes after default application are STREAM, PRINT, OUTPUT, and EXTERNAL.

Example of file variable

```
declare Account file variable,
        (Acct1,Acct2)  file
        output;

Account = Acct1;
open file(Account) print;

Account = Acct2;
open file(Account) record unbuf;
```

The file `Acct1` is opened with attributes (explicit and implied) STREAM, EXTERNAL, PRINT, and OUTPUT. The file `Acct2` is opened with attributes RECORD, EXTERNAL, and OUTPUT.

Example of implicit opening

```
declare Master file keyed internal;

read file (Master)
     into (Master_Record)
     keyto(Master_Key);
```

Attributes after merge (from the implicit opening caused by execution of the READ statement) are KEYED, INTERNAL, RECORD, and INPUT. (No additional attributes are implied.) Attributes after default application are KEYED, INTERNAL, RECORD, INPUT, and SEQUENTIAL.

Examples of declarations of file constants

```
declare File3 input direct environment( regional(1) )
```

This declaration specifies three file attributes: INPUT, DIRECT, and ENVIRONMENT. Other implied attributes are FILE (implied by each of the attributes) and RECORD and KEYED (implied by DIRECT). Scope is EXTERNAL, by default. The ENVIRONMENT attributes specifies that the data set is of the REGIONAL(1) organization.

For the previous declaration, all necessary attributes are either stated or implied in the DECLARE statement. None of the stated attributes can be changed (or overridden) in an OPEN statement.

If the declaration is written as shown in the following example, `invntry` can be opened for different purposes.

```
declare invntry file;
```

In the following example, the file attributes are the same as those specified (or implied) in the DECLARE statement in the previous example.

```
open file (Invntry)
    update sequential;
```

The file might be opened in this way, then closed, and then later opened with a different set of attributes. For example, the following OPEN statement allows records to be read with either the KEYTO or the KEY option.

```
open file (Invntry)
    input sequential keyed;
```

Because the file is SEQUENTIAL, the data set can be accessed in a purely sequential manner. It can also be accessed directly by means of a READ statement with a KEY option. A READ statement with a KEY option for a file of this description obtains a specified record. Subsequent READ statements without a KEY option access records sequentially, beginning with the next record in KEY sequence.

CLOSE statement

The CLOSE statement dissociates an opened file from its data set.
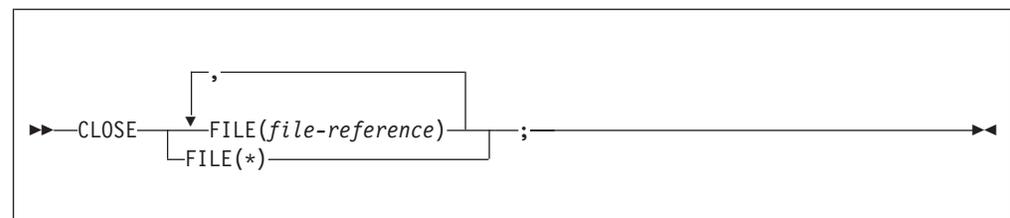
FILE
> Specifies the name of the file that is dissociated from the data set. CLOSE FILE(*) closes all open files.

CLOSE

The CLOSE statement also dissociates from the file all attributes established for it by the implicit or explicit opening process. If desired, new attributes can be specified for the file in a subsequent OPEN statement. However, all attributes explicitly given to the file constant in a DECLARE statement remain in effect.

Closing a file that was previously closed has no effect. A closed file can be reopened. If a file is not closed by a CLOSE statement, it is closed at the termination of the program.

FLUSH statement

The FLUSH statement can be used to flush one or all files.

```
►►──FLUSH──┬──FILE(file-reference)──┬──;──────────────────────►◄
           └──FILE(*)───────────────┘
```
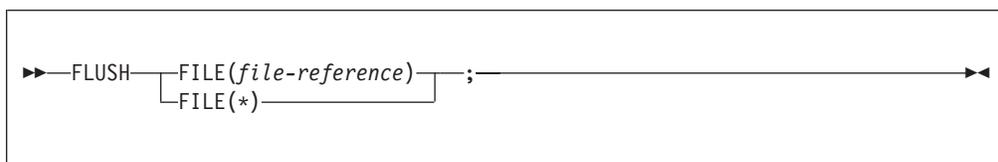
FILE
    Specifies the name of the output file.

The FLUSH statement flushes the buffers associated with an open output file (or with all open output files if * is specified). This normally happens when the file is closed or the program ends, but the FLUSH statement ensures the buffers are flushed before any other processing occurs.

SYSPRINT and SYSIN

Two files are provided that can be used by any PL/I program. One is the input file SYSIN, and the other is the output file SYSPRINT. These files need not be declared or opened explicitly. For SYSIN, the default attributes are STREAM INPUT, and for SYSPRINT they are STREAM OUTPUT PRINT. Both file names, SYSIN and SYSPRINT, have the default attribute EXTERNAL, even though SYSPRINT contains more than 7 characters.

The compiler does not reserve the names SYSIN and SYSPRINT for the specific purposes described above. They can be used for other purposes besides identifiying SYSIN and SYSPRINT files. Other attributes can be applied to them, but the PRINT attribute is applied by default to SYSPRINT when it is declared or opened as a STREAM OUTPUT file unless the INTERNAL attribute is declared for it.

Chapter 12. Record-oriented data transmission

This chapter describes features of the input and output statements used in record-oriented data transmission. Those features of PL/I that apply generally to record-oriented or stream-oriented data transmission, including declaring files, file attributes, and opening and closing files, are described in Chapter 11, "Input and output." For syntax information about the ENVIRONMENT attribute, refer to "ENVIRONMENT attribute" on page 264. For details about environment characteristics and record I/O data transmission statements for each data set organization, refer to the Programming Guide.

In record-oriented data transmission, data in a data set is a collection of records recorded in any format acceptable to the operating system. No data conversion is performed during record-oriented data transmission. On input, the READ statement either transmits a single record to a program variable exactly as it is recorded in the data set, or sets a pointer to the record. On output, the WRITE, REWRITE, or LOCATE statement transmits a single record from a program variable exactly as it is recorded internally. If the information transmitted to the file has a length N which is less than the established record length M, the resulting value of the last M-N bytes of the record is undefined.

Data transmitted

Most variables, including parameters and DEFINED variables, can be transmitted by record-oriented data transmission statements. In general, the information given in this chapter can be applied equally to all variables.

Note: A data aggregate must be in connected storage. If a graphic string is specified for input or output, the SCALARVARYING option must be specified for the file. Other data considerations are described in the following sections.

Unaligned bit strings

The following cannot be transmitted:

- BASED, DEFINED, parameter, subscripted, or structure-base-element variables that are unaligned nonvarying bit strings
- Minor structures whose first or last base elements are unaligned nonvarying bit strings (except where they are also the first or last elements of the containing major structure)
- Major structures that have the DEFINED attribute or are parameters, and that have unaligned nonvarying bit strings as their first or last elements.

Varying length strings

A locate mode output statement (see "LOCATE statement" on page 274) specifying a varying length string transmits a field having a length equal to the maximum length of the string. For VARYINGZ strings, the null terminator is also transmitted. For VARYING strings, a 2-byte prefix denoting the current length of the string is also transmitted; for this, the SCALARVARYING option of the ENVIRONMENT attribute must be specified for the file.

A move mode output statement (see "WRITE statement" on page 273 and "REWRITE statement" on page 273) specifying a varying length string variable transmits only the current length of the string. For VARYINGZ strings, the null terminator is also transmitted. For VARYING strings, a 2-byte prefix is included only if the SCALARVARYING option of the ENVIRONMENT attribute is specified for the file.

Reading and writing using varying length strings allows you to access records that can have undefined or unknown lengths.

Area variables

A locate mode output statement specifying an area variable transmits a field whose length is the declared size of the area, plus a 16-byte prefix containing control information.

A move mode statement specifying an element area variable or a structure whose last element is an area variable transmits only the current extent of the area plus a 16-byte prefix.

Data transmission statements

The data transmission statements that transmit records to or from a data set are READ, WRITE, LOCATE, and REWRITE. The DELETE statement deletes records from an UPDATE file. The attributes of the file determine which data transmission statements can be used. Statement options are described in "Options of data transmission statements" on page 274. For information about variables in data transmission statements, see the Programming Guide.

READ statement

The READ statement can be used with any INPUT or UPDATE file. It either transmits a record from the data set to the program variable or sets a pointer to the record in storage.

```
►►──READ FILE──(file-reference)─┬─IGNORE──(expression)─────────────────┬──►
                                ├─INTO(ref)─┬──────────────────────┬────┤
                                │           ├─KEY(expression)───┤    │
                                │           └─KEYTO(reference)─┘    │
                                └─SET(pointer-ref)─┬──────────────────────┤
                                                   ├─KEY(expression)───┤
                                                   └─KEYTO(reference)─┘

►──;──────────────────────────────────────────────────────────────────────►◄
```
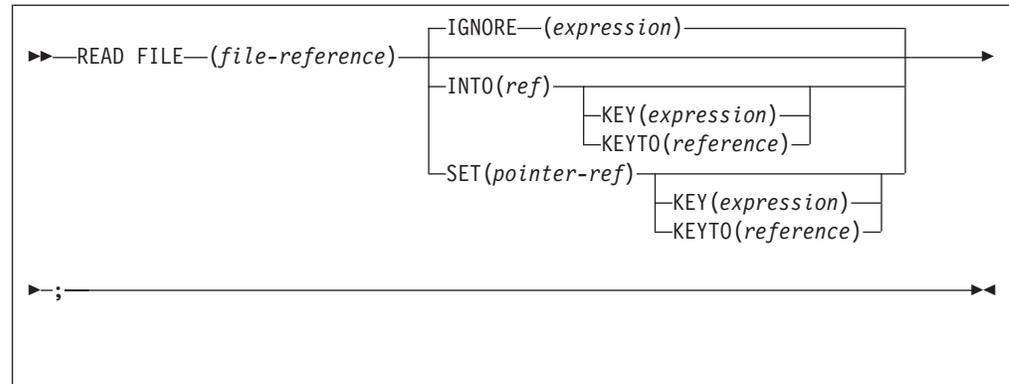
The keywords can appear in any order. A READ statement without an INTO, SET, or IGNORE option is equivalent to a READ with an IGNORE(1).

WRITE statement

The WRITE statement can be used with any OUTPUT file, DIRECT UPDATE file, or SEQUENTIAL UPDATE file. It transmits a record from the program and adds it to the data set.

```
►►──WRITE FILE──(file-reference)──FROM──(reference)────────────────────────►

►─┬──────────────────────────┬──;──────────────────────────────────────────►◄
  ├─KEYFROM(expression)─┤
  └─KEYTO(reference)──────┘
```
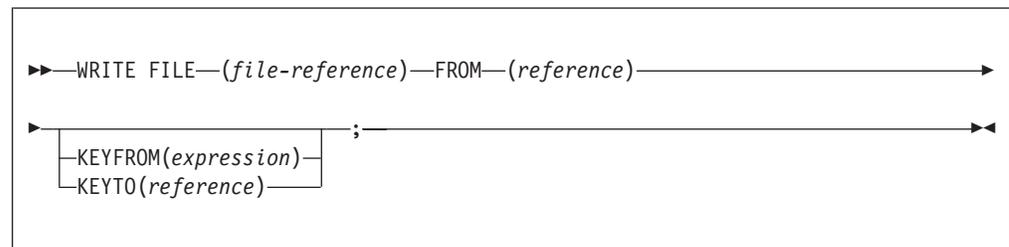
The keywords can appear in any order.

REWRITE statement

The REWRITE statement replaces a record in an UPDATE file. For SEQUENTIAL UPDATE files, the REWRITE statement specifies that the last record read from the file is rewritten; consequently a record must be read before it can be rewritten. For DIRECT UPDATE files, any record can be rewritten whether or not it has first been read.

```
►►──REWRITE FILE──(file-reference)─┬──────────────────────┬────────────────►
                                   └─FROM──(reference)─┘

►─┬──────────────────────┬──;──────────────────────────────────────────────►◄
  └─KEY──(expression)─┘
```
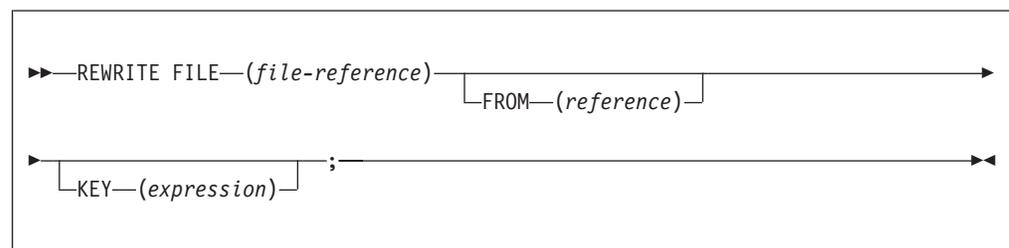
The keywords can appear in any order. The FROM option must be specified for UPDATE files with the DIRECT attribute, or with both the SEQUENTIAL and UNBUFFERED attributes.

A REWRITE statement that does not specify the FROM option has the following effect:

- If the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set.
- If the last record was read by a READ statement with the SET option, the record is updated by whatever assignments were made in the variable identified by the pointer variable in the SET option.

LOCATE statement

The LOCATE statement can be used only with an OUTPUT SEQUENTIAL BUFFERED file for locate mode processing. It allocates storage within an output buffer for a based variable and sets a pointer to the location of the next record. For further description of locate mode processing, see "Locate mode" on page 278.

```
►►──LOCATE──based-variable──FILE──(file-reference)──────────────────────►

►──────────────────────────────────────────────────;──────────────►◄
     └─SET──(pointer-reference)─┘  └─KEYFROM──(expression)─┘
```
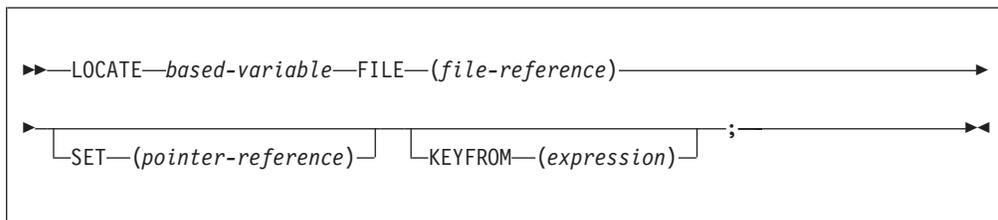
The keywords can appear in any order.

based-variable
  Must be an unsubscripted level-1 based variable.

DELETE statement

The DELETE statement deletes a record from an UPDATE file.

```
►►──DELETE FILE──(file-reference)──────────────────;──────────►◄
                                   └─KEY──(expression)─┘
```
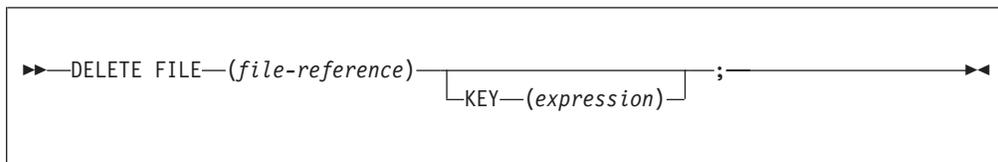
The keywords can appear in any order. If the KEY option is omitted, the record to be deleted is the last record that is read. No subsequent DELETE or REWRITE statement without a KEY is allowed until another READ statement is processed. If the KEY option is included, that record addressed by the key is deleted if found.

Options of data transmission statements

Options that are allowed for record-oriented data transmission statements differ according to the attributes of the file and the characteristics of the associated data set.

FILE option

The FILE option must appear in every record-oriented data transmission statement. It specifies the file upon which the operation takes place. An example of the FILE option is shown in each of the statements in this section. If the file specified is not open in the current process, it is opened implicitly.

FROM option

The FROM option specifies the element or aggregate variable from which the record is written. The FROM option must be used in the WRITE statement for any OUTPUT or DIRECT UPDATE file. It can also be used in the REWRITE statement for any UPDATE file.

If the variable is an aggregate, it must be in connected storage. Certain uses of unaligned nonvarying bit strings are disallowed (for details, see "Data transmitted" on page 271).

The FROM variable can be an element string variable of varying length. When using a WRITE statement with the FROM option, only the current length of a varying length string is transmitted to a data set. For a VARYINGZ string, the null terminator is attached and also transmitted. For a VARYING string, a 2-byte prefix specifying the length is attached only if the SCALARVARYING option of the ENVIRONMENT attribute is specified for the file.

Records are transmitted as an integral number of bytes. If a bit string (or a structure that starts or ends with a bit string) that is not aligned on a byte boundary is transmitted, the record is padded with bits at the start or the end of the string, and the result might be incorrect.

The FROM option can be omitted from a REWRITE statement for SEQUENTIAL UPDATE files. If the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set. If the last record was read by a READ statement with the SET option, the record (updated by whatever assignments were made) is copied back onto the data set.

In the following examples, the statements specify that the value of the variable `Mas_Rec` is written into the output file `Master`.

```
  write file (Master) from (Mas_Rec);
```

The REWRITE statement specifies that `Mas_Rec` replaces the last record read from an UPDATE file.

```
  rewrite file (Master) from (Mas_Rec);
```

IGNORE option

The IGNORE option can be used in a READ statement for any SEQUENTIAL INPUT or SEQUENTIAL UPDATE file.

The expression in the IGNORE option is evaluated and converted to an integer value n. If n is greater than zero, n records are ignored. A subsequent READ statement for the file will access the (n+1)th record. If n is less than 1, the READ statement has no effect.

The following example specifies that the next three records in the file are to be ignored:

```
  read file (In) ignore (3);
```

INTO option

The INTO option specifies an element or aggregate variable into which the logical record is read. The INTO option can be used in the READ statement for any INPUT or UPDATE file.

INTO

If the variable is an aggregate, it must be in connected storage. Certain uses of unaligned nonvarying bit strings are disallowed (for details, see "Data transmitted" on page 271).

The INTO variable can be an element string variable of varying length. For VARYINGZ strings, each record contains a null terminator. For VARYING strings, if the SCALARVARYING option of the ENVIRONMENT attribute was specified for the file, each record contains a 2-byte prefix that specifies the length of the string data.

If SCALARVARYING was not declared on input, the string length is calculated from the record length and attached as a 2-byte prefix (for VARYING strings). For VARYING bit strings, this calculation rounds up the length to a multiple of 8 and therefore the calculated length might be greater than the maximum declared length.

The following example specifies that the next sequential record is read into the variable RECORD_1:

```
read file (Detail) into (Record_1);
```

KEY option

The KEY option specifies a character, graphic or widechar key that identifies a record. It can be used in a READ statement for an INPUT or UPDATE file, or in a REWRITE statement for a DIRECT UPDATE file.

The KEY option applies only to KEYED files. The KEY option is required if the file has the DIRECT attribute and optional if the file has the SEQUENTIAL and KEYED attributes.

The expression in the KEY option is evaluated and, if not character, graphic or widechar, is converted to a character value that represents a key. It is this character, graphic or widechar value that determines which record is read.

The following example specifies that the record identified by the character value of the variable Stkey is read into the variable Item:

```
read file (Stpck) into (Item) key (Stkey);
```

KEYFROM option

The KEYFROM option specifies a character, graphic or widechar key that identifies the record on the data set to which the record is transmitted. It can be used in a WRITE statement for any KEYED OUTPUT or DIRECT UPDATE file, or in a LOCATE statement.

The KEYFROM option applies only to KEYED files. The expression is evaluated and, if not character, graphic or widechar, is converted to a character string and is used as the key of the record when it is written.

Relative data sets can be created using the KEYFROM option. The record number is specified as the key.

REGIONAL(1) data sets can be created using the KEYFROM option. The region number is specified as the key.

For indexed data sets, KEYFROM specifies a recorded key whose length must be equal to the key length specified for the data set.

The following example specifies that the value of `Loanrec` is written as a record in the file `Loans`, and that the character string value of `Loanno` is used as the key with which it can be retrieved:

```
write file (Loans) from (Loanrec) keyfrom (Loanno);
```

KEYTO option

The KEYTO option specifies the character, graphic or widechar variable to which the key of a record is assigned. The KEYTO option can specify any string pseudovariable other than STRING. It cannot specify a variable declared with a numeric picture specification. The KEYTO option can be used in a READ statement for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file.

The KEYTO option applies only to KEYED files.

Assignment to the KEYTO variable always follows assignment to the INTO variable. If an incorrect key specification is detected, the KEY condition is raised. The value assigned is as follows:

- For indexed data sets, the record key is padded or truncated on the right to the declared length of the character variable.
- For *relative* data sets, a record number is converted to a character string with leading zeros suppressed, truncated, or padded on the left to the declared length of the character variable.
- For REGIONAL(1) data sets, the 9-character region-number, padded or truncated on the left to the declared length of the character variable. If the character variable is of varying length, any leading zeros in the region number are truncated and the string length is set to the number of significant digits. An all-zero region number is truncated to a single zero.

The KEY condition is not raised for this type of padding or truncation.

The following example specifies that the next record in the file `Detail` is read into the variable `Invntry`, and that the key of the record is assigned to the variable `Keyfld`:

```
read file (Detail) into (Invntry) keyto (Keyfld);
```

SET option

The SET option can be used with a READ statement or a LOCATE statement. For the READ statement, it specifies a pointer variable that is set to point to the record read. For the LOCATE statement, it specifies a pointer variable that is set to point to the next record for output.

If the SET option is omitted for the LOCATE statement, the pointer declared with the record variable is set. If a VARYING string is transmitted, the SCALARVARYING option must be specified for the file.

The following example specifies that the value of the pointer variable `P` is set to the location in the buffer of the next sequential record:

```
read file (X) set (P);
```

Processing modes

Record-oriented data transmission has two modes of handling data:

Move mode   processes data by moving it into or out of the variable.

Locate mode processes data while it remains in a buffer. The execution of a data transmission statement assigns a pointer variable for the location of the storage allocated to a record in the buffer. Locate mode is applicable only to BUFFERED files.

The data transmission statements and options that you specify determine the processing mode used.

Move mode

In move mode, a READ statement transfers a record from the data set to the variable named in the INTO option. A WRITE or REWRITE statement transfers a record from the variable named in the FROM option to the data set. The variables named in the INTO and FROM options can be of any storage class.

The following is an example of move mode input:

```
Eof_In = '0'b;
on endfile(In) Eof_In = '1'B;
read file(In) into(Data);
do while (¬Eof_In);
      ⋮
      /* process record  */
read file(In) into(Data);
end;
```

Locate mode

Locate mode assigns to a pointer variable the location of the buffer. A based variable described the record. The same data can be interpreted in different ways by using different based variables. Locate mode can also be used to read self-defining records, in which information in one part of the record is used to indicate the structure of the rest of the record. For example, this information could be an array bound or a code identifying which based structure should be used for the attributes of the data.

A READ statement with a SET option sets the pointer variable in the SET option to a buffer containing the record. The data in the record can then be referenced by a based variable qualified with the pointer variable.

The pointer value is valid only until the execution of the next READ or CLOSE statement that refers to the same file.

The pointer variable specified in the SET option or, if SET was omitted, the pointer variable specified in the declaration of the based variable, is used. The pointer value is valid only until the execution of the next LOCATE, WRITE, or CLOSE statement that refers to the same file. It also initializes components of the based variable that have been specified in REFER options.

The LOCATE statement sets a pointer variable to a large enough area where the next record can be built.

After execution of the LOCATE statement, values can be assigned directly into the based variables qualified by the pointer variable set by the LOCATE statement.

The following example shows locate mode input:

```
dcl 1 Data based(P),
 2
 .
 .
 .
 ;

on endfile(In)
 ;
read file(In) set(P);
do while (¬endfile(In));
 .
 .
                  /* process record */
read file(In) set(P);
end;
```

The following example shows locate mode output:

```
dcl 1 Data based(P);
 2
 .
 .
 .
 ;

do while (More_records_to_write);
locate Data file(Out);
 .
 .
          /* build record */
end;
```

Locate mode

Chapter 13. Stream-oriented data transmission

This chapter describes the input and output statements used in stream-oriented data transmission. Features that apply to stream-oriented and record-oriented data transmission, including files, file attributes, and opening and closing files, are described in Chapter 11, "Input and output," on page 257.

Stream-oriented data transmission treats a data set as a continuous stream of data values in character, graphic, or mixed character data form. Within a program, record boundaries are generally ignored. However, a data set consists of a series of lines of data, and each data set created or accessed by stream-oriented data transmission has a line size associated with it. In general, a line is equivalent to a record in the data set, but the line size does not necessarily equal the record size.

The stream-oriented data transmission statements can also be used for internal data movement, by specifying the STRING option instead of specifying the FILE option. Although the STRING option is not an input/output operation, its use is described in this chapter.

Stream-oriented data transmission can be list-directed, data-directed, or edit-directed.

List-directed data transmission
> transmits the values of data-list items without your having to specify the format of the values in the stream. The values are recorded externally as a list of constants, separated by blanks or commas.

Data-directed data transmission
transmits the names of the data-list items, as well as their values, without your having to specify the format of the values in the stream. The GRAPHIC option of the ENVIRONMENT attribute must be specified if any variable name contains a DBCS character, even if no DBCS data is present.

Edit-directed data transmission
transmits the values of data-list items and requires that you specify the format of the values in the stream. The values are recorded externally as a string of characters or graphics to be treated character by character (or graphic by graphic) according to a format list.

The following sections detail the data transmission statements and their options, and how to specify the list-, data-, and edit-directed data. How to accommodate double-byte characters is discussed in "DBCS data in stream I/O" on page 301.

Data transmission statements

Stream-oriented data transmission uses GET and PUT statements. Only consecutive files can be processed with the GET and PUT statements.

The variables or pseudovariables to which data values are assigned, and the expressions from which they are transmitted, are generally specified in a data-specification with each GET or PUT statement. The statements can also include options that specify the origin or destination of the data values or indicate where they appear in the stream relative to the preceding data values. Options for the stream-data transmission statements are described in "Options of data transmission statements" on page 283.

GET statement

The GET statement is a STREAM input data transmission statement that can either:
- Assign data values from a data set to one or more variables
- Assign data values from a string to one or more variables.

For a stream input file, use the following syntax for the GET statement.

```
▶▶─GET──────────────────────────────────────────────────────────────────▶
          └─FILE──(expression)─┘  └─data-specification─┘

▶──┬────────────────────────────┬──┬──────────────────────┬──;──────────▶◀
   └─COPY──┬──────────────────┬─┘  └─SKIP──┬────────────┬─┘
           └─(file-reference)─┘            └─(expression)─┘
```
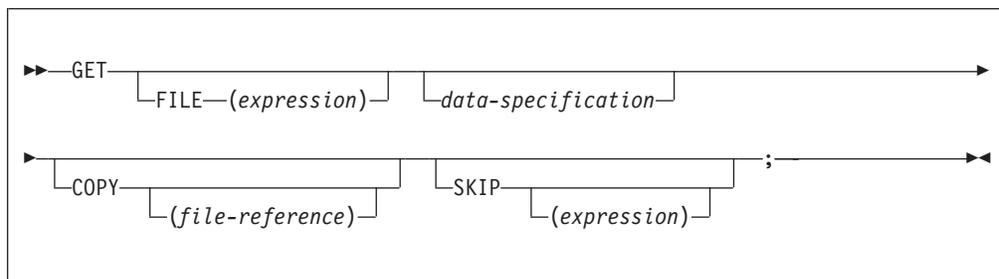
The keywords can appear in any order. The data specification must appear unless the SKIP option is specified.

For transmission from a string, use this syntax for the GET statement.

```
►►──GET STRING──(expression)──data-specification──;──────────────────►◄
```

If FILE or STRING option is not specified FILE(SYSIN) is assumed and SYSIN is implicitly declared FILE STREAM INPUT EXTERNAL.

PUT statement

The PUT statement is a STREAM output data transmission statement that can:
* Transmit values to a stream output file
* Assign values to a character variable.

Use the following syntax of the PUT statement when dealing with stream output files.

```
►►──PUT──────────────────────────────────────────────────────────────►
        └─FILE──(file-reference)─┘  └─data-specification─┘

►──────────────────────────────────────;───────────────────────────►◄
   ┌─PAGE─────────────────────┐
   │      └─LINE──(expression)─┘
   ├─SKIP─────────────────────┤
   │      └─(expression)───────┘
   └─LINE──(expression)────────┘
```
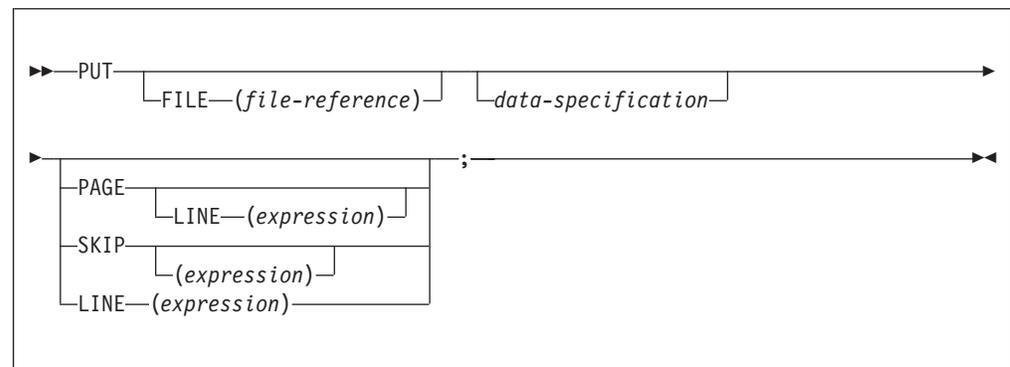
The keywords can appear in any order. The data specification can be omitted only if one of the control options (PAGE, SKIP, or LINE) appears.

For transmission to a character string, however, use this syntax of the PUT statement.

```
►►──PUT STRING──(expression)──data-specification──;──────────────────►◄
```

Options of data transmission statements

COPY option

The COPY option specifies that the source data stream is written on the specified STREAM OUTPUT file without alteration. If no file reference is given, the default is the output file SYSPRINT. Each new record in the input stream starts a new record on the COPY file. For example:

```
get file(sysin) data(A,B,C) copy(DPL);
```

not only transmits the values assigned to A, B, and C in the input stream to the variables with these names, but also writes them exactly as they appear in the input stream, on the file DPL. Data values that are skipped on input, and not transmitted to internal variables, copy intact into the output stream.

If a condition is raised during the execution of a GET statement with a COPY option and an ON-unit is entered in which another GET statement is executed for the same file, and if control is returned from the ON-unit to the first GET statement, that statement executes as if no COPY option was specified. If, in the ON-unit, a PUT statement is executed for the file associated with the COPY option, the position of the data transmitted might not immediately follow the most recently-transmitted COPY data item.

If the COPY option file is not open in the current program, the file is implicitly opened in the program for stream output transmission.

Data specification options

Data specifications in GET and PUT statements specify the data to be transmitted.
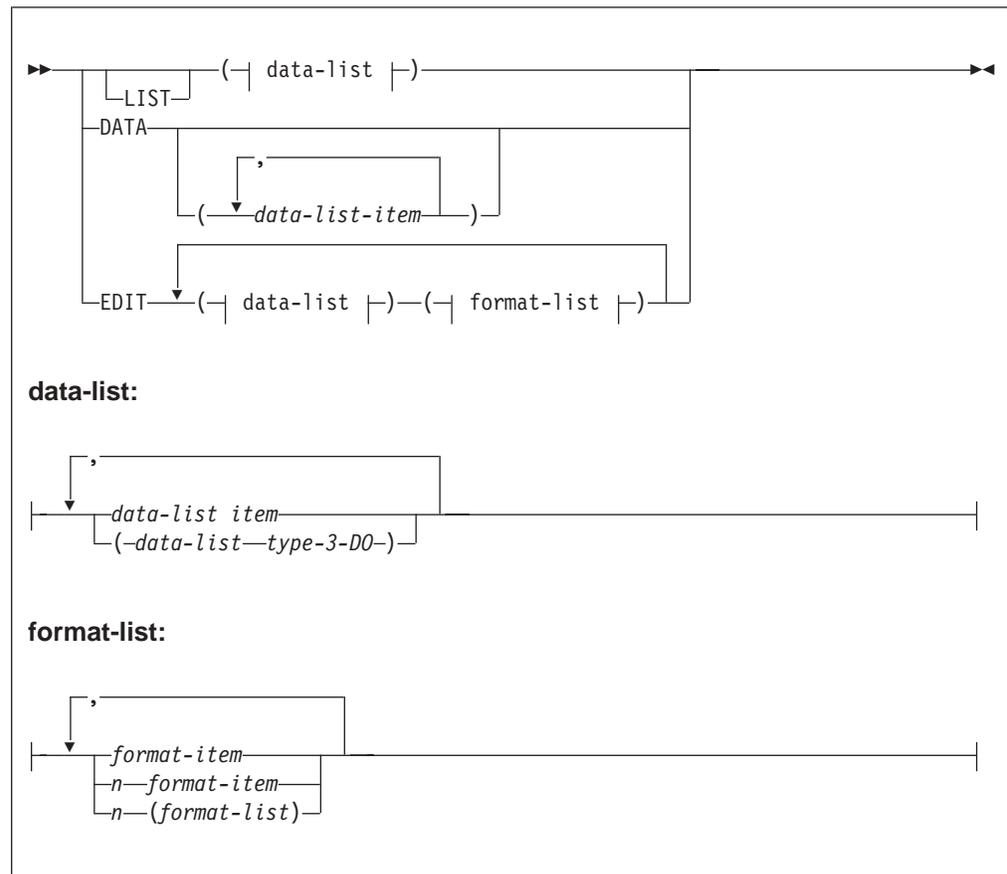
If a GET or PUT statement includes a data list that is not preceded by one of the keywords LIST, DATA, or EDIT, LIST is the default.

Important: In a statement without LIST, DATA, or EDIT preceding the data list, the data list must **immediately** follow the GET or PUT keyword. Any options required must be specified after the data list.

DATA   Refer to "Data-directed data specification" on page 289.

EDIT   Refer to "Edit-directed data specification" on page 293.

LIST   Refer to "List-directed data specification" on page 297.

data-list item
>   On input, a data-list item for edit-directed and list-directed transmission can

be one of the following: an element, array, or structure variable. For a data-directed data specification, a data-list item can be an element, array, or structure variable. None of the names in a data-directed data list can be subscripted or locator-qualified. However, qualified (that is, structure-member) or string-overlay-defined names are allowed.

On output, a data list item for edit-directed and list-directed data specifications can be an element expression, an array expression, or a structure expression. For a data-directed data specification, a data-list item can be an element, array, or structure variable. It must not be locator-qualified. It can be qualified (that is, a member of a structure) or string-overlay-defined.

The data types of a data-list item can be any computational data, and in PUT statements, the data type may also be POINTER, HANDLE, OFFSET, ENTRY, FILE or LABEL. If the data type is one of these non-computational types, then the contents of the item will be transmitted via its heximage (and for PUT DATA, the heximage will be enclosed in quotes followed by a suffix of BX).

An array or structure variable in a data-list is equivalent to *n* items in the data list, where *n* is the number of element items in the array or structure. For edit-directed transmission, each element item is associated with a separate use of a data-format item.

data-list type-3-DO

The syntax for the Type 3 DO specification is described under "DO statement" on page 195. Data list items with Type 3 DO specifications are not allowed in data-directed data lists for GET statements.

When the last repetitive specification is completed, processing continues with the next data-list item.

Each repetitive specification must be enclosed in parentheses, as shown in the syntax diagram. If a data specification contains only a repetitive specification, two sets of outer parentheses are required, since the data list is enclosed in parentheses and the repetitive specification must have a separate set.

When repetitive specifications are nested, the rightmost DO is at the outer level of nesting. For example:

```
get list (((A(I,J)
     do I = 1 to 2)
     do J = 3 to 4));
```

There are three sets of parentheses, in addition to the set used to delimit the subscripts. The outermost set is the set required by the data specification. The next set is that required by the outer repetitive specification. The third set of parentheses is required by the inner repetitive specification.

This statement is equivalent in function to the following nested do-groups:

```
do J = 3 to 4;
  do I = 1 to 2;
    get list (A (I,J));
  end;
end;
```

It assigns values to the elements of the array A in the following order:

```
A(1,3), A(2,3), A(1,4), A(2,4)
```

format list
      For a description of the format list, see "Edit-directed data specification" on page 293.

FILE option

The FILE option specifies the file upon which the operation takes place. It must be a STREAM file. For information on how to declare a file type data item, see "Files" on page 259.

If neither the FILE option nor the STRING option appears in a GET statement, the input file SYSIN is the default; if neither option appears in a PUT statement, the output file SYSPRINT is the default.

LINE option

The LINE option can be specified only for PRINT files. The LINE option defines a new current line for the data set. The expression is evaluated and converted to an integer value, *n*. The new current line is the *n*th line of the current page. If at least *n* lines have already been written on the current page or if *n* exceeds the limits set by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised. If *n* is less than or equal to zero, a value of 1 is used. If *n* specifies the current line, ENDPAGE is raised except when the file is positioned on column 1, in which case the effect is the same as if a SKIP(0) option were specified.

The LINE option takes effect before the transmission of any values defined by the data specification (if any). If both the PAGE option and the LINE option appear in the same statement, the PAGE option is applied first. For example:

```
put file(List) data(P,Q,R) line(34) page;
```

prints the values of the variables P, Q, and R in data-directed format on a new page, commencing at line 34.

For the effect of the LINE option when specified in the first GET statement following the opening of the file, see "OPEN statement" on page 265.

For output to a terminal in interactive mode, the LINE option skips three lines.

PAGE option

The PAGE option can be specified only for PRINT files. It defines a new current page within the data set. If PAGE and LINE appear in the same PUT statement, the PAGE option is applied first. The PAGE option takes effect before the transmission of any values defined by the data specification (if any).

The page remains current until the execution of a PUT statement with the PAGE option, until a PAGE format item is encountered, or until the ENDPAGE condition is raised, resulting in the definition of a new page. A new current page implies line one.

For output to a terminal in interactive mode, the PAGE option skips three lines.

SKIP option

The SKIP option specifies a new current line (or record) within the data set. The expression is evaluated and converted to an integer value, *n*. The data set is positioned to the start of the *n*th line (record) relative to the current line (record). If *expression* is not specified, the default is SKIP(1).

The SKIP option takes effect before the transmission of values defined by the data specification (if any). For example:

```
put list(X,Y,Z) skip(3);
```

prints the values of the variables X, Y, and Z on the output file SYSPRINT commencing on the third line after the current line.

For non-PRINT files and input files, if the expression in the SKIP option is less than or equal to zero, a value of 1 is used. For PRINT files, if *n* is less than or equal to zero, the positioning is to the start of the current line.

For the effect of the SKIP option when specified in the first GET statement following the opening of the file, see "OPEN statement" on page 265.

If fewer than *n* lines remain on the current page when a SKIP(n) is issued, ENDPAGE is raised.

When printing at a terminal in conversational mode, SKIP(n) with *n* greater than 3 is equivalent to SKIP(3). No more than three lines can be skipped.

STRING option

The STRING option in GET and PUT statements transmits data between main storage locations rather than between the main and a data set. DBCS data items cannot be used with the STRING option.

The GET statement with the STRING option specifies that data values assigned to the data list items are obtained from the expression, after conversion to character string. Each GET operation using this option always begins at the leftmost character position of the string. If the number of characters in this string is less than the total number of characters specified by the data specification, the ERROR condition is raised.

The PUT statement with the STRING option specifies that values of the data-list items are to be assigned to the specified character variable or pseudovariable. The PUT operation begins assigning values at the leftmost character position of the string, after appropriate conversions are performed. Blanks and delimiters are inserted as in normal I/O operations. If the string is not long enough to accommodate the data, the ERROR condition is raised.

The NAME condition is not raised for a GET DATA statement with the STRING option. Instead, the ERROR condition is raised for situations that raise the NAME condition for a GET DATA statement with the FILE option.

The following restrictions apply to the STRING option:
- The COLUMN control format option cannot be used with the STRING option.
- No pseudovariables are allowed in the STRING option of a PUT statement.

The STRING option is most useful with edit-directed transmission. It allows data gathering or scattering operations performed with a single statement, and it allows stream-oriented processing of character strings that are transmitted by record-oriented statements.

For example:

```
read file (Inputr) into (Temp);
get string(Temp) edit (Code) (F(1));
If Code = 1 then
   get string (Temp) Edit (X,Y,Z)
      (X(1), 3 F(10,4));
```

The READ statement reads a record from the input file `Inputr`. The first GET statement uses the STRING option to extract the code from the first byte of the record and assigns it to `Code`. If the code is 1, the second GET statement uses the STRING option to assign the values in the record to `X`, `Y`, and `Z`. The second GET statement specifies that the first character in the string `Temp` is ignored (the `X(1)` format item in the format list). This ignored character is the same one assigned to `Code` by the first GET statement.

An example of the STRING option in a PUT statement is:

```
put string (Record) edit
   (Name)      (X(1), A(12))
   (Pay#)      (X(10), A(7))
   (Hours*Rate) (X(10), P'$999V.99');

write file (Outprt) from (Record);
```

The PUT statement specifies, by the `X(1)` spacing format item, that the first character assigned to the character variable is a single blank, which is the ANS vertical carriage positioning character that specifies a single space before printing. Following that, the values of the variables `Name` and `Pay#` and of the expression `Hours*Rate` are assigned. The WRITE statement specifies that record transmission is used to write the record into the file `Outprt`.

The variable referenced in the STRING option should not be referenced by name or by alias in the data list. For example:

```
declare S char(8) init('YYMMDD');
put string (S) edit
   (substr (S, 3, 2), '/',
   substr (S, 5, 2), '/',
   substr (S, 1, 2))
   (A);
```

The value of `S` after the PUT statement is 'MM/bb/MM' and not 'MM/DD/YY' because `S` is blanked after the first data item is transmitted. The same effect is obtained if the data list contains a variable based or defined on the variable specified in the STRING option.

Transmission of data-list items

If a data-list item is of complex mode, the real part is transmitted before the imaginary part.

If a data-list item is an array expression, the elements of the array are transmitted in row-major order; that is, with the rightmost subscript of the array varying most frequently.

If a data-list item is a structure expression, the elements of the structure are transmitted in the order specified in the structure declaration.

For example, the statements

```
declare 1 A (10),
        2 B,
        2 C;
put file(X) list(A);
```

result in the output being ordered as follows:

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

If, however, the declaration is:

```
declare 1 A,
        2 B(10),
        2 C(10);
```

the same PUT statement results in the output ordered as follows:

```
A.B(1) A.B(2) A.B(3) ... A.B(10)
A.C(1) A.C(2) A.C(3) ... A.C(10)
```

If an input statement for list- or edit-directed transmission assigns a value to a variable in a data list, the assigned value is used if the variable appears in a later reference in the data list. For example:

```
get list (N,(X(I) do I=1 to N),J,K,);
    substr (Name, J,K));
```

When this statement is executed, values are transmitted and assigned in the following order:

1. A new value is assigned to N.
2. Elements are assigned to the array X as specified in the repetitive specification in the order X(1),X(2),...X(N), with the new value of N specifying the number of assigned items.
3. A new value is assigned to J.
4. A new value is assigned to K.

Data-directed data specification

For a description of the syntax of the DATA data specification, refer to "Data specification options" on page 284.

Names of structure elements in the data-list item need only have enough qualification to resolve any ambiguity. Full qualification is not required.

Omission of the data list results in a default data list that contains all computational variables that could be named in a data-directed statement.

On output, all items in the data list are transmitted.

Restrictions on data-directed data

Subscripted variables are not allowed in data-directed input.

Data-directed data specification

References to based variables in a data-list for data-directed input/output cannot be explicitly locator qualified. For example:

```
dcl Y based(Q), Z based;
put data(Y);
```

The variable *Z* cannot be transmitted since it must be explicitly qualified by a locator.

A based variable in the data-list has the following restrictions:

*   The variable must not be based on an OFFSET variable.
*   The pointer on which the variable is based must not be in DEFINED storage.
*   If the pointer on which the variable is based is itself BASED, then the chain of basing pointers must end with a pointer that is neither BASED nor DEFINED.

A defined variable in the data-list must:

*   Be a picture or character variable
*   Not be defined on a controlled variable
*   Not be defined on an element or cross section of an array
*   Not be defined with a nonconstant POSITION attribute

Typed structures can not be used in data-directed input/output statements.

Syntax of data-directed data

The stream associated with data-directed data transmission is in the form of a list of element assignments. The element assignments that have optionally signed constants, like variable names and equal signs, are in character or graphic form.
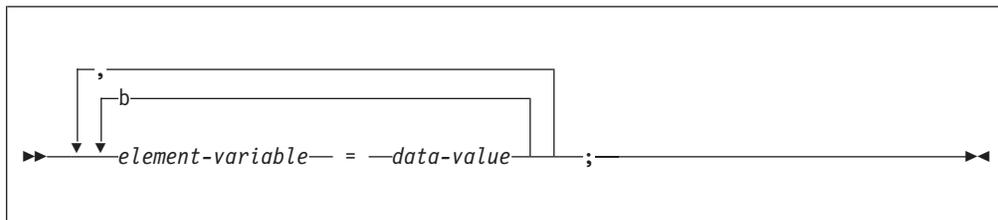
On input, the element assignments can be separated by either a blank or a comma. Blanks can surround periods in qualified names, subscripts, subscript parentheses, and the assignment symbols. On output, the assignments are separated by a blank. For PRINT files, items are separated according to program tab settings.

Each data-value in the stream has one of the syntaxes described for list-directed transmission. For a description of list-directed transmission syntax, refer to "Syntax of list-directed data" on page 298.

The length of the data value in the stream is a function of the attributes declared for the variable and, because the name is also included, the length of the fully qualified subscripted name. The length for output items converted from coded arithmetic data, numeric character data, and bit-string data is the same as that for list-directed output data, and is governed by the rules for data conversion to character type as described in Chapter 5, "Data conversion."

Qualified names in the input stream must be fully qualified.

Interleaved subscripts cannot appear in qualified names in the stream. For example, assume that `Y` is declared as follows:

```
declare 1 Y(5,5),
         2 A(10),
           3 B,
           3 C,
           3 D;
```

An element name has to appear in the stream as follows:

```
 Y.A.B(2,3,8)= 8.72
```

GET data-directed

For more information about the GET statement, see "GET statement" on page 282.

If a data list is used, each data-list item must be an element, array, or structure variable. Names cannot be subscripted, but qualified names are allowed in the data list. All names in the stream should appear in the data list; however, the order of the names need not be the same, and the data list can include names that do not appear in the stream.

If the data list contains a name that is not included in the stream, the value of the named variable remains unchanged.

If the stream contains an unrecognizable element-variable or a name that does not have a counterpart in the data list, the NAME condition is raised.

Transmission ends when a semicolon that is not enclosed in quotation marks or an end-of-file is reached. The recognition of the semicolon or end-of-file determines the number of element assignments that are actually transmitted by a particular statement, whether or not a data list is specified.

For example, consider the following data list, where `A`, `B`, `C`, and `D` are names of element variables:

```
 Data (B, A, C, D)
```

This data list can be associated with the following input data stream:

```
 A= 2.5, B= .0047, D= 125, Z= 'ABC';
```

Because `C` appears in the data list but not in the stream, its value remains unaltered. `Z`, which is not in the data list, raises the NAME condition.

If the data list includes the name of an array, subscripted references to that array can appear in the stream although subscripted names cannot appear in the data list. The entire array need not appear in the stream; only those elements that actually appear in the stream are assigned. If a subscript is out of range, or is missing, the NAME condition is raised.

For example:

```
 declare X (2,3);
```

Consider the following data list and input data stream:

Data Specification
               Input Data Stream

data (X)   `X(1,1)= 7.95,`

```
                              X(1,2)= 8085,

                              X(1,3)= 73;
```

Although the data list has only the name of the array, the input stream can contain values for individual elements of the array. In this case, only three elements are assigned; the remainder of the array is unchanged.

If the data list includes the names of structures, minor structures, or structure elements, fully qualified names must appear in the stream, although full qualification is not required in the data list. For example:

```
dcl 1 In,
      2 Partno,
      2 Descrp,
      2 Price,
         3 Retail,
         3 Whsl;
```

If it is desired to read a value for `In.Price.Retail`, the input data stream must have the following form:

```
In.Price.Retail=1.23;
```

The data specification can be any of:

```
data(In)
data(Price)
data(In.Price)
data(Retail)
data(Price.Retail)
data(In.Retail)
data(In.Price.Retail)
```

PUT data-directed

For more information about the PUT statement, see "PUT statement" on page 283.

A data-list item can be an element, array, or structure variable, or a repetitive specification. The names appearing in the data list, together with their values, are transmitted in the form of a list of element assignments separated by blanks and terminated by a semicolon. For PRINT files, items are separated according to program tab settings; see "PRINT attribute" on page 300.

A semicolon is written into the stream after the last data item transmitted by each PUT statement.

Names are transmitted as a mixed string, which can contain SBCS and/or DBCS characters. Any SBCS characters expressed in DBCS form are first translated to SBCS. For example:

```
put data (.AB.Ckk);
```

would be transmitted as:

```
ABCkk=value-of-variable
```

Note: In the previous example, `.AB.Ckk` is a scalar variable.

Data-directed output is not valid for subsequent data-directed input when the character-string value of a numeric character variable does not represent a valid optionally signed arithmetic constant, or a complex expression.

For character data, the contents of the character string are written out enclosed in quotation marks. Each quotation mark contained within the character string is represented by two successive quotation marks.

The following example shows data-directed transmission (both input and output):

```
declare (A(6), B(7)) fixed;
get file (X) data (B);
do I  = 1 to 6;
     A (I) = B (I+1) + B (I);
end;
put file (Y) data (A);
```

input stream:

```
B(1)=1, B(2)=2, B(3)=3,
B(4)=1, B(5)=2, B(6)=3, B(7)=4;
```

output stream:

```
A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
A(5)= 5 A(6)= 7;
```

In the following example:

```
dcl 1 A,
      2 B FIXED,
      2 C,
        3 D FIXED;
A.B = 2;
A.D = 17;
put data (A);
```

The data fields in the output stream are as follows:

```
A.B= 2 A.C.D= 17;
```

Edit-directed data specification

For information on the syntax of the EDIT data specification, refer to "Data specification options" on page 284.
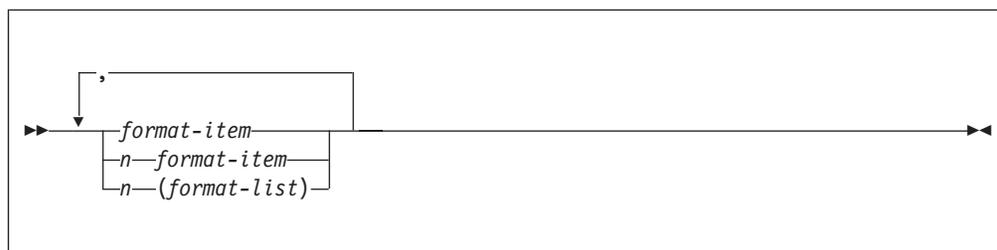
n   Specifies an iteration factor, which is either an expression enclosed in parentheses or an integer. If it is the latter, a blank must separate the integer and the following format item.

The iteration factor specifies that the associated format item or format list is used *n* successive times. A zero or negative iteration factor specifies that the associated format item or format list is skipped and not used (the data-list item is associated with the next data-format item).

If an expression is used to represent the iteration factor, it is evaluated and converted to an integer, once for each set of iterations.

Edit-directed data specification

The associated format item or format list is that item or list of items immediately to the right of the iteration factor.

format item
Specifies either a data-format item, a control-format item, or the remote format item. Syntax and detailed discussions of the format items appear in Chapter 14, "Edit-directed format items."

Data-format items
describe the character or graphic representation of a single data item. They are:

A	character
B	bit
C	complex
E	floating point
F	fixed point
G	graphic
L	line
P	picture

Control-format items
specify the layout of the data set associated with a file. They are:

> COLUMN
> LINE
> PAGE
> SKIP
> X

Remote-format item
specifies a label reference whose value is the label constant of a FORMAT statement located elsewhere. The FORMAT statement contains the remotely situated format items. The label reference item is:

R(label-reference)
Where label is the label constant name of the FORMAT statement. For information on specifying the R-format item, see "R-format item" on page 312.

The first data-format item is associated with the first data-list item, the second data-format item with the second data-list item, and so on. If a format list contains fewer data-format items than there are items in the associated data list, the format list is reused. If there are excessive format items, they are ignored.

Suppose a format list contains five data-format items and its associated data list specifies ten items to be transmitted. The sixth item in the data list is associated with the first data-format item, and so forth. Suppose a format list contains ten data-format items and its associated data list specifies only five items. The sixth through the tenth format items are ignored.

If a control-format item is encountered, the control action is executed.

The PAGE and LINE control-format items can be used only with PRINT files and, consequently, can appear only in PUT statements. The SKIP, COLUMN, and X-format items apply to both input and output.

The PAGE, SKIP, and LINE format items have the same effect as the corresponding options of the PUT statement (and of the GET statement, in the case of SKIP),

except that the format items take effect when they are encountered in the format list, while the options take effect before any data is transmitted.

The COLUMN format item cannot be used in a GET STRING or PUT STRING statement.

For the effects of control-format items when specified in the first GET or PUT statement following the opening of a file, see "OPEN statement" on page 265.

A value read into a variable can be used in a format item that is associated with another variable later in the data list.

```
get edit (M,String_A,I,String_B)(F(2),A(M),X(M),F(2),A(I));
```

In this example, the first two characters are assigned to M. The value of M specifies the number of characters assigned to String_A and the number of characters being ignored before two characters are assigned to I, whose value is used to specify the number of characters assigned to String_B.

The value assigned to a variable during an input operation can be used in an expression in a format item that is associated with a later data item. An expression in a format item is evaluated and converted to an integer each time the format item is used.

The transmission is complete when the last data-list item has been processed. Subsequent format items, including control-format items, are ignored.

GET edit-directed

For more information about the GET statement, see "GET statement" on page 282.

Data in the stream is a continuous string of characters and graphics with no delimiters between successive values. The number of characters for each data value is specified by a format item in the format list. The characters are interpreted according to the associated format item. When the data list has been processed, execution of the GET statement stops and any remaining format items are not processed.

Each data-format item specifies the number of characters or graphics to be associated with the data-list item and how to interpret the data value. The data value is assigned to the associated data-list item, with any necessary conversion.

Fixed-point binary and floating-point binary data values must always be represented in the input stream with their values expressed in decimal digits. The F-, P-, and E-format items can then be used to access them, and the values are converted to binary representation upon assignment.

All blanks and quotation marks are treated as characters in the stream. Strings should not be enclosed in quotation marks. Quotation marks should not be doubled. The letter B should not be used to identify bit strings or G to identify graphic strings. If characters in the stream cannot be interpreted in the manner specified, the CONVERSION condition is raised.

Example:

```
get edit (Name, Data, Salary)(A(N), X(2), A(6), F(6,2));
```

This example specifies the following:

- The first `N` characters in the stream are treated as a character string and assigned to `Name`.
- The next two characters are skipped.
- The next six characters are assigned to `Data` in character format.
- The next six characters are considered an optionally signed decimal fixed-point constant and assigned to `Salary`.

PUT edit-directed

For more information about the PUT statement, see "PUT statement" on page 283.

The value of each data-list item is converted to the character or graphic representation specified by the associated data-format item and placed in the stream in a field whose width also is specified by the format item. When the data list has been processed, execution of the PUT statement stops and any remaining format items are not processed.

On output, binary items are converted to decimal values and the associated F- or E-format items must state the field width and point placement in terms of the converted decimal number. For the P-format these are specified by the picture specification.

On output, blanks are not inserted to separate data values in the output stream. String data is left-adjusted in the field to the width specified. Arithmetic data is right-adjusted. Because of the rules for conversion of arithmetic data to character type which can cause up to 3 leading blanks to be inserted (in addition to any blanks that replace leading zeros), generally there is at least 1 blank preceding an arithmetic item in the converted field. Leading blanks do not appear in the stream, however, unless the specified field width allows for them. Truncation, due to inadequate field-width specification, is on the left for arithmetic items, and on the right for string items. SIZE or STRINGSIZE is raised if truncation occurs.

Example 1

```
put edit('Inventory='‖Inum,Invcode)(A,F(5));
```

This example specifies that the character string `'Inventory='` is concatenated with the value of `Inum` and placed in the stream in a field whose width is the length of the resultant string. Then the value of `Invcode` is converted to character, as described by the F-format item, and placed in the stream right-adjusted in a field with a width of five characters (leading characters can be blanks).

Example 2

The following example shows the use of the COLUMN, LINE, PAGE, and SKIP format items in combination with one another:

```
put edit ('Quarterly Statement')
    (page, line(2), A(19))(Acct#, Bought, Sold, Payment, Balance)
    (skip(3), A(6), column(14), F(7,2), column(30), F(7,2),
    column(45), F(7,2), column(60), F(7,2));
```

This PUT statement specifies the following:

1. The heading `Quarterly Statement` is written on line two of a new page in the output file SYSPRINT.

2. Two lines are skipped. The next line in the output is the third line following the heading, or the fifth line of the report.

3. The following values are written:
    `Acct#`, beginning at character position 1

> > > > > `Bought`, beginning at character position 14
> > > > > `Sold`, beginning at character position 30
> > > > > `Payment`, beginning at character position 45
> > > > > `Balance` at character position 60.

Example 3
In the following example, the value of `Name` is inserted in the stream as a character string left-adjusted in a field of `N` characters.

```
put edit (Name,Number,City) (A(N),A(N-4),A(10));
```

`Number` is left-adjusted in a field of `N-4` characters; and `City` is left-adjusted in a field of 10 characters.

FORMAT statement

The FORMAT statement specifies a format list that can be used by edit-directed data transmission statements to control the format of the data being transmitted.
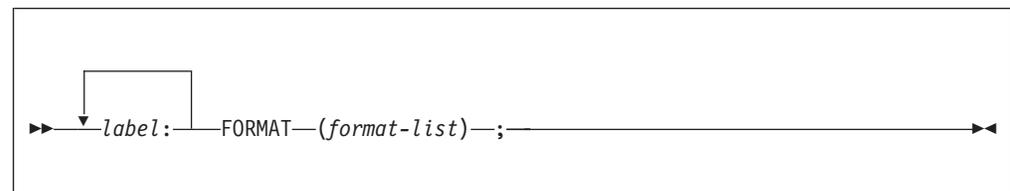
label   Same as the label-reference of the remote-format item, R, discussed in "R-format item" on page 312.

format list
>   Specified as described under "Edit-directed data specification" on page 293.

A GET or PUT EDIT statement can include an R-format item in its format-list option. That portion of the format list represented by the R-format item is supplied by the identified FORMAT statement.

A condition prefix associated with a FORMAT statement is not allowed.

List-directed data specification

For information on the syntax of the LIST data specification, refer to "Data specification options" on page 284.

Examples of list-directed data specifications are:

```
list (Card_Rate, Dynamic_Flow)

list ((Thickness(Distance)
    do Distance = 1 to 1000))

list (P, Z, M, R)

list (A*B/C, (X+Y)**2)
```

The specification in the last example can be used only for output, since it contains expressions. These expressions are evaluated when the statement is executed, and the result is placed in the stream.

Syntax of list-directed data

Data values in the stream, either input or output, are character or graphic representations.
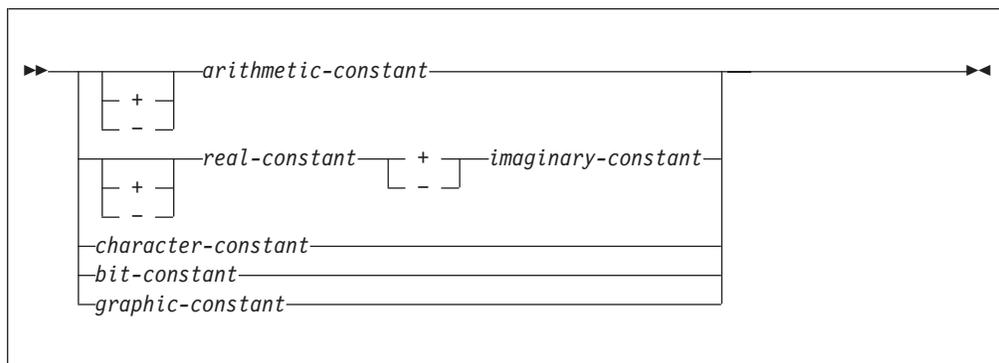
String repetition factors are not allowed. A blank must not follow a sign preceding a real constant, and must not precede or follow the central positive (+) or negative (–) symbol in complex expressions.

The length of the data value in the stream is a function of the attributes of the data value, including precision and length. Detailed discussions of the conversion rules and their effect upon precision are listed in the descriptions of conversion to character type in Chapter 5, "Data conversion," on page 73.

GET list-directed

For information about the GET statement, see "GET statement" on page 282.

On input, data values in the stream must be separated either by a blank or by a comma. This separator can be surrounded by one or more blanks. A null field in the stream is indicated either by the first nonblank character in the data stream being a comma, or by two commas separated by an arbitrary number of blanks. A null field specifies that the value of the associated data-list item remains unchanged.

Transmission of the list of constants or complex expressions on input is terminated by expiration of the list or at the end-of-file. For transmission of constants, the file is positioned in the stream ready for the next GET statement.

If the items are separated by a comma, the first character scanned when the next GET statement is executed is the one immediately following the comma:

```
Xbb,bbbXX
    _
```

If the items are separated by blanks only, the first item scanned is the next nonblank character:

```
XbbbbXXX
    _
```

unless the end-of-record is encountered, in which case the file is positioned at the end of the record:

```
Xbb–bbXXX
    _
```

However, if the end-of-record immediately follows a nonblank character (other than a comma), and the following record begins with blanks, the file is positioned at the first nonblank character in the following record:

```
X—bbbXXX
    —
```

If the record does terminate with a comma, the next record is not read until the next GET statement requires it.

If the data is a character constant, the surrounding quotation marks are removed, and the enclosed characters are interpreted as a character string. A double quotation mark is treated as a single quotation mark.

If the data is a bit constant, the enclosing quotation marks and the trailing character B are removed, and the enclosed characters are interpreted as a bit string.

If the data is a hexadecimal constant (X, BX, B4, GX), the enclosing quotation marks and the suffix are removed, and the enclosed characters are interpreted as a hexadecimal representation of a character, bit, or graphic string.

If the data is a mixed constant, the enclosing quotation marks and the suffix M are removed, and the enclosed constant is interpreted as a character string.

If the data is a graphic constant, the enclosing quotation marks and the trailing character G are removed, and the enclosed graphics are interpreted as a graphic string.

If the data is an arithmetic constant or complex expression, it is interpreted as coded arithmetic data with the base, scale, mode, and precision implied by the constant or by the rules for expression evaluation.

PUT list-directed

For more information about the PUT statement, see "PUT statement" on page 283.

The values of the data-list items are converted to character representations (except for graphics) and transmitted to the data stream. A blank separates successive data values transmitted. For PRINT files, items are separated according to program tab settings (see "PRINT attribute" on page 300).

Arithmetic values are converted to character.

Binary data values are converted to decimal notation before being placed in the stream.

For numeric character values, the character value is transmitted.

Bit strings are converted to character strings. The character string is enclosed in quotation marks and followed by the letter B.

Character strings are written out as follows:
• If the file does not have the attribute PRINT, enclosing quotation marks are supplied, and contained single quotation marks or apostrophes are replaced by two quotation marks. The field width is the current length of the string plus the number of added quotation marks.

- If the file has the attribute PRINT, enclosing quotation marks are not supplied, and contained single quotation marks or apostrophes are unmodified. The field width is the current length of the string.

Mixed strings are written out as follows:
- If the file does not have the attribute PRINT, SBCS quotation marks and the letter M are supplied. Contained SBCS quotes are replaced by two quotes.
- If the file has the attribute PRINT, the enclosing quotation marks and letter M are not supplied, and contained single quotation marks are unmodified.

Graphic strings are written out as follows:
- If the file does not have the attribute PRINT, SBCS quotation marks, and the letter G are supplied. Because the enclosing quotation marks are SBCS, contained graphic quotation marks are represented by a single graphic quotation mark (unmodified).
- If the file has the attribute PRINT, the enclosing quotation marks and letter G are not supplied, and graphic quotation marks are represented by a single graphic quotation mark (unmodified).

PRINT attribute

The PRINT attribute applies to files with the STREAM and OUTPUT attributes. It indicates that the file is intended to be printed; that is, the data associated with the file is to appear on printed pages, although it can first be written on some other medium.

```
►►──PRINT──────────────────────────────────────────────────►◄
```
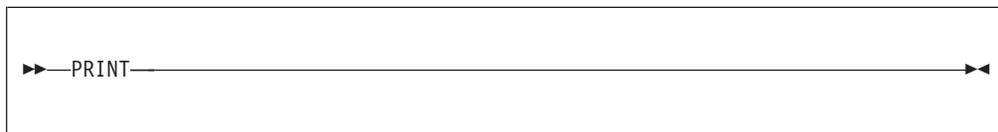
When PRINT is specified, the first data byte of each record of a PRINT file is reserved for an American National Standard (ANS) printer control character. The control characters are inserted by PL/I.

Data values transmitted by list- and data-directed data transmission are automatically aligned on the left margin and on implementation-defined preset tab positions.

The layout of a PRINT file can be controlled by the use of the options and format items listed in Table 36.

Table 36. Options and format items for PRINT files

Statement	Statement Option	Edit directed format item	Effect
OPEN	LINESIZE(n)	–	Established line width
OPEN	PAGESIZE(n)	–	Establishes page length
PUT	PAGE	PAGE	Skip to new page
PUT	LINE(n)	LINE(n)	Skip to specified line
PUT	SKIP[(n)]	SKIP[(n)]	Skip specified number of lines
PUT	–	COLUMN(n)	Skip to specified character position in line

Table 36. Options and format items for PRINT files (continued)

Statement	Statement Option	Edit directed format item	Effect
PUT	–	X(n)	Places blank characters in line to establish position.

LINESIZE and PAGESIZE establish the dimensions of the printed area of the page, excluding footings. The LINESIZE option specifies the maximum number of characters included in each printed line. If it is not specified for a PRINT file, a default value of 120 characters is used. There is no default for a non-PRINT file. The PAGESIZE option specifies the maximum number of lines in each printed page; if it is not specified, a default value of 60 lines is used. For example:

```
open file(Report) output stream print PAGESIZE(55) LINESIZE(110);
on endpage(Report) begin;
     put file(Report) skip list (Footing);
     Pageno = Pageno + 1;
     put file(Report) page list ('Page '||Pageno);
     put file(Report) skip (3);
end;
```

The OPEN statement opens the file Report as a PRINT file. The specification PAGESIZE(55) indicates that each page contains a maximum of 55 lines. An attempt to write on a page after 55 lines have already been written (or skipped) raises the ENDPAGE condition. The implicit action for the ENDPAGE condition is to skip to a new page, but you can establish your own action through use of the ON statement, as shown in the example.

LINESIZE(110) indicates that each line on the page can contain a maximum of 110 characters. An attempt to write a line greater than 110 characters places the excess characters on the next line.

When an attempt is made to write on line 56 (or to skip beyond line 55), the ENDPAGE condition is raised, and the begin-block shown here is executed. The ENDPAGE condition is raised only once per page. Consequently, printing can be continued beyond the specified PAGESIZE after the ENDPAGE condition has been raised. This can be useful, for example, if you want to write a footing at the bottom of each page.

The first PUT statement specifies that a line is skipped, and the value of Footing, presumably a character string, is printed on line 57 (when ENDPAGE is raised, the current line is always PAGESIZE+1). The page number, Pageno, is incremented, the file Report is set to the next page, and the character constant 'Page' is concatenated with the new page number and printed. The final PUT statement skips three lines, so that the next printing is on line 4. Control returns from the begin-block to the PUT statement that raised the ENDPAGE condition. However, any SKIP or LINE option specified in that statement has no further effect.

DBCS data in stream I/O

If DBCS data is used in list-directed or data-directed transmission, the GRAPHIC option of the ENVIRONMENT attribute must be specified for that file. It also must be specified if data-directed transmission uses DBCS names even though no DBCS data is present. DBCS continuation rules are applied and are the same rules as

DBCS data in stream I/O

those described in "DBCS continuation rules" on page 20. For information on how graphics are handled for edit-directed transmission, see "Edit-directed data specification" on page 293.

Chapter 14. Edit-directed format items

This chapter describes each of the edit-directed format items that can appear in the format list of a GET, PUT, or FORMAT statement. (See also "Edit-directed data specification" on page 293.) The format items are described in alphabetic order.

A-format item

The character (or A) format item describes the representation of a character value.
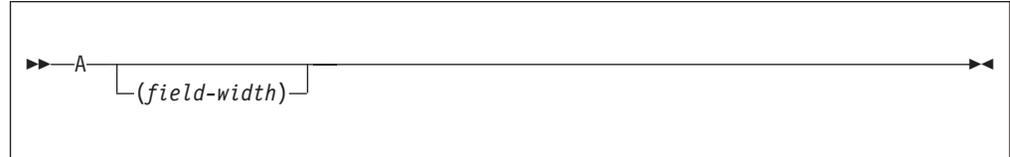
field-width
Specifies the number of character positions in the data stream that contain (or will contain) the string. It is an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

If an A-format item is specified without a length in a GET EDIT statement, the compiler issues a warning message and treats it as an L-format item (rather than issuing an error message and assigning it a length of 1).

On input, the specified number of characters is obtained from the data stream and assigned, with any necessary conversion, truncation, or padding, to the data-list item. The field width is always required on input and, if it is zero, a null string is obtained. If quotation marks appear in the stream, they are treated as characters in the string.

Consider the following example:

```
get file (Infile) edit (Item) (A(20));
```

The GET statement assigns the next 20 characters in `Infile` to `Item`. The value is converted from its character representation specified by the format item `A(20)`, to the representation specified by the attributes declared for `Item`.

On output, the data-list item is converted, if necessary, to a character string and is truncated or extended with blanks on the right to the specified field-width before being placed into the data stream. If the field-width is zero, no characters are placed into the data stream. Enclosing quotation marks are never inserted, nor are contained quotation marks doubled. If the field width is not specified, the default is equal to the character-string length of the data-list item (after conversion, if necessary, according to the rules given in Chapter 5, "Data conversion").

B-format item

The bit (or B) format item describes the character representation of a bit value. Each bit is represented by the character zero or one.
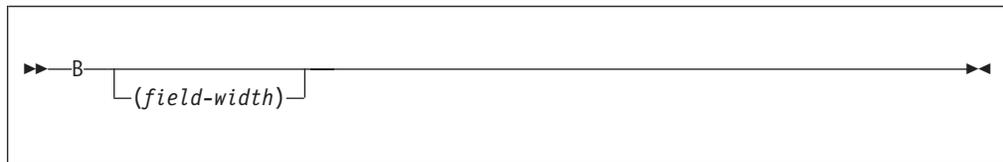
field-width
> Specifies the number of data-stream character positions that contain (or will contain) the bit string. It is an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

On input, the character representation of the bit string can occur anywhere within the specified field. Blanks, which can appear before and after the bit string in the field, are ignored. Any necessary conversion occurs when the bit string is assigned to the data-list item. The field width is always required on input, and if it is zero, a null string is obtained. Any character other than 0 or 1 in the string, including embedded blanks, quotation marks, or the letter B, raises the CONVERSION condition.

On output, the character representation of the bit string is left-adjusted in the specified field, and necessary truncation or extension with blanks occurs on the right. Any necessary conversion to bit-string is performed. No quotation marks are inserted, nor is the identifying letter B. If the field width is zero, no characters are placed into the data stream. If the field width is not specified, the default is equal to the bit-string length of the data-list item (after conversion, if necessary, according to the rules given in Chapter 5, "Data conversion").

In the example:
```
declare Mask bit(25);
put file(Maskfle) edit (Mask) (B);
```

The PUT statement writes the value of `Mask` in `Maskfle` as a string of 25 characters consisting of zeros and ones.

C-format item

The complex (or C) format item describes the character representation of a complex data value. You use one real-format-item to describe both the real and imaginary parts of the complex data value in the data stream.

```
►►─C─(real-format-item)──────────────────────────────────►◄
```

real-format-item
> Specified by one of the F-, E-, or P-format items. The P-format item must describe numeric character data.

On input, the letter I in the input raises the CONVERSION condition.

On output, the letter I is never appended to the imaginary part. If the second real format item (or the first, if only one appears) is an F or E item, the sign is transmitted only if the value of the imaginary part is less than zero. If the real format item is a P item, the sign is transmitted only if the S or - or + picture character is specified.

If you require an I to be appended, it must be specified as a separate data item in the data list, immediately following the variable that specifies the complex item. The I, then, must have a corresponding format item (either A or P). If a second real format item is specified, it is ignored.

COLUMN format item

The COLUMN format item positions the file to a specified character position within the current or following line.

```
►►─COLUMN─(character-position)──────────────────────────►◄
```

character-position
> Specifies an expression which is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

The file is positioned to the specified character position in the current line, provided it has not already passed this position. If the file is already positioned after the specified character position, the current line is completed and a new line is started; the format item is then applied to the following line.

Then, if the specified character position lies beyond the rightmost character position of the current line, or if the value of the expression for the character position is less than one, the default character position is one.

The rightmost character position is determined as follows:
- For output files, it is determined by the line size.
- For input files, it is determined using the length of the current logical record to determine the line size and, hence, the rightmost character position.

COLUMN must not be used in a GET STRING or PUT STRING statement.

COLUMN cannot be used with input or output lines that contain graphics or widechars.

On input, intervening character positions are ignored.

On output, intervening character positions are filled with blanks.

E-format item

The floating-point (or E) format item describes the character representation of a real floating-point decimal arithmetic data value.
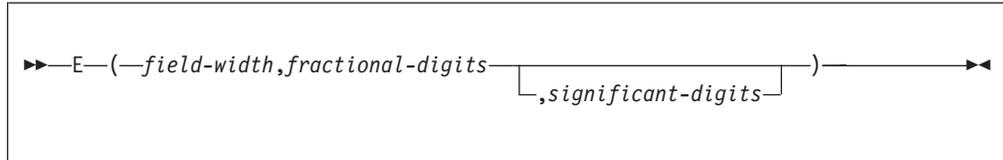
```
►►─E─(─field-width,fractional-digits────────────────────)───────►◄
                                └─,significant-digits─┘
```

field-width
> Specifies the total number of characters in the field. It is evaluated and converted to an integer value *w* each time the format item is used.

fractional-digits
> Specifies the number of digits in the mantissa that follow the decimal point. It is evaluated and converted to an integer value *d* each time the format item is used.

significant-digits
> Specifies the number of digits that must appear in the mantissa. It is evaluated and converted to an integer value *s* each time the format item is used.

The following must be true:

```
  w >= s = d+1 or w = 0
```

When w ¬= 0

```
  s > 0, d >= 0
```

The values for *w, d,* and *s* are field-width, fractional-digits, and significant-digits, respectively.

On input, either the data value in the data stream is an optionally signed real decimal floating-point or fixed-point constant located anywhere within the specified field or the CONVERSION condition is raised. (For convenience, the **E** preceding a signed exponent can be omitted.)

The field width includes leading and trailing blanks, the exponent position, the positions for the optional plus or minus signs, the position for the optional letter **E**, and the position for the optional decimal point in the mantissa.

The data value can appear anywhere within the specified field; blanks can appear before and after the data value in the field and are ignored. If the entire field is blank, the CONVERSION condition is raised. When no decimal point appears, *fractional-digits* specifies the number of character positions in the mantissa to the right of the assumed decimal point. If a decimal point does appear in the number, it overrides the specification of *fractional-digits.*

If *field-width* is 0, there is no assignment to the data-list item.

The statement:

```
  get file(A) edit (Cost) (E(10,6));
```

obtains the next 10 characters from `A` and interprets them as a floating-point decimal number. A decimal point is assumed before the rightmost 6 digits of the mantissa. The value of the number is converted to the attributes of `COST` and assigned to this variable.

On output, the data-list item is converted to floating-point and rounded if necessary. The rounding of data is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, 1 is added to the digit to the left of the truncated digit. This addition might cause adjustment of the exponent.

The character string written in the stream for output has one of the following syntaxes:

Note: Blanks are not allowed between the elements of the character strings.

- For d=0

```
          ┌─────────────┐
          │             │
►►──┬───┬──▼─s-digits────E──┬───┬──┬─exponent─────────────────►◄
    └─-─┘                   └─+─┘  │
                                   └─-─┘
```
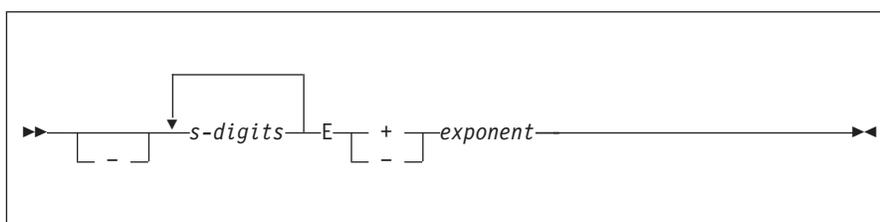
w must be >=s+6 for positive values, or >=s+7 for negative values. When the value is nonzero, the exponent is adjusted so that the leading digit of the mantissa is nonzero. When the value is zero, zero suppression is applied to all digit positions (except the rightmost) of the mantissa.

- For 0<d<s

```
          ┌─────────────┐      ┌───────────┐
          │             │      │           │
►►──┬───┬──▼─s-d-digits───.──▼─d-digits────E──┬───┬──┬──►
    └─-─┘                                      └─+─┘  │
                                               └─-─┘
►─exponent──────────────────────────────────────────►◄
```
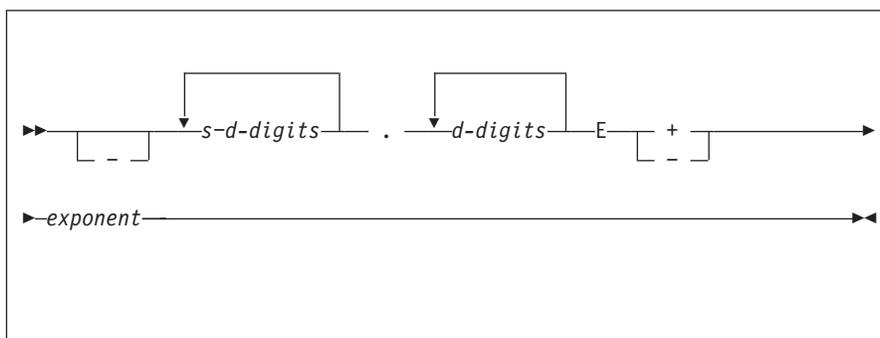
w must be >=s+7 for positive values, or >=s+8 for negative values. When the value is nonzero, the exponent is adjusted so that the leading digit of the mantissa is nonzero. When the value is zero, zero suppression is applied to all digit positions (except the first) to the left of the decimal point. All other digit positions contain zero.

- For d=s

```
              ┌───────────┐
              │           │
►►──┬───┬──0.──▼─d-digits────E──┬───┬──┬─exponent─────────────►◄
    └─-─┘                       └─+─┘  │
                                └─-─┘
```
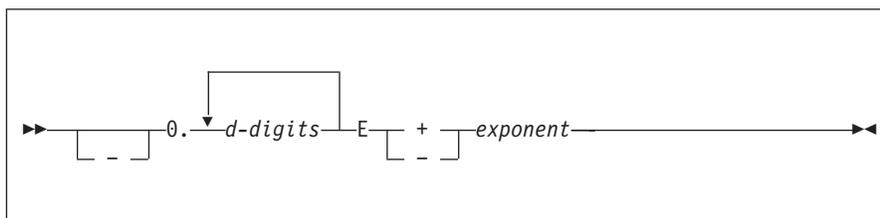
w must be >=d+8 for positive values, or >=d+9 for negative values.

When the value is nonzero, the exponent is adjusted so that the first fractional digit is nonzero. When the value is zero, each digit position contains zero.

The exponent is a 4-digit integer, which can be 4 zeros.

If the field width is such that significant digits or the sign are lost, the SIZE condition is raised. If the character string does not fill the specified field on output, the character string is right-adjusted and extended on the left with blanks.

F-format item

The fixed-point (or F) format item describes the character representation of a real fixed-point decimal arithmetic value.
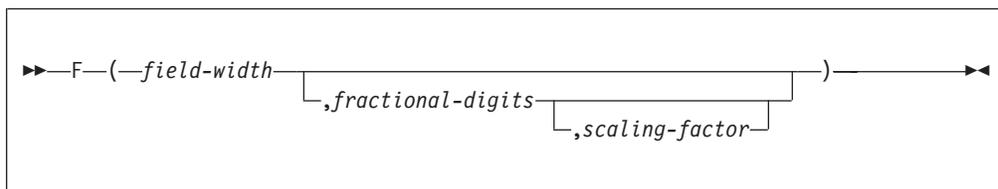
```
>>--F--(--field-width---------------------------------------)-------><
                      |                              |
                      +--,fractional-digits----------+
                                         |           |
                                         +--,scaling-factor--+
```

field-width
Specifies the total number of characters in the field. It is evaluated and converted to an integer value *w* each time the format item is used. The converted value must be nonnegative.

fractional-digits
Specifies the number of digits in the mantissa that follow the decimal point. It is evaluated and converted to an integer value *d* each time the format item is used. The converted value must be nonnegative. If *fractional-digits* is not specified, the default value is 0.

scaling-factor
Specifies the number of digits that must appear in the mantissa. It is evaluated and converted to an integer value *p* each time the format item is used.

On input, either the data value in the data stream is an optionally signed real decimal fixed-point constant located anywhere within the specified field or the CONVERSION condition is raised. Blanks can appear before and after the data value in the field and are ignored. If the entire field is blank, it is interpreted as zero.

If no *scaling-factor* is specified and no decimal point appears in the field, the expression for *fractional-digits* specifies the number of digits in the data value to the right of the assumed decimal point. If a decimal point does appear in the data value, it overrides the expression for *fractional-digits*.

If a *scaling-factor* is specified, it effectively multiplies the data value in the data stream by 10 raised to the integer value (*p*) of the *scaling-factor*. Thus, if *p* is positive, the data value is treated as though the decimal point appeared *p* places to the right of its given position. If *p* is negative, the data value is treated as though the decimal point appeared *p* places to the left of its given position. The given position of the decimal point is that indicated either by an actual point, if it appears, or by the expression for *fractional-digits*, in the absence of an actual point.

If the *field-width* is 0, there is no assignment to the data-list item.

On output, the data-list item is converted, if necessary, to fixed-point. Floating point data converts to FIXED DECIMAL (N,q) where *q* is the *fractional-digits* specified. The data value in the stream is the character representation of a real decimal fixed-point number, rounded if necessary, and right-adjusted in the specified field.

The conversion from decimal fixed-point type to character type is performed according to the normal rules for conversion. Extra characters can appear as blanks preceding the number in the converted string. And, since leading zeros are converted to blanks (except for a 0 immediately to the left of the point), additional blanks can precede the number. If a decimal point or a minus sign appears, either will cause one leading blank to be replaced.

If only the *field-width* is specified, only the integer portion of the number is written; no decimal point appears.

If both the *field-width* and *fractional-digits* are specified, both the integer and fractional portions of the number are written. If the value (*d*) of *fractional-digits* is greater than 0, a decimal point is inserted before the rightmost *d* digits. Trailing zeros are supplied when *fractional-digits* is less than *d* (the value *d* must be less than *field-width*). If the absolute value of the item is less than 1, a 0 precedes the decimal point. Suppression of leading zeros is applied to all digit positions (except the first) to the left of the decimal point.

The rounding of the data value is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, 1 is added to the digit to the left of the truncated digit.

On output, if the data-list item is less than 0, a minus sign is prefixed to the character representation; if it is greater than or equal to 0, no sign appears. Therefore, for negative values, the *field-width* might need to include provision for the sign, a decimal point, and a 0 before the point.

If the *field-width* is such that any character is lost, the SIZE condition is raised.

In the example:

```
declare Total fixed(4,2);
put edit (Total) (F(6,2));
```

The PUT statement specifies that the value of Total is converted to the character representation of a fixed-point number and written into the output file SYSPRINT. A decimal point is inserted before the last two numeric characters, and the number is right-adjusted in a field of six characters. Leading zeros are changed to blanks (except for a zero immediately to the left of the point), and, if necessary, a minus sign is placed to the left of the first numeric character.

G-format item

For the compiler, the graphic (or G) format item describes the representation of a graphic string.

```
►►─┬─G─────────────────┬─────────────────────────────────►◄
   └─(field-width)─┘
```
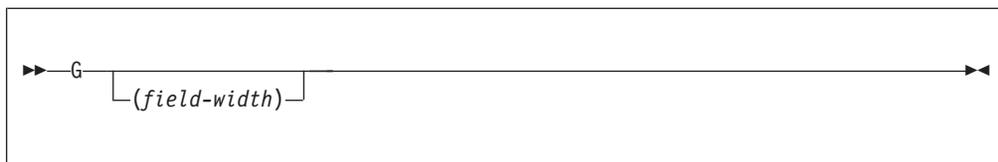
field-width

Specifies the number of 2-byte positions in the data stream that contain (or will contain) the graphic string. It is an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used. End-of-line must not occur between the 2 bytes of a graphic.

On input, the specified number of graphics is obtained from the data stream and assigned, with any necessary truncation or padding, to the data-list item. The *field-width* is always required on input, and if it is zero, a null string is obtained.

On output, the data-list item is truncated or extended (with the padding graphic) on the right to the specified *field-width* before being placed into the data stream. No enclosing quotation marks are inserted, nor is the identifying suffix, *G*, inserted. If the *field-width* is zero, no graphics are placed into the data stream. If the *field-width* is not specified, it defaults to be equal to the graphic-string length of the data-list item.

In the following example, if file OUT has the GRAPHIC option, six bytes are transmitted.

```
declare A graphic(3);
put file(Out) edit (A) (G(3));
```

L-format item

On input, L indicates that all data up to the end of the line is assigned to the data item.

```
►►─L─────────────────────────────────────────────────────►◄
```
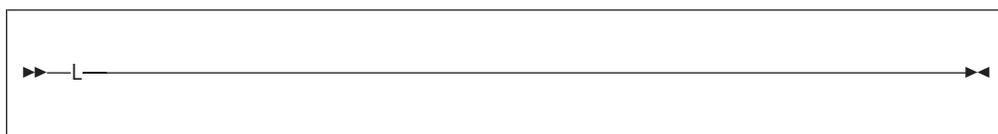
On output, L indicates that the data item, padded on the right with blanks, if necessary, is to fill the remainder of the output line.

LINE format item

The LINE format item specifies the line on the current page of a PRINT file upon which the next data-list item will be printed, or it raises the ENDPAGE condition.

```
►►──LINE──(line-number)──────────────────────────────►◄
```

line-number
Can be represented by an expression, which is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

Blank lines are inserted, if necessary.

If the specified *line-number* is less than or equal to the current line number, or if the specified line is beyond the limits set by the PAGESIZE option of the OPEN statement (or by default), the ENDPAGE condition is raised. An exception is that if the specified *line-number* is equal to the current line number, and the column 1 character has not yet been transmitted, the effect is as for a SKIP(0) item, that is, a carriage return with no line spacing.

If *line-number* is zero, it defaults to one (1).

P-format item

The picture (or P) format item describes the character representation of real numeric character values and of character values.

The picture specification of the P-format item, on input, describes the form of the data item expected in the data stream and, in the case of a numeric character specification, how the item's arithmetic value is interpreted. If the indicated character does not appear in the stream, the CONVERSION condition is raised.

On output, the value of the associated element in the data list is converted to the form specified by the picture specification before it is written into the data stream.

```
►►──P──'picture-specification'──────────────────────►◄
```

picture-specification
Is discussed in detail in Chapter 15, "Picture specification characters," on page 315.

For example:

```
get edit (Name, Total) (P'AAAAA',P'9999');
```

When this statement is executed, the input file SYSIN is the default. The next five characters input from SYSIN must be alphabetic or blank and they are assigned to Name. The next four characters must be digits and they are assigned to Total.

PAGE format item

The PAGE format item specifies that a new page is established. It can be used only with PRINT files.

```
►►──PAGE─────────────────────────────────────────────►◄
```

Starting a new page positions the file to the first line of the next page.

R-format item

The remote (or R) format item specifies that the format list in a FORMAT statement is to be used (as described under "FORMAT statement" on page 297).

```
►►──R──(label-reference)──────────────────────────────►◄
```
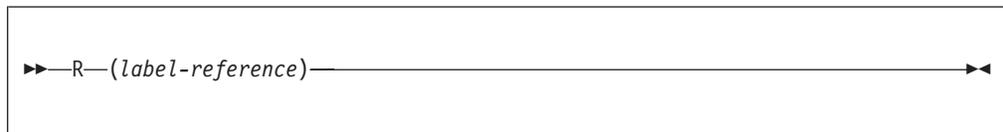
label-reference
   Has as its value the label constant of a FORMAT statement.

The R-format item and the specified FORMAT statement must be internal to the same block, and they must be in the same invocation of that block.

A remote FORMAT statement cannot contain an R-format item that references itself as a label reference, nor can it reference another remote FORMAT statement that leads to the referencing of the original FORMAT statement.

Conditions enabled for the GET or PUT statement must also be enabled for the remote FORMAT statement(s) that are referred to.

If the GET or PUT statement is the single statement of an ON-unit, that statement is a block, and it cannot contain a remote format item.

Example

```
 declare Switch label;
 get file(In) list(Code);
 if Code = 1 then
   Switch = L1;
 else
   Switch = L2;
 get file(In) edit (W,X,Y,Z)
     (R(Switch));
L1:  format (4 F(8,3));
L2:  format (4 E(12,6));
```

`Switch` has been declared a label variable. The second GET statement can be made to operate with either of the two FORMAT statements.

SKIP format item

The SKIP format item specifies that a new line is to be defined as the current line.

```
►►──SKIP──────────────────────────────────────────────────────────►◄
          └─(relative-line)─┘
```
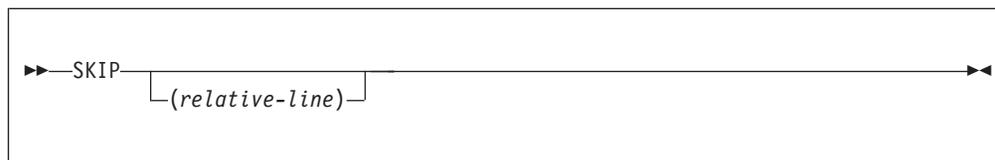
relative-line

Specifies an expression, which is evaluated and converted to an integer value, *n*, each time the format item is used. The converted value must be nonnegative and less than 32,768. It must be greater than zero for non-PRINT files. If it is zero, or if it is omitted, the default is 1.

The new line is the *n*th line after the present line.

If *n* is greater than one, one or more lines are ignored on input; on output, one or more blank lines are inserted.

The value *n* can be zero for PRINT files only, in which case the positioning is at the start of the current line. Characters previously written can be overprinted.

For PRINT files, if the specified *relative-line* is beyond the limit set by the PAGESIZE option of the OPEN statement (or the default), the ENDPAGE condition is raised.

If the SKIP format item is the first item to be executed after a file has been opened, output commences on the *n*th line of the first page. If *n* is zero or 1, it commences on the first line of the first page.

For example:

```
get file(In) edit(Man,Overtime)
    (skip(1), A(6), COL(60), F(4,2));
```

This statement positions the data set associated with file In to a new line. The first 6 characters on the line are assigned to `Man`, and the 4 characters beginning at character position 60 are assigned to `Overtime`.

X-format item

The spacing (or X) format item specifies the relative spacing of data values in the data stream.

```
►►──X──(field-width)──────────────────────────────────────────────►◄
```
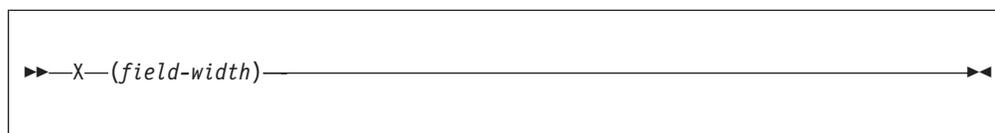
field-width

Specifies an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used. The integer value specifies the number of characters before the next field of the data stream, relative to the current position in the stream.

X-format

On input, the specified number of characters are spaced over in the data stream and not transmitted to the program.

For example:

```
get edit (Number, Rebate)
        (A(5), X(5), A(5));
```

The next 15 characters from the input file, SYSIN, are treated as follows: the first five characters are assigned to Number, the next five characters are ignored, and the remaining five characters are assigned to Rebate.

On output, the specified number of blank characters are inserted into the stream.

In the example:

```
put file(Out) edit (Part, Count) (A(4), X(2), F(5));
```

Four characters that represent the value of Part, then two blank characters, and finally five characters that represent the fixed-point value of Count, are placed in the file named Out.

Chapter 15. Picture specification characters

A picture specification consists of a sequence of picture characters enclosed in single or double quotation marks. The characters describe the contents of each position of the character or numeric character data item, and the contents of the output. The specification can be made in two ways:

- As part of the PICTURE attribute in a declaration
- As part of the P-format item (described in "P-format item" on page 311) for edit-directed input and output.

A picture specification describes either a character data item or a numeric character data item. The presence of an A or X picture character defines a picture specification as a character picture specification; otherwise, it is a numeric character picture specification.

A *character pictured item* can consist of alphabetic characters, decimal digits, blanks, currency and punctuation characters.

A *numeric character pictured item* can consist only of decimal digits, an optional decimal point, an optional letter E, and, optionally, one or two plus or minus signs. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are part of the character value of the variable.

Figures in this section illustrate how different picture specifications affect the representation of values when assigned to a pictured variable or when printed using the P-format item. Each figure shows the original value of the data, the attributes of the variable from which it is assigned (or written), the picture specification, and the character value of the numeric character or pictured character variable.

The concepts of the two types of picture specifications are described separately in the sections that follow.

Picture repetition factor

A picture repetition factor specifies the number of repetitions of the next picture character in the specification.

```
►►──(n)────────────────────────────────────────────────►◄
```
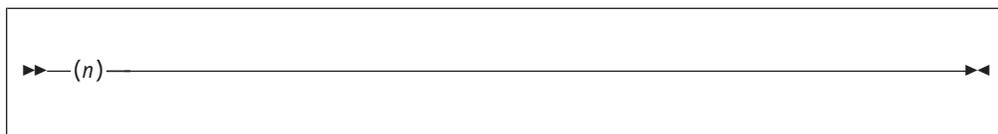
n   An integer. No blanks are allowed within the parentheses. If n is 0, the picture character is ignored.

For example, the following picture specifications result in the same description:

```
'999V99'
'(3)9V(2)9'
```

Picture characters for character data

A character picture specification describes a nonvarying character data item. You can specify that any position in the data item can contain only characters from certain subsets of the complete set of available characters. The data can consist of alphabetic characters, decimal digits, and blanks.

The only valid characters in a character picture specification are X, A, and 9. Each of these specifies the presence of one character position in the character value, which can contain the following:

X   Any character of the 256 possible bit combinations represented by the 8-bit byte.

A   Any alphabetic or extralingual (#, @, $) character, or blank.

9   Any digit, or blank. (Note that the 9 picture specification character allows blanks only for character data.)

When a character value is assigned, or transferred, to a picture character data item, the particular character in each position is validated according to the corresponding picture specification character. If the character data does not match the specification for that position, the CONVERSION condition is raised for the invalid character. (However, if you change the value by record-oriented transmission or by using an alias, there is no checking.) For example:

```
declare Part# picture 'AAA99X';
put edit (Part#) (P'AAA99X');
```

The following values are valid for `Part#`:

```
'ABC12M'
'bbb09/'
'XYZb13'
```

The following values are not valid for `Part#` (the invalid characters are underscored):

```
'AB123M'
'ABC1/2'
'Mb#A5;'
```

Table 37 shows examples of character picture specifications.

Table 37. Character picture specification examples

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
CHARACTER(5)	'9B/2L'	XXXXX	9B/2L
CHARACTER(5)	'9B/2L'	XXX	9B/
CHARACTER(5)	'9B/2L'	XXXXXXX	9B/2Lbb
CHARACTER(5)	'ABCDE'	AAAAA	ABCDE
CHARACTER(5)	'ABCDE'	AAAAAA	ABCDEb
CHARACTER(5)	'ABCDE'	AAA	ABC
CHARACTER(5)	'12/34'	99X99	12/34
CHARACTER(5)	'L26.7'	A99X9	L26.7

Picture characters for numeric character data

Numeric character data represents numeric values. The picture specification cannot contain the character data picture characters X or A. The picture characters for numeric character data can also specify editing of the data.

A numeric character variable can have two values, depending upon how the variable is used. The types of values are as follows:

Arithmetic
> The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point, possibly a sign, and an optionally-signed exponent or scaling factor. The arithmetic value of a numeric character variable is used in the following situations:
>
> - Whenever the variable appears in an expression that results in a coded arithmetic value or bit value (this includes expressions with the ¬, &, |, and comparison operators; even comparison with a character string uses the arithmetic value of a numeric character variable)
> - Whenever the variable is assigned to a coded arithmetic, numeric character, or bit variable
> - When used with the C, E, F, B, and P (numeric) format items in edit-directed I/O.

The arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

Character value
> The character value is the value expressed by the decimal digits of the data item, as well as all of the editing and insertion characters appearing in the picture specification. The character value does not, however, include the assumed location of a decimal point, as specified by the picture characters V, K, or F. The character value of a numeric character variable is used:
>
> - Whenever the variable appears in a character expression
> - In an assignment to a character variable
> - Whenever the data is printed using list-directed or data-directed output
> - Whenever a reference is made to a character variable that is defined or based on the numeric character variable
> - Whenever the variable is printed using edit-directed output with the A or P (character) format items.

Picture characters for numeric character data

No data conversion is necessary.

Numeric character data can contain only decimal digits, an optional decimal point, an optional letter E, and one or two plus or minus signs. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are part of the character value of the variable.

A numeric character specification consists of one or more fields, each field describing a fixed-point number. A floating-point specification has two fields—one for the mantissa and one for the exponent. The first field can be divided into subfields by inserting a V picture specification character. The data preceding the V (if any) and that following it (if any) are subfields of the specification.

A requirement of the picture specification for numeric character data is that each field must contain at least one picture character that specifies a digit position. This picture character, however, need not be the digit character 9. Other picture characters, such as the zero suppression characters (Z or *), also specify digit positions.

Note: All characters except K, V, and F specify the occurrence of a character in the character representation.

The picture characters for numeric character specifications are discussed in the following sections:
- "Digits and decimal points" describes data specified with the picture characters 9 and V.
- "Zero suppression" on page 320 describes picture data specified with the picture characters Z and asterisk (*).
- "Insertion characters" on page 321 discusses the use of the insertion characters (point, comma, slash, and B).
- "Insertion and decimal point characters" on page 321 describes the use of the decimal point and insertion characters with the V picture character.
- "Defining currency symbols" on page 322 describes how to define your own character(s) as a currency symbol, and "Signs and currency symbols" on page 324 describes the use of signs and currency symbols.
- "Credit, debit, overpunched, and zero replacement characters" on page 326 discusses the picture characters CR, DB, T, I, R, and Y used for credit, debit, overpunched, and zero replacement functions.
- "Exponent characters" on page 328 discusses the picture characters K and E used for exponents.
- "Scaling factor" on page 328 describes the picture character F used for scaling factors.
- "Picture repetition factor" on page 316 describes the picture repetition character.

Digits and decimal points

The picture characters 9 and V are used in numeric character specifications that represent fixed-point decimal values.

9   Specifies that the associated position in the data item contains a decimal digit. (Note that the 9 picture specification character for numeric character data is different from the specification for character data because the corresponding character cannot be a blank for character data.)

A string of n 9 picture characters specifies that the item is a nonvarying character-string of length n, each of which is a digit (0 through 9). For example:

```
dcl digit picture'9',
    Count picture'999',
    XYZ picture '(10)9';
```

An example of use is:

```
dcl 1 Record,
      2 Data char(72),
      2 Identification char(3),
      2 Sequence pic'99999';
  dcl Count fixed dec(5);
  ⋮
  Count=Count+1;
  Sequence=Count;
  write file(Output) from(Record);
```

V Specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point or decimal comma is inserted. The integer value and fractional value of the assigned value, after modification by the optional scaling factor F(±x), are aligned on the V character. Therefore, an assigned value can be truncated or extended with zero digits at either end. (If significant digits are truncated on the left, the result is undefined and the SIZE condition is raised if enabled.)

If no V character appears in the picture specification of a fixed-point decimal value (or in the first field of a picture specification of a floating-point decimal value), a V is assumed at the right end of the field specification. This can cause the assigned value to be truncated, if necessary, to an integer.

The V character cannot appear more than once in a picture specification.

For example:

```
dcl Value picture 'Z9V999';
Value = 12.345;
dcl Cvalue char(5);
Cvalue = Value;
```

`Cvalue`, after assignment of `Value`, contains '12345'.

Table 38 shows examples of digit and decimal point characters.

Table 38. Examples of digit and decimal point characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5)	12345	99999	12345
FIXED(5)	12345	99999V	12345
FIXED(5)	12345	999V99	undefined
FIXED(5)	12345	V99999	undefined
FIXED(7)	1234567	99999	undefined
FIXED(3)	123	99999	00123
FIXED(5,2)	123.45	999V99	12345
FIXED(7,2)	12345.67	9V9	undefined
FIXED(5,2)	123.45	99999	00123

Note: When the character value is undefined, the SIZE condition is raised.

Zero suppression

The picture characters Z and asterisk (*) specify conditional digit positions in the character value and can cause leading zeros to be replaced by asterisks or blanks. Leading zeros are those that occur in the leftmost digit positions of fixed-point numbers or in the leftmost digit positions of the two parts of floating-point numbers, that are to the left of the assumed position of a decimal point, and that are not preceded by any of the digits 1 through 9. The leftmost nonzero digit in a number and all digits, zeros or not, to the right of it represent significant digits.

Z Specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank. Otherwise, the digit in the position is unchanged. The picture character Z cannot appear in the same field as the picture character * or a drifting character, nor can it appear to the right of any of the picture characters in a field.

* Specifies a conditional digit position. It is used the way the picture character Z is used, except that leading zeros are replaced by asterisks. The picture character asterisk cannot appear in the same field as the picture character Z or a drifting character, nor can it appear to the right of any of the picture characters in a field.

Table 39 shows examples of zero suppression characters.

Table 39. Examples of zero suppression characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5)	12345	ZZZ99	12345
FIXED(5)	00100	ZZZ99	bb100
FIXED(5)	00100	ZZZZZ	bb100
FIXED(5)	00000	ZZZZZ	bbbbb
FIXED(5,2)	123.45	ZZZ99	bb123
FIXED(5,2)	001.23	ZZZV99	bb123
FIXED(5)	12345	ZZZV99	undefined
FIXED(5,2)	000.08	ZZZVZZ	bbb08
FIXED(5,2)	000.00	ZZZVZZ	bbbbb
FIXED(5)	00100	*****	**100
FIXED(5)	00000	*****	*****
FIXED(5,2)	000.01	***V**	***01
FIXED(5,2)	95	$**9.99	$**0.95
FIXED(5,2)	12350	$**9.99	$123.50

Note: When the character value is undefined, the SIZE condition is raised.

If one of the picture characters Z or asterisk appears to the right of the picture character V, all fractional digit positions in the specification, as well as all integer digit positions, must use the Z or asterisk picture character, respectively. When all digit positions to the right of the picture character V contain zero suppression picture characters, fractional zeros of the value are suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The character value of the data item will then consist of blanks or asterisks. No digits in the fractional part are replaced by blanks or asterisks if the fractional part contains any significant digit.

Insertion characters

The picture characters comma (,), point (.), slash (/), and blank (B) cause the specified character to be inserted into the associated position of the numeric character data. They do not indicate digit or character positions, but are inserted between digits or characters. Each does, however, actually represent a character position in the character value, whether or not the character is suppressed. The comma, point, and slash are conditional insertion characters and can be suppressed within a sequence of zero suppression characters. The blank is an unconditional insertion character, and always specifies that a blank appears in the associated position.

Insertion characters are applicable only to the character value. They specify nothing about the arithmetic value of the data item. They never cause decimal point or decimal comma alignment in the picture specifications of a fixed-point decimal number and are not a part of the arithmetic value of the data item. Decimal alignment is controlled by the picture characters V and F.

Comma (,), point (.), or slash (/)
> Inserts a character into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur, the character is inserted only under the following conditions:
>
> * When an unsuppressed digit appears to the left of the character's position
> * When a V appears immediately to the left of the character and the fractional part of the data item contains any significant digits
> * When the character is at the start of the picture specification
> * When the character is preceded only by characters that do not specify digit positions.
>
> In all other cases where zero suppression occurs, a comma, point, or slash insertion character is treated as a zero suppression character identical to the preceding character.

B Specifies that a blank character be inserted into the associated position of the character value of the numeric character data.

Insertion and decimal point characters

The point, comma, or slash can be used in conjunction with the V to cause insertion of the point (or comma or slash) in the position that delimits the end of the integer portion in and the beginning of the fractional portion of a fixed-point (or floating-point) number, as might be desired in printing, since the V does not cause printing of a point. The point must immediately precede or immediately follow the V. If the point precedes the V, it is inserted only if an unsuppressed digit appears to the left of the V, even if all fractional digits are significant. If the point immediately follows the V, it is suppressed if all digits to the right of the V are suppressed, but it appears if there are any unsuppressed fractional digits (along with any intervening zeros).

The following example shows decimal conventions that are used in different countries.

```
  declare A picture 'Z,ZZZ,ZZZV.99',
          B picture 'Z.ZZZ.ZZZV,99',
          C picture 'ZBZZZBZZZV,99';
 A,B,C = 1234;
 A,B,C = 1234.00;
```

Insertion characters and decimal points

A, B, and C represent nine-digit numbers with a decimal point or decimal comma assumed between the seventh and eighth digits. The actual point specified by the decimal point insertion character is not a part of the arithmetic value. It is, however, part of its character value. The two assignment statements assign the same character value to A, B, and C as follows:

```
1,234.00    /* value of A */
1.234,00    /* value of B */
1 234,00    /* value of C */
```

In the following example, decimal point alignment during assignment occurs on the character V. If Rate is printed, it appears as '762.00', but its arithmetic value is 7.6200.

```
declare Rate picture '9V99.99';
Rate = 7.62;
```

Table 40 shows examples of insertion characters.

Table 40. Examples of insertion characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(4)	1234	9,999	1,234
FIXED(6,2)	1234.56	9,999V.99	1,234.56
FIXED(4,2)	12.34	ZZ.VZZ	12.34
FIXED(4,2)	00.03	ZZ.VZZ	bbb03
FIXED(4,2)	00.03	ZZV.ZZ	bb.03
FIXED(4,2)	12.34	ZZV.ZZ	12.34
FIXED(4,2)	00.00	ZZV.ZZ	bbbbb
FIXED(9,2)	1234567.89	9,999,999.V99	1,234,567.89
FIXED(7,2)	12345.67	**,999V.99	12,345.67
FIXED(7,2)	00123.45	**,999V.99	***123.45
FIXED(9,2)	1234567.89	9.999.999V,99	1.234.567,89
FIXED(6)	123456	99/99/99	12/34/56
FIXED(6)	101288	99-99-99	10-12-88
FIXED(6)	123456	99.9/99.9	12.3/45.6
FIXED(6)	001234	ZZ/ZZ/ZZ	bbb12/34
FIXED(6)	000012	ZZ/ZZ/ZZ	bbbbb12
FIXED(6)	000000	ZZ/ZZ/ZZ	bbbbbbbb
FIXED(6)	000000	**/**/**	********
FIXED(6)	000000	**B**B**	**b**b**
FIXED(6)	123456	99B99B99	12b34b56
FIXED(3)	123	9BB9BB9	1bb2bb3
FIXED(2)	12	9BB/9BB	1bb/2bb

Defining currency symbols

A currency symbol can be used as a picture character denoting a character value of numeric character data. This symbol can be the dollar sign ($) or any symbol you choose. The symbol can be any sequence of characters enclosed in < and > characters.
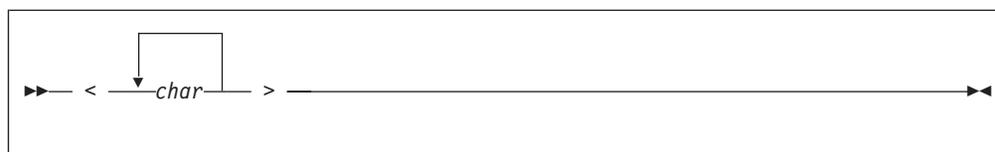
< Indicates the start of the currency symbol. It acts as an escape character. If you want to use the character <, you must specify <<.

char Is any character that will be part of your currency symbol(s).

> indicates the end of the currency symbol. If you want to use the character >, you must specify <>.

More than one > indicates a drifting string (discussed in 325).

Examples of general insertion strings include the following:

<DM> represents the Deutschemark

<Fr> represents the French Franc

<K$> represents the Khalistan Dollar

<Sur.f> represents the Surinam Guilder

<$> represents the dollar sign

If the character < or > must be included in the sequence, it must be preceded by another <. Therefore, < acts as an escape character also.

The entire sequence enclosed in < > represents one ″symbol″ and therefore represents the character value for one numeric character. If the symbol needs to be represented as a drifting picture character, you specify > following the ″< >″ to represent each occurrence.

For example:

Pic '<DM>>>.>>9,V99'
represents a 10 character numeric picture, yielding 11 characters after assignment.

Pic '<Sur.f>999,V99'
represents a 7 character numeric picture, yielding 11 characters after assignment.

Pic '<K$>>>,>>9.V99'
represents a 10 character numeric picture, yielding 11 characters after assignment.

Pic '<$>>>,>>9.V99'
represents a 10 character numeric picture, yielding 10 characters after assignment.

Pic '$$$,$$9.V99'
has the same value as the previous picture specification.

More examples of currency symbol definition include the following:
```
dcl P pic'<DM>9.999,V99';
P = 1234.40;                            /* Yields 'DM1.234,40'  */


dcl P pic'<DM>9.999,V99';
```

```
         P =   34.40;                          /* Yields 'DM   34,40'  */


         dcl P pic'<DM>>.>>9,V99';
         P = 1234.40;                          /* Yields 'DM1.234,40'  */


         dcl P pic'<DM>>.>>9,V99';
         P =   34.40;                          /* Yields '   DM34,40'  */


         dcl P pic'9.999,V99<K$>';
         P = 1234.40;                          /* Yields '1.234,40K$' */
```

In this chapter, the term *currency symbol* and the $ symbol refer to the dollar sign or any user-defined currency symbol.

Signs and currency symbols

The picture characters S, +, and – specify signs in numeric character data. The picture character $ (or the currency symbol) specifies a currency symbol in the character value of numeric character data. Only one type of sign character can appear in each field.

currency symbol
Specifies the currency symbol.

In the following example:
```
dcl Price picture '$99V.99';
Price = 12.45;
```

The character value of `Price` is `'$12.45'`. Its arithmetic value is `12.45`.

For information on specifying a character as a currency symbol, refer to "Defining currency symbols" on page 322.

S   Specifies the plus sign character (+) if the data value is >=0; otherwise, it specifies the minus sign character (–). The rules are identical to those for the currency symbol.

Consider the following example:
```
dcl Root picture 'S999';
```

The value `50` is held as `'+050'`, the value `0` as `'+000'` and the value `-243` as `'-243'`.

+   Specifies the plus sign character (+) if the data value is >=0; otherwise, it specifies a blank. The rules are identical to those for the currency symbol.

–   Specifies the minus sign character (–) if the data value is <0; otherwise, it specifies a blank. The rules are identical to those for the currency symbol.

Signs and currency symbols can be used in either a static or a drifting manner.

Static use
Static use specifies that a sign, a currency symbol, or a blank appears in the associated position. An S, +, or – used as a static character can appear to the right or left of all digits in the mantissa and exponent fields of a floating-point specification, and to the right or left of all digit positions of a fixed-point specification.

Drifting use

Drifting use specifies that leading zeros are to be suppressed. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a currency symbol (except that where all digit positions are occupied by drifting characters and the value of the data item is zero, the drifting character is not inserted).

A drifting character is specified by multiple use of that character in a picture field. The drifting character must be specified in each digit position through which it can drift. Drifting characters must appear in a sequence of the same drifting character, optionally containing a V and one of the insertion characters comma, point, slash, or B. Any of the insertion characters slash, comma, or point within or immediately following the string is part of the drifting string. The character B always causes insertion of a blank, wherever it appears. A V terminates the drifting string, except when the arithmetic value of the data item is zero; in that case, the V is ignored. A field of a picture specification can contain only one drifting string. A drifting string cannot be preceded by a digit position nor can it occur in the same field as the picture characters * and Z.

The position in the data associated with the characters slash, comma, and point appearing in a string of drifting characters contains one of the following:
- Slash, comma, or point if a significant digit appears to the left
- The drifting symbol, if the next position to the right contains the leftmost significant digit of the field
- Blank, if the leftmost significant digit of the field is more than one position to the right.

If a drifting string contains the drifting character *n* times, the string is associated with *n*-1 conditional digit positions. The position associated with the leftmost drifting character can contain only the drifting character or blank, never a digit. Two different picture characters cannot be used in a drifting manner in the same field.

If a drifting string contains a V within it, the V delimits the preceding portion as a subfield, and all digit positions of the subfield following the V must also be part of the drifting string that commences the second subfield.

In the case in which all digit positions after the V contain drifting characters, suppression in the subfield occurs only if all of the integer and fractional digits are zero. The resulting edited data item is then all blanks (except for any insertion characters at the start of the field). If there are any nonzero fractional digits, the entire fractional portion appears unsuppressed.

If, during or before assignment to a picture, the fractional digits of a decimal number are truncated so that the resulting value is zero, the sign inserted in the picture corresponds to the value of the decimal number prior to its truncation. Thus, the sign in the picture depends on how the decimal value was calculated.

Table 41 on page 325 shows examples of signs and currency symbol characters.

Table 41. Examples of signs and currency characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5,2)	123.45	$999V.99	$123.45
FIXED(5,2)	012.00	99$	12$
FIXED(5,2)	001.23	$ZZZV.99	$bb1.23

Table 41. Examples of signs and currency characters (continued)

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5,2)	000.00	$ZZZV.ZZ	bbbbbbb
FIXED(1)	0	$$$.$$	bbbbbb
FIXED(5,2)	123.45	$$$9V.99	$123.45
FIXED(5,2)	001.23	$$$9V.99	bb$1.23
FIXED(2)	12	$$$,999	bbb$012
FIXED(4)	1234	$$$,999	b$1,234
FIXED(5,2)	2.45	SZZZV.99	+bb2.45
FIXED(5)	214	SS,SS9	bb+214
FIXED(5)	−4	SS,SS9	bbbb−4
FIXED(5,2)	−123.45	+999V.99	b123.45
FIXED(5,2)	−123.45	−999V.99	−123.45
FIXED(5,2)	123.45	999V.99S	123.45+
FIXED(5,2)	001.23	++B+9V.99	bbb+1.23
FIXED(5,2)	001.23	− − −9V.99	bbb1.23
FIXED(5,2)	−001.23	SSS9V.99	bb−1.23

Credit, debit, overpunched, and zero replacement characters

The picture characters CR, DB, T, I, and R cannot be used with any other sign characters in the same field.

Credit and debit

The character pairs CR (credit) and DB (debit) specify the signs of real numeric character data items.

CR Specifies that the associated positions contain the letters CR if the value of the data is <0. Otherwise, the positions will contain two blanks. The characters CR can appear only to the right of all digit positions of a field.

DB Specifies that the associated positions contain the letters DB if the value of the data is <0. Otherwise, the positions will contain two blanks. The characters DB can appear only to the right of all digit positions of a field.

Overpunch

Any of the picture characters T, I, or R (known as overpunch characters) specifies that a character represents the corresponding digit and the sign of the data item. A floating-point specification can contain two—one in the mantissa field and one in the exponent field. The overpunch character can be specified for any digit position within a field.

The T, I, and R picture characters specify how the input characters are interpreted, as shown in Table 42.

Table 42. Interpretation of the T, I, and R picture characters

T or I	T or R	
Digit with +	Digit with −	Digit
Character	Character	

Table 42. Interpretation of the T, I, and R picture characters (continued)

{	}	0
A	J	1
B	K	2
C	L	3
D	M	4
E	N	5
F	O	6
G	P	7
H	Q	8
I	R	9

T, I, and R specify the following values:

T   On input, T specifies that the characters { through | and the digits 0 through 9 represent positive values, and that the characters } through R represent negative values.

On output, T specifies that the associated position contains one of the characters { through | if the input data represents positive values, and one of the characters } through R if the input data represents negative values. The T can appear anywhere a '9' picture specification character occurs. For example:

```
dcl Credit picture 'ZZV9T';
```

The character representation is 4 characters; +21.05 is held as '210E', −0.07 is held as 'bb0P'.

I   On input, I specifies that the characters { through | and the digits 0 through 9 represent positive values.

On output, I specifies that the associated position contains one of the characters { through | if the input data represents positive values; otherwise, it contains one of the digits, 0 through 9.

R   On input, R specifies that the characters } through R represent negative values and the digits 0 through 9 represent positive values.

On output, R specifies that the associated position contains one of the characters } through R if the input data represents negative values; otherwise, it contains one of the digits 0 through 9. For example:

```
dcl X fixed decimal(3);
get edit (x) (P'R99');
```

sets X to 132 on finding '132' in the next three positions of the input stream, but sets X to −132 on finding 'J32'.

Zero replacement

Y   Specifies that a zero in the specified digit position is replaced unconditionally by the blank character.

Table 43 on page 328 shows examples of credit, debit, overpunched, and zero replacement characters.

Credit, debit, overpunched and zero replacement

Table 43. Examples of credit, debit, overpunched, and zero replacement characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(3)	−123	$Z.99CR	$1.23CR
FIXED(4,2)	12.34	$ZZV.99CR	$12.34bb
FIXED(4,2)	−12.34	$ZZV.99DB	$12.34DB
FIXED(4,2)	12.34	$ZZV.99DB	$12.34bb
FIXED(4)	1021	999I	102A
FIXED(4)	−1021	Z99R	102J
FIXED(4)	1021	99T9	10B1
FIXED(5)	00100	YYYYY	bb1bb
FIXED(5)	10203	9Y9Y9	1b2b3
FIXED(5,2)	000.04	YYYVY9	bbbb4

Exponent characters

The picture characters K and E delimit the exponent field of a numeric character specification that describes floating-point decimal numbers. The exponent field is the last field of a numeric character floating-point picture specification. The picture characters K and E cannot appear in the same specification.

K   Specifies that the exponent field appears to the right of the associated position. It does not specify a character in the numeric character data item.

E   Specifies that the associated position contains the letter E, which indicates the start of the exponent field.

The value of the exponent is adjusted in the character value so that the first significant digit of the first field (the mantissa) appears in the position associated with the first digit specifier of the specification (even if it is a zero suppression character).

Table 44 shows examples of exponent characters.

Table 44. Examples of exponent characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FLOAT(5)	.12345E06	V.99999E99	.12345E06
FLOAT(5)	.12345E−06	V.99999ES99	.12345E−06
FLOAT(5)	.12345E+06	V.99999KS99	.12345+06
FLOAT(5)	−123.45E+12	S999V.99ES99	−123.45E+12
FLOAT(5)	001.23E−01	SSS9.V99ESS9	+123.00Eb−3
FLOAT(5)	001.23E+04	ZZZV.99KS99	123.00+02
FLOAT(5)	001.23E+04	SZ99V.99ES99	+123.00E+02
FLOAT(5)	001.23E+04	SSSSV.99E−99	+123.00Eb02

Scaling factor

The picture character F specifies a picture scaling factor for fixed-point decimal numbers. It can appear only once at the right end of the picture specification.

```
►►──F─(─────────integer)──────────────────────────────────────────────►◄
         ┌─ + ─┐
         └─ ─ ─┘
```
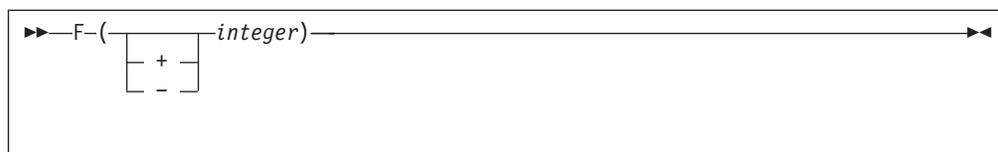
F   Specifies the picture scaling factor. The picture scaling factor specifies that the decimal point in the arithmetic value of the variable is that number of places to the right (if the picture scaling factor is positive) or to the left (if negative) of its assumed position in the character value.

The number of digits following the V picture character minus the integer specified with F must be between -128 and 127.

Table 45 shows examples of the picture scaling factor character.

Table 45. Examples of scaling factor characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(4,0)	1200	99F(2)	12
FIXED(7,0)	−1234500	S999V99F(4)	−12345
FIXED(5,5)	.00012	99F(−5)	12
FIXED(6,6)	.012345	999V99F(−4)	12345

Scaling factor

Chapter 16. Condition handling

While a PL/I program is running, certain events can occur for which you can do some testing, issue a response, or take recovery action. These events are called *conditions*, and are *raised* when detected. Conditions can be unexpected errors (e.g. overflow, input/output transmission error) or expected errors (e.g. end of an input file). Conditions can be raised directly in a program through the use of the SIGNAL statement (this can be very useful during testing).

Application control over conditions is accomplished through the *enablement* of conditions and the *establishment* of actions to be performed when an enabled condition is raised. When a condition is disabled, its raising causes no action; the program is unaware that the event was raised. The established action can be an ON-unit or the implicit action defined for the condition.

When an ON-unit is invoked, it is treated as a procedure without parameters. To assist you in making use of ON-units, built-in functions and pseudovariables are provided that you can use to inquire about the cause of a condition. Pseudovariables are often used for error correction and recovery. Built-in functions and pseudovariables are listed in Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365

The implicit action for many conditions is to raise the ERROR condition. This provides a common condition that can be used to check for a number of different conditions, rather than checking each condition separately. The ONCODE built-in function is particularly useful here, as it can be used to identify the specific circumstances that raised the conditions. Codes corresponding to the conditions and errors detected are listed in *Messages and Codes.*

Condition prefixes

You can specify whether or not some conditions are enabled or disabled. If a condtion is enabled, the compiler generates any extra code needed in order to detect the condition. If a condition is disabled, the compiler generates no extra code to detect it.

Disabling a condition is equivalent to asserting that the condition cannot occur; if it does, your program is in error.

Condition prefixes

For instance, if the SUBSCRIPTRANGE condition is enabled, the compiler generates extra code to ensure that any array index is within the bounds of its array. If the SUBSCRIPTRANGE condition is disabled, the extra code is not generated and using an invalid array index leads to unpredictable results.

If a condition is detected by hardware, disabling the condition has no effect.

Enabling and disabling can be specified for the eligible conditions by a condition prefix.

```
>>----(----condition----)--:----statement---------------------><
             |  ,  |           |           |
             |_____|           |____;_____|
```
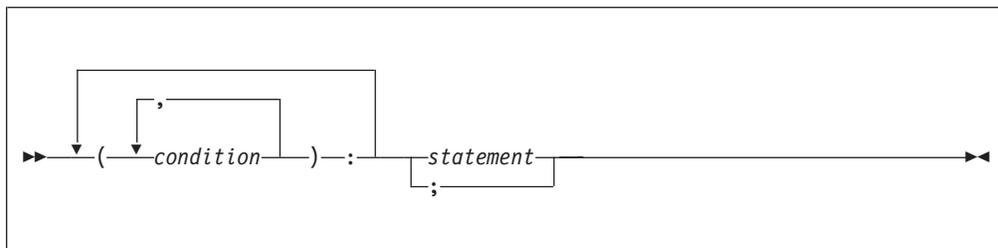
condition
>   Some conditions are always enabled, and cannot be disabled. Some are enabled unless you disable them, and some are disabled unless you enable them. The conditions are listed in Chapter 17, "Conditions," on page 339.

statement
>   Condition prefixes are not valid for DECLARE, DEFAULT, FORMAT, OTHERWISE, END, ELSE, ENTRY, and %statements. For information on the scope of condition prefixes, refer to "Scope of the condition prefix" on page 333.

In the following example `(size):` is the condition prefix. The conditional prefix indicates that the corresponding condition is enabled within the scope of the prefix.

```
(size):  L1:  X=(I**N) / (M+L);
```

Conditions can be enabled using the condition prefix specifying the condition name. They can be disabled using the condition prefix specifying the condition name preceded by NO without intervening blanks. Types and status of conditions are shown in Table 46.

Table 46. Classes and status of conditions

Class and conditions	Status
Computational (for data handling, expression evaluation, and computation)	
CONVERSION	Enabled by default
FIXEDOVERFLOW	Enabled by default
INVALIDOP	Enabled by default
OVERFLOW	Enabled by default
UNDERFLOW	Always enabled
ZERODIVIDE	Enabled by default
Input/Output	
ENDFILE	Always enabled
ENDPAGE	Always enabled
KEY	Always enabled
NAME	Always enabled
RECORD	Always enabled
TRANSMIT	Always enabled
UNDEFINEDFILE	Always enabled

Table 46. Classes and status of conditions (continued)

Class and conditions	Status
Program checkout (useful for developing/debugging a program)	
SIZE	Disabled by default
STRINGRANGE	Disabled by default
STRINGSIZE	Disabled by default
SUBSCRIPTRANGE	Disabled by default
Miscellaneous	
ANYCONDITION	Always enabled
AREA	Always enabled
ATTENTION	Always enabled
CONDITION	Always enabled
ERROR	Always enabled
FINISH	Always enabled
STORAGE	Always enabled

For information about the performance effects of enabling and disabling conditions, refer to the Programming Guide.

Scope of the condition prefix

The scope of a condition prefix (the part of the program to which it applies) is the statement or block to which the prefix is attached. The prefix does not necessarily apply to any procedures or ON-units that can be invoked in the execution of the statement.

A condition prefix attached to a PACKAGE, PROCEDURE, or BEGIN statement applies to all the statements up to and including the corresponding END statement. This includes other PROCEDURE or BEGIN statements nested within that block.

Condition status can be redefined within a block by attaching a prefix to statements within the block, including PROCEDURE and BEGIN statements (thus redefining the enabling or disabling of the condition within nested blocks). The redefinition applies only to the execution of the statement to which the prefix is attached. In the case of a nested PROCEDURE or BEGIN statement, it applies only to the block the statement defines, as well as any blocks contained within that block.

Raising conditions with OPTIMIZATION

When OPTIMIZATION is in effect, conditions for the same expression that appear multiple times can be raised only once. In the following example, SUBSCRIPTRANGE for IX can be raised only once:

```
  call P (55);
  (subscriptrange): P: proc (IX);
    dcl (Ar, Br, Cr) (10);
    Ar(IX) = Ar(IX) + Br(IX);
    T = Cr(IX);
  End P;
```

On-units

An implicit action exists for every condition. When an enabled condition is raised, the implicit action is executed unless an ON-unit for the enabled condition is established.

ON statement

The ON statement establishes the action to be executed for any subsequent raising of an enabled condition in the scope of the established condition.
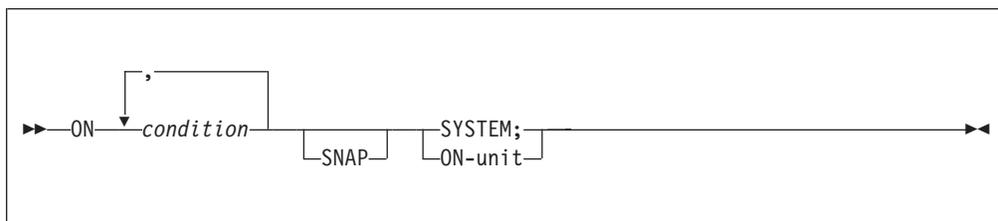
```
        ┌─────,──────┐
        │            │
►►─ON──▼─condition──┴─────────────┬──SYSTEM;─────────────────────►◄
                    └─SNAP─┘       └─ON-unit─┘
```

condition
> Is any one of those described in Chapter 17, "Conditions," on page 339 or defined with the CONDITION attribute.

SNAP
> Specifies that when the enabled condition is raised, diagnostic information relating to the condition is printed. The action of the SNAP option precedes the action of the ON-unit.
>
> If SNAP and SYSTEM are specified, the implicit action is followed immediately by SNAP information.

SYSTEM
> Specifies that the implicit action is taken. The implicit action is not the same for every condition, although for most conditions a message is printed and the ERROR condition is raised. The implicit action for each condition is given in Chapter 17, "Conditions," on page 339.

ON-unit
> Specifies the action to be executed when the condition is raised and is enabled. The action is defined by the statement or statements in the ON-unit itself. When the ON statement is executed, the ON-unit is said to be *established* for the specified condition. The ON-unit is not executed at the time the ON statement is executed; it is executed only when the specified enabled condition is raised.
>
> The ON-unit can be either a single unlabeled simple statement or an unlabeled begin-block. If it is a simple statement, it can be any statement except BEGIN, DECLARE, DEFAULT, DO, END, ENTRY, FORMAT, ITERATE, LEAVE, OTHERWISE, PROCEDURE, RETURN, SELECT, WHEN, or %statements. If the ON-unit is a begin-block, a RETURN statement can appear only within a procedure nested within the begin-block; a LEAVE statement can appear only within a do-group nested within the begin-block.
>
> Except for ON-units consisting only of either a semicolon (;) or the RESIGNAL statement, an ON-unit is treated as a procedure (without parameters) that is internal to the block in which it appears. Any names referenced in an ON-unit are those known in the environment in which the ON statement for that ON-unit was executed, rather than the environment in which the condition was raised.
>
> When execution of the ON-unit is complete, control generally returns to the block from which the ON-unit was entered. Just as with a procedure, control can be transferred out of an ON-unit by a GO TO statement. In this case, control is transferred to the point specified in the GO TO, and a normal return does not occur.
>
> The specific point to which control returns from an ON-unit varies for different conditions. Normal return for each condition is described in Chapter 17, "Conditions," on page 339.

Null ON-unit

The effect of a null statement ON-unit is to execute normal return from the condition.

Use of the null ON-unit is different from disabling a condition for two reasons:

- A null ON-unit can be specified for any condition, but not all conditions can be disabled.
- Disabling a condition, if possible, can save time by avoiding any checking for this condition. (If a null ON-unit is specified, the PL/I must still check for the raising of the condition.)

Scope of the ON-unit

The execution of an ON statement establishes an action specification for a condition. Once this action is established, it remains established throughout that block and throughout all dynamically descendent blocks until it is overridden by the execution of another ON statement or a REVERT statement or until termination of the block in which the ON statement is executed. (For information on dynamically descendent ON-units, refer to "Dynamically descendent ON-units.")

When another ON statement specifies the same conditions:

- If a later ON statement specifies the same condition as a prior ON statement and this later ON statement is executed in a block which is a dynamic descendant of the block containing the prior ON statement, the action specification of the prior ON statement is temporarily suspended, or stacked. It can be restored either by the execution of a REVERT statement, or by the termination of the block containing the later ON statement.

  When control returns from a block, all established actions that existed at the time of its activation are reestablished. This makes it impossible for a subroutine to alter the action established for the block that invoked the subroutine.

- If the later ON statement and the prior ON statement are internal to the same invocation of the same block, the effect of the prior ON statement is logically nullified. No reestablishment is possible, except through execution of another ON statement (or re-execution of an overridden ON statement).

Dynamically descendent ON-units

It is possible to raise a condition during execution of an ON-unit that specifies another ON-unit. An ON-unit entered because a condition is either raised or signalled in another ON-unit is a dynamically descendent ON-unit. A normal return from a dynamically descendent ON-unit reestablishes the environment of the ON-unit in which the condition was raised.

A loop can occur if an ERROR condition raised in an ERROR ON-unit executes the same ERROR ON-unit, raising the ERROR condition again. In any situation where a loop can cause the maximum nesting level to be exceeded, a message is printed and the application is terminated. To avoid a loop caused by this situation, use the following technique:

```
on error begin;
on error system;
   :
end;
```

ON-units for file variables

An ON statement that specifies a file variable refers to the file constant that is the current value of the variable when the ON-unit is established.

Example 1

```
dcl  F file,
     G file variable;
     G = F;
L1:  on endfile(G);
L2:  on endfile(F);
```

The statements labeled L1 and L2 are equivalent.

Example 2

```
declare FV file variable,
FC1 file,
FC2 file;
FV = FC1;
on endfile(FV) go to Fin;
 ⋮
FV = FC2;
read file(FC1) into (X1);
read file(FV) into (X2);
```

An ENDFILE condition raised during the first READ statement causes the ON-unit to be entered, because the ON-unit refers to file FC1. If the condition is raised in the second READ statement, however, the ON-unit is not entered, because this READ refers to file FC2.

Example 3

```
E:  procedure;
declare F1 file;
on endfile (F1) goto L1;
call E1 (F1);
 ⋮
E1:  procedure (F2);
declare F2 file;
on endfile (F2) go to L2;
read file (F1);
read file (F2);
end E1;
```

An end-of-file encountered for F1 in E1 causes the ON-unit for F2 in E1 to be entered. If the ON-unit in E1 was not specified, an ENDFILE condition encountered for either F1 or F2 would cause entry to the ON-unit for F1 in E.

Example 4

```
declare FV file variable,
FC1 file,
FC2 file;


do FV=FC1,FC2;
on endfile(FV) go to Fin;
end;
```

If an ON statement specifying a file variable is executed more than once, and the variable has a different value each time, a different ON-unit is established at each execution.

REVERT statement

Execution of the REVERT statement in a given block cancels the ON-unit for the condition that executed in that block. The ON-unit that was established at the time the block was activated is then reestablished. REVERT affects only ON statements that are internal to the block in which the REVERT statement occurs and that have been executed in the same invocation of that block.
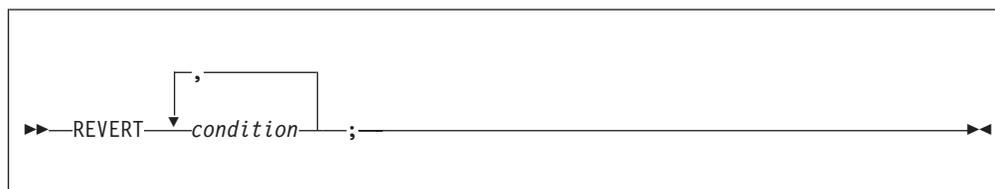
```
>>──REVERT──┬─condition─┬──;─────────────────────────────><
            └─────,◄────┘
```

condition
 Is any one of those described in Chapter 17, "Conditions," on page 339 or defined with the CONDITION attribute.

The REVERT statement cancels an ON-unit only if both of the following conditions are true:

1. An ON statement that is eligible for reversion, and that specifies a condition listed in the REVERT statement, was executed after the block was activated.
2. A REVERT statement with the specified condition was not previously executed in the same block.

If either of these two conditions is not met, the REVERT statement is treated as a null statement.

SIGNAL statement

You can raise a condition by means of the SIGNAL statement. This statement can be used in program testing to verify the action of an ON-unit and to determine whether the correct action is associated with the condition. The established action is taken unless the condition is disabled.

If the specified condition is disabled, the SIGNAL statement becomes equivalent to a null statement.

```
>>──SIGNAL──condition──;──────────────────────────────────><
```
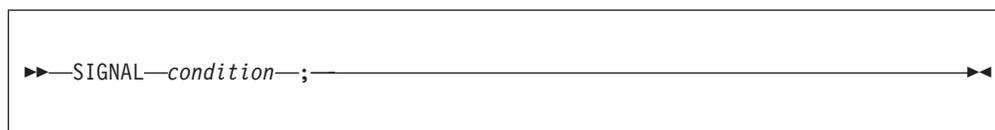
condition
 Is any condition described in Chapter 17, "Conditions," on page 339 or defined with the CONDITION attribute.

RESIGNAL statement

The RESIGNAL statement terminates the current ON-unit and allows another ON-unit for the same condition to get control. The processing continues as if the ON-unit executing the RESIGNAL did not exist and was never given control. It allows multiple ON-units to get control for the same condition.

```
►►──RESIGNAL──;──────────────────────────────────────────────────────────►◄
```

RESIGNAL is valid only within an ON-unit or its dynamic descendants.

Multiple conditions

A multiple condition is the simultaneous raising of two or more conditions.

The conditions for which a multiple condition can occur are:
RECORD, discussed in "RECORD condition" on page 351
TRANSMIT, discussed in "TRANSMIT condition" on page 355

The TRANSMIT condition is always processed first. The RECORD condition is ignored unless there is a normal return from the TRANSMIT ON-unit.

Multiple conditions are processed successively. When one of the following events occurs, no subsequent conditions are processed:

- Condition processing terminates the program, through implicit action for the condition, normal return from an ON-unit, or abnormal termination in the ON-unit.
- A GO TO statement transfers control from an ON-unit, so that a normal return is prevented.

CONDITION attribute

The CONDITION attribute specifies that the declared name identifies a programmer-defined condition.

```
►►──CONDITION──────────────────────────────────────────────────────────────►◄
```

A name that appears with the CONDITION condition in an ON, SIGNAL, or REVERT statement is contextually declared to be a condition name.

The default scope is EXTERNAL. An example of the CONDITION condition appears in 342.

Chapter 17. Conditions

This chapter describes conditions in alphabetic order. In general, the following information is given for each condition:

- **Status**—an indication of the enabled/disabled status of the condition at the start of the program, and how the condition can be disabled (if possible) or enabled. Table 46 on page 332 classifies the conditions into types, shows their status, and lists the conditions for disabling an enabled one.

- **Result**—the result of the operation that raised the condition. This applies when the condition is disabled as well as when it is enabled. In some cases, the result is undefined.

- **Cause and syntax**—a discussion of the condition, including the circumstances under which the condition can be raised. Raising conditions with the SIGNAL statement is discussed in "SIGNAL statement" on page 337.

- **Implicit action**—the action taken when an enabled condition is raised and no ON-unit is currently established for the condition.

- **Normal return**—the point to which control is returned as a result of the normal termination of the ON-unit. A GO TO statement that transfers control out of an ON-unit is an abnormal ON-unit termination. If a condition (except the ERROR condition) has been raised by the SIGNAL statement, the normal return is always to the statement immediately following SIGNAL.

- **Condition codes**—the codes corresponding to the conditions and errors for which the program is checked. An explanation for each code is given in the "Condition codes" chapter of the *Messages and Codes*.

ANYCONDITION condition

Status

ANYCONDITION is always enabled.

Result

The result is the same as for the underlying condition.

Cause and syntax

SIGNAL ANYCONDITION is not allowed. ANYCONDITION can be used only in ON (and REVERT) statements to establish (and cancel) an ON-unit which will trap any condition, including the CONDITION condition, that occurs in a block and which is not trapped by some other eligible ON-unit in that block.

In the following example, all ERROR conditions would be handled in the begin-block, the FINISH condition would be handled by the system, and all other conditions would be handled by the call to the routine named *handle_All_Others*.

```
on error
begin;
 :
 :
end;

on finish system;
on anycondition call Handle_all_others;
```

Note: To avoid infinite loops, the use of ON FINISH (as in the previous example) may be necessary when ON ANYCONDITION is used.

Note that when a condition is raised, the call stack will be walked (backwards) to search for a block that has an ON-unit for that condition. The search will stop when the first block with such an ON-unit or with an ON ANYCONDITION ON-unit is found. If no such ON-units are found and the implicit action for the condition is to promote it to ERROR, the stack will then (and only then) be walked again to search for an ON ERROR ON-unit.

You can use the ONCONDID built-in function in an ANYCONDITION ON-unit to determine what condition is being handled, and the ONCONDCOND built-in function to determine the name of the CONDITION condition. Other ON built-in functions, such as ONFILE, can be used to determine the exact cause and other related information. These built-in functions are listed in Chapter 19, "Built-in functions, pseudovariables, and subroutines.," on page 365.

```
►►──ANYCONDITION────────────────────────────────────────────────►◄
```
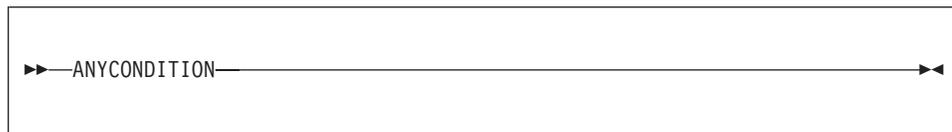
Abbreviation

ANYCOND

Implicit action

The implicit action is that of the underlying condition.

Normal return

Normal return is the same as for the underlying condition.

Condition codes

There are no condition codes unique to the ANYCONDITION.

AREA condition

Status

AREA is always enabled.

Result

An attempted allocation or assignment that raises the AREA condition has no effect.

Cause and syntax

The AREA condition is raised in either of the following circumstances:

- When an attempt is made to allocate a based variable within an area that contains insufficient free storage for the allocation to be made.

- When an attempt is made to perform an area assignment, and the target area contains insufficient storage to accommodate the allocations in the source area

```
►►──AREA──────────────────────────────────────────────►◄
```

Implicit action

A message is printed and the ERROR condition is raised.

Normal return

On normal return from the ON-unit, the action is as follows:

- If the condition was raised by an allocation and the ON-unit is a null ON-unit, the allocation is not attempted again.

- If the condition was raised by an allocation, the allocation is attempted again. Before the attempt is made, the area reference is reevaluated. Thus, if the ON-unit has changed the value of a pointer qualifying the reference to the inadequate area so that it points to another area, the allocation is attempted again within the new area.

- If the condition was raised by an area assignment, or by a SIGNAL statement, execution continues from the point at which the condition was raised.

Condition codes

360, 361, 362

ATTENTION condition

Status

ATTENTION is always enabled.

Result

Raising the condition causes an ATTENTION ON-unit to be entered. If there is no ATTENTION ON-unit, the application is terminated.

Cause and syntax

The ATTENTION condition is raised when the user hits a specific key combination to interrupt an application. The specific key is determined by the operating system as follows:

- On Windows, CTRL-BRK and CTRL-C. No ATTENTION ON-units will be driven on Windows as a result of the user entering CTRL-BRK or CTRL-C key combinations. The implicit action will be taken.

- On the host, the ATTN key, if available.

The condition can also be raised by a SIGNAL ATTENTION statement.

```
►►──ATTENTION──────────────────────────────────────────────►◄
```

Abbreviation
ATTN

Implicit action
The application is terminated.

Normal return
On return from an ATTENTION ON-unit, processing is resumed at a point in the program immediately following the point at which the condition was raised.

Condition code
400

CONDITION condition

Status
CONDITION is always enabled.

Result
The CONDITION condition allows you to establish an ON-unit that will be executed whenever a SIGNAL statement for the appropriate CONDITION condition is executed.

As a debugging aid, the CONDITION condition can be used to establish an ON-unit that prints information about the current status of the program.

Cause and syntax
The CONDITION condition is raised by a SIGNAL statement. The name specified in the SIGNAL statement determines which CONDITION condition is raised. The ON-unit can be executed from any point in the program through placement of a SIGNAL statement. Normal rules of name scope apply. A condition name is external by default, but can be declared INTERNAL.

The following example shows the use of the CONDITION condition.

```
dcl Test condition;


on condition (Test)
begin;
:
end;
```

The begin-block is executed whenever the following statement is executed:

```
  signal condition (Test);
```

```
►►──CONDITION──(name)────────────────────────────────────►◄
```

Abbreviation

COND

Implicit action

A message is printed and execution continues with the statement following SIGNAL.

Normal return

Execution continues with the statement following the SIGNAL statement.

Condition code

500

CONVERSION condition

Status

CONVERSION is enabled throughout the program, except within the scope of the NOCONVERSION condition prefix. You can use the ONSOURCE, ONCHAR, ONGSORCE and ONWSOURCE pseudovariables in CONVERSION ON-units to correct conversion errors.

Result

When CONVERSION is raised, the contents of the entire result field are undefined.

Cause and syntax

The CONVERSION computational condition is raised whenever an invalid conversion is attempted on character, widechar or graphic data. This attempt can be made internally or during an input/output operation. For example, the condition is raised when:

- A character other than 0 or 1 exists in character data being converted to bit data.

- A character value being converted to a numeric character field, or to a coded arithmetic value, contains characters which are not the representation of an optionally signed arithmetic constant, or an expression to represent a complex constant.

- A graphic (DBCS) string being converted to character contains a graphic which cannot be converted to SBCS.

- A value being converted to a character pictured item contains characters not allowed by the picture specification.

All conversions of character data are carried out character-by-character in a left-to-right sequence. The condition is raised for each invalid character. The condition is also raised if all the characters are blank, with the following exceptions:

- For input with the F-format item, a value of zero is assumed

- For input with the E-format item, be aware that sometimes the ON-unit will be repeatedly entered.

When an invalid character is encountered, the current action specification for the condition is executed (provided that CONVERSION is not disabled). If the action specification is an ON-unit, the invalid character can be replaced within the unit.

CONVERSION

- For character source data, use the ONSOURCE or ONCHAR pseudovariables.
- For widechar source data, use the ONWSOURCE or ONWCHAR pseudovariables.
- For graphic source data, use the ONGSOURCE pseudovariable.

If the CONVERSION condition is raised and it is disabled, the program is in error.

If the CONVERSION condition is raised during conversion from graphic data to nongraphic data, the ONCHAR and ONSOURCE built-in functions do not contain valid source data. The ONGSOURCE built-in function contains the original graphic source data. The graphic conversion is retried if the ONGSOURCE pseudovariable is used in the CONVERSION ON-unit to attempt to fix the graphic data that raised the CONVERSION condition. If the ONGSOURCE pseudovariable is not used in the CONVERSION ON-unit, the ERROR condition is raised.

```
►►──CONVERSION──────────────────────────────────────────────────►◄
```
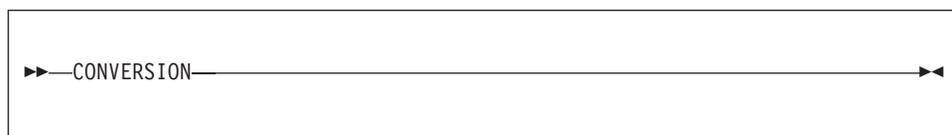
Abbreviation
   CONV

Implicit action
   A message is printed and the ERROR condition is raised.

Normal return
   If CONVERSION was raised on a character string source (not graphic source) and either ONSOURCE or ONCHAR pseudovariables are used in the ON-unit, the program retries the conversion on return from the ON-unit.

   If CONVERSION was raised on a graphic source and the ONGSOURCE pseudovariable is used in the ON-unit, the program retries the conversion on return from the ON-unit.

   If CONVERSION was raised on a widechar source and the ONWSOURCE pseudovariable is used in the ON-unit, the program retries the conversion on return from the ON-unit.

   If the conversion error is not corrected using these pseudovariables, the program loops.

Condition codes
   600-672

ENDFILE condition

Status
   The ENDFILE condition is always enabled.

Result
   If the specified file is not closed after the condition is raised, subsequent GET or READ statements to the file are unsuccessful and cause additional ENDFILE conditions to be raised.

Cause and syntax

The ENDFILE input/output condition can be raised during an operation by an attempt to read past the end of the file specified in the GET or READ statement. It applies only to SEQUENTIAL INPUT, SEQUENTIAL UPDATE, and STREAM INPUT files.

In record-oriented data transmission, ENDFILE is raised whenever an end of file is encountered during the execution of a READ statement.

In stream-oriented data transmission, ENDFILE is raised during the execution of a GET statement if an end of file is encountered either before any items in the GET statement data list have been transmitted or between transmission of two of the data items. If an end of file is encountered while a data item is being processed, or if it is encountered while an X-format item is being processed, the ERROR condition is raised.

```
►►──ENDFILE──(file-reference)──────────────────────►◄
```

Implicit action

A message is printed and the ERROR condition is raised.

Normal return

Execution continues with the statement immediately following the GET or READ statement that raised the ENDFILE.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition code

70

ENDPAGE condition

Status

ENDPAGE is always enabled.

Result

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option (default is 61) so that it is possible to continue writing on the same page. The ON-unit can start a new page by execution of a PAGE option or a PAGE format item, which sets the current line to one.

If the ON-unit does not start a new page, the current line number can increase indefinitely. If a subsequent LINE option or LINE format item specifies a line number that is less than or equal to the current line number, ENDPAGE is not raised, but a new page is started with the current line set to one. An exception is that if the current line number is equal to the specified line number, and the file is positioned on column one of the line, ENDPAGE is not raised.

If ENDPAGE is raised during data transmission, on return from the ON-unit, the data is written on the current line, which might have been changed by the ON-unit. If ENDPAGE results from a LINE or SKIP option, on return from the ON-unit, the action specified by LINE or SKIP is ignored.

Cause and syntax

The ENDPAGE input/output condition is raised when a PUT statement results in an attempt to start a new line beyond the limit specified for the current page. This limit can be specified by the PAGESIZE option in an OPEN statement; if PAGESIZE has not been specified, a default limit of 60 is applied. The attempt to exceed the limit can be made during data transmission (including associated format items, if the PUT statement is edit-directed), by the LINE option, or by the SKIP option. ENDPAGE can also be raised by a LINE option or LINE format item that specified a line number less than the current line number. ENDPAGE is raised only once per page, except when it is raised by the SIGNAL statement.

```
►►──ENDPAGE──(file-reference)──────────────────────────────────►◄
```

Implicit action

A new page is started. If the condition is signalled, execution is unaffected and continues with the statement following the SIGNAL statement.

Normal return

Execution of the PUT statement continues in the manner described above.

Condition code

90

ERROR condition

Status

ERROR is always enabled.

Result

An error message is issued if no ON-unit is active when the ERROR condition arises or if the ON-unit does not use a GOTO (to exit the block) to recover from the condition.

Cause and syntax

The ERROR condition is the implicit action for many conditions. This provides a common condition that can be used to check for a number of different conditions, rather than checking each condition separately.

The ERROR condition is raised under the following circumstances:

- As a result of the implicit action for a condition, which is to raise the ERROR condition
- As a result of the normal return action for some conditions, such as SUBSCRIPTRANGE CONVERSION or when no retry is attempted
- As a result of an error (for which there is no other PL/I-defined condition) during program execution
- As a result of a SIGNAL ERROR statement

```
▶▶──ERROR──────────────────────────────────────────────────────────▶◀
```

Implicit action

The message is printed and the FINISH condition is raised.

Normal return

The implicit action is taken.

Condition codes

All codes 1000 and above are ERROR conditions.

FINISH condition

Status

FINISH is always enabled.

Result

Control passes to the FINISH ON-unit and processing continues.

Cause and syntax

The FINISH condition is raised during execution of a statement that would terminate the procedures. The following actions take place:

- If the termination is normal—the FINISH ON-unit, if established, is given control only if the main procedure is PL/I.
- If the termination is abnormal—the FINISH ON-unit, if established in an active block, is given control.

```
▶▶──FINISH─────────────────────────────────────────────────────────▶◀
```

Implicit action

- If the condition is raised in the major task, no action is taken and processing continues from the point where the condition was raised.
- If the condition is raised as part of the implicit action for another condition, the program is terminated.

Normal return

Processing resumes at the point where the condition was raised. This point is the statement following the SIGNAL statement if the conditions was signalled.

Condition code

4

FIXEDOVERFLOW condition

Status

FIXEDOVERFLOW is enabled throughout the program, except within the scope of the NOFIXEDOVERFLOW condition prefix.

Result

The result of the invalid FIXED DECIMAL operation is undefined.

FIXEDOVERFLOW

Cause and syntax

The FIXEDOVERFLOW computational condition is raised when the length of the result of a FIXED DECIMAL arithmetic operation exceeds the maximum length allowed by the implementation.

The FIXEDOVERFLOW condition is not raised for FIXED BINARY operations.

The FIXEDOVERFLOW condition differs from the SIZE condition in that SIZE is raised when a result exceeds the declared size of a variable, while FIXEDOVERFLOW is raised when a result exceeds the maximum allowed by the computer.

If the FIXEDOVERFLOW condition is raised and it is disabled, the program is in error.

```
►►──FIXEDOVERFLOW──────────────────────────────────────────────►◄
```
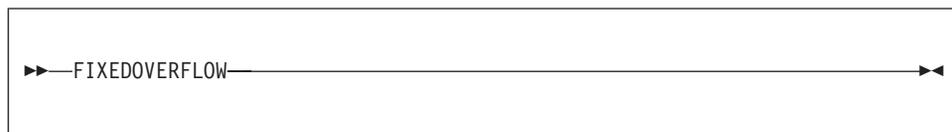
Abbreviation
FOFL

Implicit action
A message is printed and the ERROR condition is raised.

Normal return
Control returns to the point immediately following the point at which the condition was raised.

Condition code
310

Note: If the SIZE condition is disabled, an attempt to assign an oversize number to a fixed decimal variable can raise the FIXEDOVERFLOW condition.

INVALIDOP condition

Status
INVALIDOP is enabled throughout the program, except within the scope of the NOINVALIDOP condition prefix.

Result
The result of the invalid operation is undefined.

Cause and syntax
The INVALIDOP computational condition is raised when any of the following are detected during the evaluation of IEEE floating-point expressions.
* Subtraction of two infinities
* Multiplication of infinity by 0
* Division of two infinities
* Division of zero by zero
* Invalid floating-point data

```
►►──INVALIDOP─────────────────────────────────────────────►◄
```

Implicit action
The ERROR condition is raised.

Normal return
A message is printed and the ERROR condition is raised.

Condition code
290

KEY condition

Status
KEY is always enabled.

Result
The keyed record is undefined, and the statement in which it appears is ignored.

Cause and syntax
The KEY input/output condition is raised when a record with a specified key cannot be found. The condition can be raised only during operations on keyed records. It is raised for the condition codes listed below.

When a LOCATE statement is used for the data set, the KEY condition for this LOCATE statement is not raised until the next WRITE or LOCATE statement for the file, or when the file is closed.

```
►►──KEY──(file-reference)──────────────────────────────────►◄
```

Implicit action
A message is printed and the ERROR condition is raised.

Normal return
Control passes to the statement immediately following the statement that raised KEY.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition codes
50-58

NAME condition

Status
NAME is always enabled.

Result
The named data is undefined.

Cause and syntax

The NAME input/output condition can be raised only during execution of a data-directed GET statement with the FILE option. It is raised in any of the following situations:

- The syntax is not correct, as described under "Syntax of data-directed data" on page 290.
- The name is missing or invalid, for example:
  - No counterpart is found in the data list.
  - If there is no data list, the name is not known in the block.
  - A qualified name is not fully qualified.
  - DBCS contains a byte outside the valid range X'41' to X'FE'.
- A subscript list is missing or invalid, for example.
  - A subscript is missing.
  - The number of subscripts is incorrect.
  - More than 10 digits are in a subscript (leading zeros ignored).
  - A subscript is outside the allowed range of the current allocation of the variable.

You can retrieve the incorrect data field by using the built-in function DATAFIELD in the ON-unit.

```
►►──NAME──(file-reference)──────────────────────────────────►◄
```
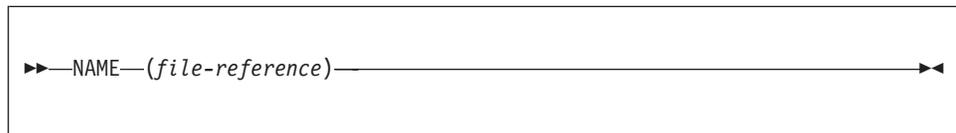
Implicit action

The incorrect data field is ignored, a message is printed, and execution of the GET statement continues.

Normal return

The execution of the GET statement continues with the next name in the stream.

Condition code

10

OVERFLOW condition

Status

OVERFLOW is enabled throughout the program, except within the scope of the NOOVERFLOW condition prefix.

Result

The value of such an invalid floating-point number is undefined.

Cause and syntax

The OVERFLOW computational condition is raised when the magnitude of a floating-point number exceeds the maximum allowed.

The OVERFLOW condition differs from the SIZE condition in that SIZE is raised when a result exceeds the declared size of a variable, while OVERFLOW is raised when a result exceeds the maximum allowed by the computer.

If the OVERFLOW condition is raised and it is disabled, the program is in error.

```
►►──OVERFLOW───────────────────────────────────────────────────►◄
```

Abbreviation

OFL

Implicit action

A message is printed and the ERROR condition is raised.

Normal return

The ERROR condition is raised.

Condition code

300

RECORD condition

Status

RECORD is always enabled.

Result

The length prefix for the specified file can be inaccurately transmitted.

Cause and syntax

The RECORD input/output condition is raised if the specified record is truncated. The condition can be raised only during a READ, WRITE, LOCATE, or REWRITE operation.

If the SCALARVARYING option is applied to the file (it must be applied to a file using locate mode to transmit varying-length strings), a 2-byte length prefix is transmitted with an element varying-length string. The length prefix is not reset if the RECORD condition is raised. If the SCALARVARYING option is not applied to the file, the length prefix is not transmitted. On input, the current length of a varying-length string is set to the shorter of the record length and the maximum length of the string.

```
►►──RECORD──(file-reference)──────────────────────────────────►◄
```

Implicit action

A message is printed and the ERROR condition is raised.

Normal return

Execution continues with the statement immediately following the one for which RECORD was raised.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition codes

20-24

SIZE condition

Status

SIZE is disabled throughout the program, except within the scope of the SIZE condition prefix.

Result

The result of the assignment is undefined.

Cause and syntax

The SIZE computational condition is raised only when high-order (that is, leftmost) significant binary or decimal digits are lost in an attempted assignment to a variable or an intermediate result or in an input/output operation. This loss can result from a conversion involving different data types, different bases, different scales, or different precisions. Even if the SIZE condition is disabled, any conversion that is not done inline may cause the condition to be raised.

SIZE is raised when the size of the value being assigned to a data item exceeds the declared (or default) size of the data item, even if this is not the actual size of the storage that the item occupies. For example, a fixed binary item of precision (20) occupies a fullword in storage, but SIZE is raised if a value whose size exceeds FIXED BINARY(20) is assigned to it.

Because checking sizes requires substantial overhead in both storage space and run time, the SIZE condition is primarily used for program testing. It can be removed from production programs. For more information on test and production application programs, refer to the Programming Guide.

The SIZE condition differs from the FIXEDOVERFLOW condition in that FIXEDOVERFLOW is raised when the size of a calculated fixed-point value exceeds the maximum allowed by the implementation. SIZE can be raised on assignment of a value regardless of whether or not FIXEDOVERFLOW was raised in the calculation of that value.

If the SIZE condition is raised and it is disabled, the program is in error.
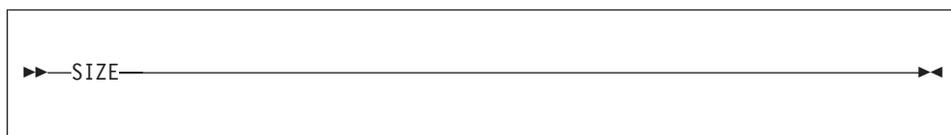
```
►►──SIZE──────────────────────────────────────────────────────►◄
```

Implicit action

A message is printed and the ERROR condition is raised.

Normal return

Control returns to the point immediately following the point at which the condition was raised.

Condition codes

340, 341

STORAGE condition

Status

STORAGE is always enabled.

Result

The result depends on the type of variable for which attempted storage allocation raised the condition.

- After an ALLOCATE statement for a controlled variable, that variable's generation is not allocated. A reference to that controlled variable results in accessing the generation (if any) before the ALLOCATE statement.
- After an ALLOCATE statement for a based variable, the variable is not allocated and its associated pointer is undefined.
- After an ALLOCATE built-in function for a based variable, the variable is not allocated and the use of the associated pointer is undefined.

Cause and syntax

The STORAGE condition allows the program to gain control for the failure of an ALLOCATE built-in function or ALLOCATE statement that attempted to allocate BASED or CONTROLLED storage outside of an AREA. Failure of an ALLOCATE statement in an AREA raises the AREA condition.

Failure of the AUTOMATIC built-in function does not raise the STORAGE condition.

The ON-unit for the STORAGE condition can attempt to free allocated storage. If the ON-unit is unable to free sufficient storage, it can provide the necessary steps to terminate the program without losing diagnostic information.
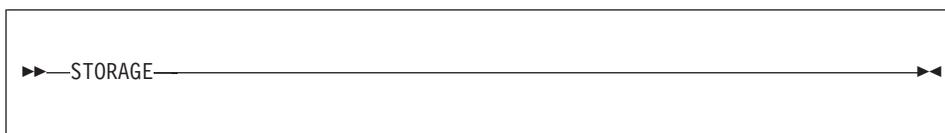
```
►►──STORAGE──────────────────────────────────────────►◄
```

Implicit action

The ERROR condition is raised.

Normal return

The ERROR condition is raised.

Condition codes

450, 451

STRINGRANGE condition

Status

STRINGRANGE is disabled throughout the program, except within the scope of the STRINGRANGE condition prefix.

Result

The value of the specified SUBSTR is altered.

Cause and syntax

The STRINGRANGE program-checkout condition is raised whenever the values of the arguments to a SUBSTR reference fail to comply with the rules described for the SUBSTR built-in function. It is raised for each reference to an invalid argument.

```
►►──STRINGRANGE──────────────────────────────────────►◄
```
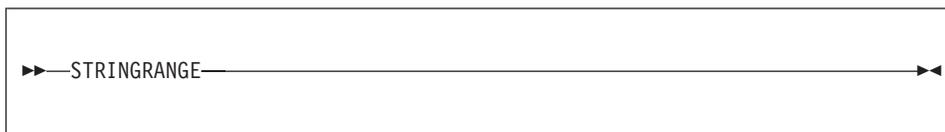
Abbreviation

STRG

Implicit action

A message is printed and processing continues as described for normal return.

Normal return

Execution continues with a revised SUBSTR reference for which the value is defined as follows:

Assuming that the length of the source string (after execution of the ON-unit, if specified) is k, the starting point is i, and the length of the substring is j:

- If i is greater than k, the value is the null string.
- If i is less than or equal to k, the value is that substring beginning at the mth character, bit, widechar or graphic of the source string and extending n characters, bits, widechars or graphics, where m and n are defined by:

  ```
  M = max( I,1 )
  N = max( 0,min( J + min(I,1) - 1,K - M + 1 ))
  ```

  if J is specified.

  ```
  N = K - M + 1
  ```

  if J is not specified.

This means that the new arguments are forced within the limits.

The values of i and j are established before entry to the ON-unit. They are not reevaluated on return from the ON-unit.

The value of k might change in the ON-unit if the first argument of SUBSTR is a varying-length string. The value n is computed on return from the ON-unit using any new value of k.

Condition code

350

STRINGSIZE condition

Status

STRINGSIZE is disabled throughout the program, except within the scope of the STRINGSIZE condition prefix.

Result

After the condition action, the truncated string is assigned to its target string. The right-hand characters, bits, widechars or graphics of the source string are truncated so that the target string can accommodate the source string.

Cause and syntax

The STRINGSIZE program-checkout condition is raised when you attempt to assign a string to a target with a shorter maximum length.

```
►►──STRINGSIZE──────────────────────────────────────────►◄
```
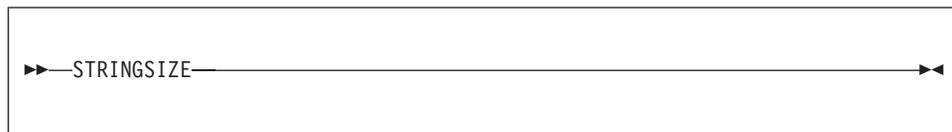
Abbreviation

STRZ

Implicit action

A message is printed and processing continues.

Normal return
Execution continues from the point at which the condition was raised.

Condition codes
150, 151

SUBSCRIPTRANGE condition

Status
SUBSCRIPTRANGE is disabled throughout the program, except within the scope of the SUBSCRIPTRANGE condition prefix.

Result
When SUBSCRIPTRANGE has been raised, the value of the invalid subscript is undefined, and, hence, the reference is also undefined.

Cause and syntax
The SUBSCRIPTRANGE program-checkout condition is raised whenever a subscript is evaluated and found to lie outside its specified bounds. The order of raising SUBSCRIPTRANGE relative to evaluation of other subscripts is undefined.

```
▶▶──SUBSCRIPTRANGE──────────────────────────────────────▶◀
```
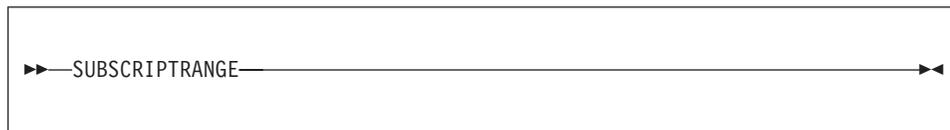
Abbreviation
SUBRG

Implicit action
A message is printed and the ERROR condition is raised.

Normal return
Normal return from a SUBSCRIPTRANGE ON-unit raises the ERROR condition.

Condition codes
520

TRANSMIT condition

Status
TRANSMIT is always enabled.

Result
Raising the TRANSMIT condition indicates that any data transmitted is potentially incorrect.

Cause and syntax
The TRANSMIT input/output condition can be raised during any input/output operation. It is raised by an uncorrectable transmission error of a record (or of a block, if records are blocked), which is an input/output error that could not be corrected during execution. It can be caused by a damaged recording medium, or by incorrect specification or setup.

During input, TRANSMIT is raised after transmission of the potentially incorrect record. If records are blocked, TRANSMIT is raised for each subsequent record in the block.

During output, TRANSMIT is raised after transmission. If records are blocked, transmission occurs when the block is complete rather than after each output statement.

When a spanned record is being updated, the TRANSMIT condition is raised on the last segment of a record only. It is not raised for any subsequent records in the same block, although the integrity of these records cannot be assumed.

```
►►──TRANSMIT──(file-reference)──────────────────────────────────────►◄
```

Implicit action
A message is printed and the ERROR condition is raised.

Normal return
Processing continues as though no error had occurred, allowing another condition (for example, RECORD) to be raised by the statement or data item that raised the TRANSMIT condition.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition codes
40-46

UNDEFINEDFILE condition

Status
UNDEFINEDFILE is always enabled.

Result
Specified files are undefined to the application program.

Cause and syntax
The UNDEFINEDFILE input/output condition is raised whenever an unsuccessful attempt to open a file is made. If the attempt is made by means of an OPEN statement that specifies more than one file, the condition is raised after attempts to open all specified files.

If UNDEFINEDFILE is raised for more than one file in the same OPEN statement, ON-units are executed according to the order of appearance (taken from left to right) of the file names in that OPEN statement.

If UNDEFINEDFILE is raised by an implicit file opening in a data transmission statement, upon normal return from the ON-unit, processing continues with the remainder of the data transmission statement. If the file was not opened in the ON-unit, the statement cannot continue and the ERROR condition is raised.

The UNDEFINEDFILE condition is raised not only by conflicting attributes (such as DIRECT with PRINT), but also by the following:

- Block size smaller than record size (except when records are spanned)
- LINESIZE exceeding the maximum allowed
- KEYLENGTH zero or not specified for creation of INDEXED data sets
- Specifying a KEYLOC option, for an INDEXED data set, with a value resulting in KEYLENGTH + KEYLOC exceeding the record length

- Specifying a V-format logical record length of less than 18 bytes for STREAM data sets
- Specifying a block size that is not an integral multiple of the record size for FB-format records
- Specifying a logical record length that is not at least 4 bytes smaller than the specified block size for VB-format records.

```
►►──UNDEFINEDFILE──(file-reference)──────────────────────────────────────►◄
```

Abbreviation
UNDF

Implicit action
A message is printed and the ERROR condition is raised.

Normal return
Upon the normal completion of the final ON-unit, control is given to the statement immediately following the statement that raised the condition.

Condition codes
80-89, 91-95

UNDERFLOW condition

Status
UNDERFLOW is enabled throughout the program, except within the scope of the NOUNDERFLOW condition prefix.

Result
The invalid floating-point value is set to 0.

Cause and syntax
The UNDERFLOW computational condition is raised when the magnitude of a floating-point number is smaller than the minimum allowed. save UNDERFLOW is not raised when equal numbers are subtracted (often called significance error).

The expression $X^{(-Y)}$ (where Y>0) can be evaluated by taking the reciprocal of X^Y; hence, the OVERFLOW condition might be raised instead of the UNDERFLOW condition.

```
►►──UNDERFLOW──────────────────────────────────────────────────────────►◄
```

Abbreviation
UFL

Implicit action
Unless running under IEEE on z/OS, a message is printed, and execution continues from the point at which the condition was raised; under IEEE on z/OS, a message is printed and the ERROR condition is raised.

Normal return

Unless running under IEEE on z/OS, control returns to the point immediately following the point at which the condition was raised; under IEEE on z/OS, the ERROR condition is raised.

Condition code

330

ZERODIVIDE condition

Status

ZERODIVIDE is enabled throughout the program, except within the scope of the NOZERODIVIDE condition prefix.

Result

The result of a division by zero is undefined.

Cause and syntax

The ZERODIVIDE computational condition is raised when an attempt is made to divide by zero. This condition is raised for fixed-point and floating-point division. If the numerator of a floating-point divide is also zero, INVALIDOP is raised.

If the ZERODIVIDE condition is raised and it is disabled, the program is in error.

```
►►──ZERODIVIDE────────────────────────────────────────────────►◄
```
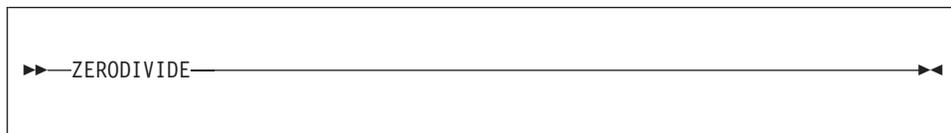
Abbreviation

ZDIV

Implicit action

A message is printed and the ERROR condition is raised.

Normal return

The ERROR condition is raised.

Condition code

320

Chapter 18. Multithreading facility

A PL/I program is a set of one or more procedures. The program normally executes as a single execution unit, or as part of a single execution unit. When a procedure invokes another procedure, control is passed to the invoked procedure, and execution of the invoking procedure is suspended until the invoked procedure passes control back. This execution with a single flow of control is *synchronous* flow.

When using the PL/I multithreading facility, the invoking procedure does not relinquish control to the invoked procedure. Instead, an additional flow of control is established so that both procedures are executed concurrently. The execution of such concurrent procedures is called *asynchronous* flow.

The PL/I multithreading facility allows the execution of parts of a PL/I program asynchronously in multiple threads. A **thread** is a unit of work that competes for the resources of the computing system. A thread is not the equivalent of a task in OS PL/I. Except for the main thread in a program, a thread is always independent of and unrelated to other threads in the program. When a procedure invokes another procedure as a thread, this action is known as *attaching*, or creating the thread.

Execution of one or more threads occurs in a **process**, which can be thought of as a PL/I program. PL/I does not provide the capabilities to create and manage multiple processes or tasks, but it does allow creation and management of multiple threads in a single program (process).

There is normally one application thread per process. Multiple threads can be attached (created) to:

* Perform a piece of work in a shorter elapsed time. Such applications can be batch applications that are not interacting with the user. For example, one procedure could attach a thread which would compile a PL/I program.
* Perform high response parts of a program in one thread and I/O parts in another thread, and typical response parts in a third.
* Use computing system resources that might be idle. These resources can include I/O devices as well as the CPUs.
* Implement real-time multi-user applications where the response time is critical.
* Isolate independent pieces of work for reliability. That is, the failure of a part of a job can be isolated while other independent parts are processed.

Note: Operating system services must not be directly used when PL/I provides the appropriate function.

359

Creating a thread

A thread:

- Is an independent unit of work
- Executes concurrently and independently of other threads in the process and system
- Can attach other threads
- Can wait for a thread to complete
- Can stop itself or another thread

Any procedures or functions synchronously invoked by the thread become part of the thread's execution.

ATTACH statement

A thread is attached (or created) by the execution of the ATTACH statement. You can specify explicit characteristics for the thread if the defaults are not desired.

```
►►──ATTACH──entry-reference─────────────────────────────────────────────►

                        └─THREAD──(task-reference)─┘

►──────────────────────────────────────;──────────────────────────────►◄
    └─ENVIRONMENT──(──────────────────)─┘
                    └─TSTACK(expression)─┘
```
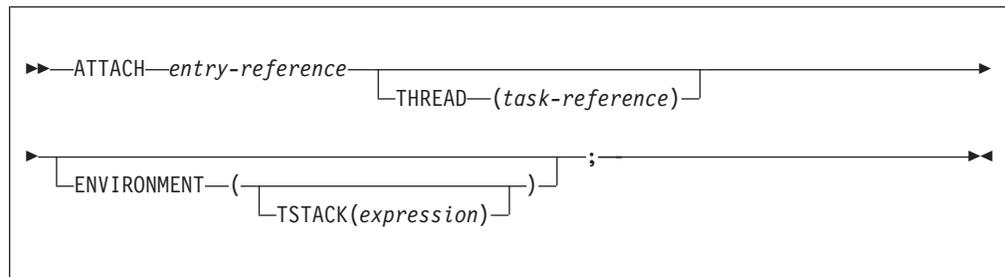
entry
> Specifies the name of a limited entry variable, or the name of an external entry or level-1 procedure. It cannot be the name of an internal procedure or a fetchable procedure. The ATTACHed entry must be declared as having no parameters or as having exactly one BYVALUE POINTER parameter. However, you can fetch a procedure, assign it to a limited entry variable, and then attach the entry variable as a thread.
>
> Arguments can be passed to the new thread just as you would pass arguments to a synchronous entry in a CALL statement.

THREAD (task reference)
> Specifies the name of a task variable that becomes associated with the thread. The task variable can then be used to refer to the thread.
>
> Unless explicitly declared, the named variable is given a contextual declaration.
>
> If the THREAD option is not specified, the attached thread cannot be stopped or waited upon by another thread.
>
> If a thread is attached with the THREAD option, it should be detached using the DETACH statement (see "Detaching a thread" on page 362) to free all the system resources associated with the thread.
>
> Operating system services must not be used directly to create a thread.

ENVIRONMENT (abbrev: ENV)
> Specifies environmental characteristics and is usually operating system dependent.

TSTACK (expression)

> On Intel, specifies the size of the stack to be used for the attached thread. The expression should be FIXED BINARY(31,0). If the stack size is not specified, the run-time default will be used.

> On z/OS, TSTACK is ignored, and the size of the stack is determined by LE.

Examples

```
attach input (File1);

attach input (File2)
   thread (T2);
```

> An attached procedure may have any supported linkage.

Terminating a thread

A thread is terminated when any of the following occurs:

- The END statement corresponding to the first procedure (the initial procedure in the thread) is reached.
- The ERROR condition is raised and either there is no ERROR ON-unit or the ERROR ON-unit terminates normally (reaches the END statement for the ON-unit or executes a RESIGNAL statement).
- The EXIT statement is executed in any procedure within the thread.
- The initial thread in the program terminates.
- The STOP statement is executed in any thread within the program. This stops the entire program, causing all threads, including the main thread, to be terminated.

The FINISH condition is raised only in the thread initiating program termination. Any ON-units established within the thread are given control before the thread actually terminates.

Except as noted above, when a thread terminates, no other threads are terminated, unless the thread being terminated is the main thread. In that case, all other threads are stopped and terminated before the main thread is terminated.

When a thread terminates, only its stack space is released. All other resources such as allocated storage, open files, etc. remain intact. The user must ensure that any resources a thread has acquired are released and open files are closed, unless they are needed by other threads that are still active.

When the main thread terminates, all resources are released and files are closed.

Waiting for a thread to complete

To wait for a thread, use the WAIT statement.

```
►►──WAIT──THREAD──(task-reference)──;──────────────────────►◄
```

THREAD (task-reference)
> The THREAD option specifies the thread upon which the process is waiting.

Waiting for a thread

The current thread is suspended until the specified thread terminates. As soon as the specified thread has terminated, the current thread resumes.

```
WAIT THREAD (TI1);
```

Detaching a thread

The DETACH statement should be used to free the system resources associated with a thread that was attached with the THREAD option.

```
►►──DETACH──THREAD──(task-reference)──;──────────────────────────►◄
```

THREAD (task-reference)

The THREAD option specifies the thread to be detached.

Normally, this statement would be executed immediately after the WAIT statement for the terminating thread.

Condition handling

When a new thread is created, no ON-units are assumed to be established. The ON-units which are in effect at the time a thread is created are not inherited by the new attached thread. Conditions that occur within a thread are handled within the thread and are not handled across thread boundaries.

For example, assume that thread **A** opens file **F**; then, **A** creates thread **T**. **T** then causes the ENDFILE condition to be raised. If an ON ENDFILE condition, is not established in thread **T** itself, the ERROR condition is raised in **T** and the usual condition handling takes place all within thread **T**. Whether or not **A** has established ON-units for ENDFILE or ERROR does not affect the execution of thread **T**.

A thread must establish ON-units for appropriate conditions if it wishes to handle them. There is no mechanism to signal conditions across threads.

If CTRL-BREAK is used to raise the ATTENTION condition, the ATTENTION condition is raised only in the main thread, not in any threads created by ATTACH statements.

Task data and attribute

Task variables hold thread related information, such as thread identification, service category, and dispatching priority. A variable is given the TASK attribute by explicit declaration, or implicitly by appearing in a THREAD option.

```
►►──TASK────────────────────────────────────────────────────►◄
```

A task variable is associated with a thread by the task reference in the THREAD option of the ATTACH statement creating the thread. A task variable is active if it is associated with a thread that is active. A task variable must be allocated before it is associated with a thread and must not be freed while it is active. An active task variable cannot be associated with another thread.

THREADID built-in function

THREADID (short for thread identifier) returns a FIXED BINARY(31,0) value that is the operating system thread identifier for an attached thread.

The value used by this built-in function can be used as a parameter to system functions such as DosSetPriority, but it should not be used as a parameter to DosKillThread.

```
►►──THREADID──────────────────────────────────►◄
              └─(──x──)─┘
```
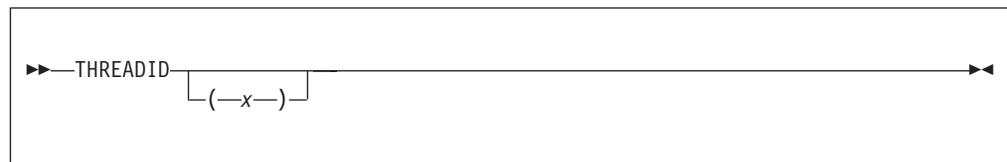
x Task reference. *x* must be a reference for a thread which is currently attached.

Sharing data between threads

All static and controlled data is shared between threads. All other data can also be shared via arguments/parameters and via based references, as long as the data is allocated and is not freed until all of the threads have finished using the data. For example, if automatic variables in the attaching thread are shared with the attached thread, the attaching block must not terminate until the attached thread has finished using the shared variables.

Serialization of data is the responsibility of the user. If new generations of controlled data are allocated or if existing generations are freed, it is possible to have certain threads still accessing an older generation or a generation that no longer exists. This can lead to unpredictable results.

All allocated storage, unless freed explicitly, is not freed until program termination.

PL/I does not serialize either ALLOCATEs or FREEs in AREA variables.

Sharing files between threads

All files are shared between threads. If a thread opens a file, it is not closed until it is explicitly closed or the program terminates.

Serialization is the responsibility of the user. Refer to "Sharing data between threads."

The message file and the display statement are automatically serialized by PL/I.

Chapter 19. Built-in functions, pseudovariables, and subroutines.

Built-in functions, pseudovariables, and subroutines

Built-in functions, pseudovariables, and subroutines

A large number of common tasks are available in the form of built-in functions, subroutines, and pseudovariables. When you use them, you can write less code more quickly with greater reliability.

The built-in functions, subroutines, and pseudovariables are listed in alphabetic order in this chapter. In general, each description has the following format:
*   A heading showing the syntax of the reference
*   A description of the value returned or, for a pseudovariable, the value set
*   A description of any arguments
*   Any other qualifications on using the function or pseudovariable

The abbreviations for built-in functions have separate declarations (explicit or contextual) and name scopes. In the following example:

```
dcl (Dim, Dimension) builtin;
```

is not a multiple declaration, and

```
dcl Binary file;
X = Bin (var, 6,3);
```

is valid even though *Bin* is an abbreviation of the *Binary* built-in function.

Declaring and invoking built-in functions, pseudovariables, and built-in subroutines

Built-in functions, pseudovariables, and subroutines can be contextually or explicitly declared.

BUILTIN attribute

The BUILTIN attribute specifies that the name is a built-in function, pseudovariable, or a subroutine.

```
►►──BUILTIN─────────────────────────────────────────────────────────►◄
```
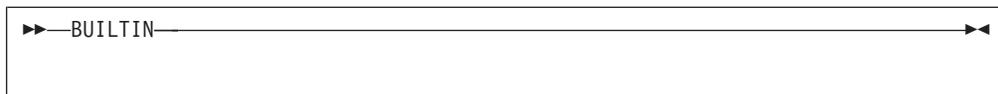
Built-in names can be used as programmer-defined names. BUILTIN can be declared for a built-in name in any block that has inherited, from a containing block, a programmer-defined declaration or use of the same name. The following example shows built-in names with the BUILTIN attribute.

Example 1

```
.1.   A: procedure;
         declare Sqrt float binary;
.2.      X = Sqrt;

.3.      B:  Begin;
            Declare Sqrt builtin;
            Z = Sqrt(P);
          end B;

        end A;
```

In this example:

.1.   Sqrt is a programmer-defined name.

.2.   The assignment to the variable X is a reference to the programmer-defined name Sqrt.

.3.   Sqrt is declared with the BUILTIN attribute so that any reference to Sqrt within B is recognized as a reference to the built-in function and not to the programmer-defined name Sqrt declared in **1**.

Invoking built-in functions and pseudovariables

The following syntax is used to invoke built-in functions and pseudovariables.

```
►►──name───────────────────────────────────────►◄
           ┌────────,───────┐
           │   ┌──▼───────┐ │
        └─(─┴──┴─argument─┴─┴─)─┘
```
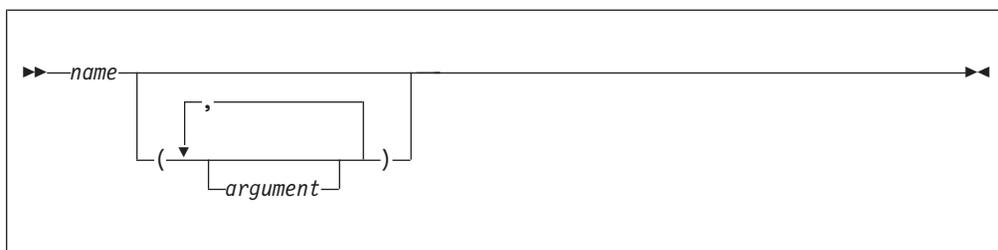
Invoking built-in subroutines

The following syntax is used to invoke built-in subroutines.

```
►►──CALL──name─────────────────────────────;────────────────►◄
              └─(─────────────────)─┘
                  ┌──────,──────┐
                  ▼─argument─┘
```
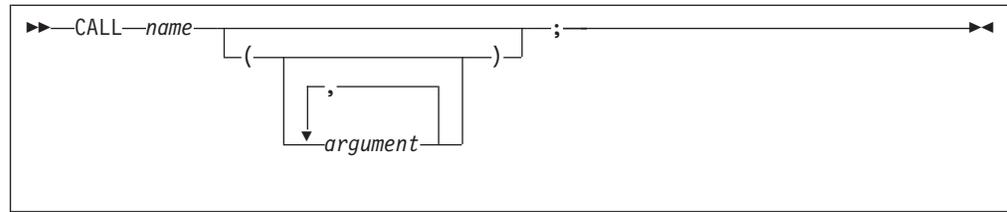
Specifying arguments for built-in functions, pseudovariables, and built-in subroutines

Arguments, which can be expressions, are evaluated and converted to a data type suitable for the built-in function according to the rules for data conversion.

Aggregate arguments

All built-in functions and pseudovariables that can have arguments can have array arguments (if more than one is an array, the bounds must be identical).

- ADDR, ALLOCATION, CURRENTSIZE, SIZE, STRING, and the array-handling functions return an element value.
- Under the compiler option USAGE(UNSPEC(ANS)), UNSPEC returns an element value; Under USAGE(UNSPEC(IBM)) returns an array of values.
- All other functions return an array of values.

Specifying an array argument is equivalent to placing the function reference or pseudovariable in a do-group where one or more arguments is a subscripted array reference that is modified by the control variable.

For example:

```
dcl A(2) char(2) varying;
dcl B(2) char(2)
    init('AB','CD');
dcl C(2) fixed bin
    init(1,2);
A=substr(B,1,C);
```

results in `A(1)` having the value `A` and `A(2)` having the value `CD`.

The built-in functions and pseudovariables that can accept structure or union arguments are ADDR, ALLOCATION, CURRENTSIZE, SIZE, STRING, and UNSPEC. UNSPEC may be applied to a structure or union only if the compiler option USAGE(UNSPEC((ANS)) is in effect.

Null and optional arguments

Some built-ins do not require arguments. You must either explicitly declare these with the BUILTIN attribute or contextually declare them by including a null argument list in the reference—for example, ONCHAR(). Otherwise, the name is not recognized as a built-in.

Accuracy of mathematical functions

The accuracy of a result is influenced by two factors:
- The accuracy of the argument
- The accuracy of the algorithm

Accuracy of mathematical functions

Most arguments contain errors. An error in a given argument can accumulate over several steps before the evaluation of a function. Even data fresh from input conversion can contain errors. The effect of argument error on the accuracy of a result depends entirely on the nature of the mathematical function, and not on the algorithm that computes the result. This book does not discuss argument errors of this type.

The mathematical built-in functions that are implemented using inline machine instructions produce results of different accuracy.

Categories of built-in functions

The following sections list built-in functions, subroutines, and pseudovariables.

Only full function names are listed in these tables. Existing abbreviations are provided in the sections that describe each built-in function, subroutine, and pseudovariable.

Arithmetic built-in functions

The arithmetic built-in functions allow you to do the following:

- Determine properties of arithmetic values. For example, the SIGN function indicates the sign of an arithmetic variable.
- Perform routine arithmetic operations.

Table 47 lists the arithmetic built-in functions and a short description of each.

Some of the arithmetic functions derive the data type of their results from one or more arguments. When the data types of the arguments differ, they are converted as described in Chapter 5, "Data conversion," on page 73.

Table 47. Arithmetic built-in functions

Function	Description
ABS	Calculates the absolute value of a value
CEIL	Calculates the smallest integer value greater than or equal to a value
COMPLEX	Returns the complex number with given real and imaginary parts
CONJG	Returns the complex conjugate of a value
FLOOR	Calculates the largest integer value less than or equal to a value
IMAG	Returns the imaginary part of a complex number
MAX	Calculates the maximum of 2 or more values
MIN	Calculates the minimum of 2 or more values
MOD	Returns the modular equivalent of the remainder of one value divided by another
RANDOM	Returns a pseudo-random value
REAL	Returns the real part of a complex number
REM	Calculates the remainder of one value divided by another
ROUND	Rounds a value at a specified digit
SIGN	Returns a -1, 0 or 1 if a value is negative, zero, or positive, respectively
TRUNC	Calculates the nearest integer for value rounded towards zero

Array-handling built-in functions

The array-handling built-in functions operate on array arguments and return an element value. Any conversion of arguments required for these functions is noted in the function description. Table 48 on page 375 lists the array-handling built-in functions.

Table 48. Array-handling built-in functions

Function	Description
ALL	Calculates the bitwise "and" of all the elements of an array
ANY	Calculates the bitwise "or" of all the elements of an array
DIMENSION	Returns the number of elements in a dimension of an array
HBOUND	Returns the upper bound for a dimension of an array
LBOUND	Returns the lower bound for a dimension of an array
POLY	Returns floating-point approximate of two arrays
PROD	Calculates the product of all the elements of an array
SUM	Calculates the sum of all the elements of an array

Buffer-management built-in functions

The buffer-management built-in functions operate on a ″buffer″, which is an area of storage specified by an address and a number of bytes. The PLIFILL, PLIMOVE and PLIOVER built-in subroutines are also useful in managing buffers. Table 49 lists the buffer-management built-in functions.

Table 49. Buffer-management built-in functions

Function	Description
COMPARE	Compares two buffers
HEXIMAGE	Returns a character string that is the hex representation of a buffer
MEMINDEX	Finds the location of one string or buffer within a buffer
MEMSEARCH	Searches for the first occurrence of any one of the elements of a string within a buffer
MEMSEARCHR	Searches for the first occurrence of any one of the elements of a string within a buffer, but the search starts from the right
MEMVERIFY	Searches for the first nonoccurrence of any one of the elements of a string within a buffer
MEMVERIFYR	Searches for the first occurrence of any one of the elements of a string within a buffer, but the search starts from the right
XMLCHAR	Writes XML corresponding to a structure to a buffer

Condition-handling built-in functions

The condition-handling built-in functions enable you to determine the cause of a condition that has occurred.

Use of these functions is valid only within the scope of an ON-unit or dynamic descendant for:
* the condition specific to the built-in function
* the ERROR or FINISH condition when raised as an implicit action

All other uses are out of context.

Table 50. Condition-handling built-in functions

Function	Description
DATAFIELD	Returns the value of a string that raised the NAME condition
ONCHAR	Returns the value of a character that caused a conversion condition
ONCODE	Returns the condition code value
ONCONDCOND	Returns the name of CONDITION condition being processed
ONCONDID	Returns a number which identifies a particular condition
ONCOUNT	Returns the number of outstanding conditions
ONFILE	Returns the name of a file for which a condition is raised
ONGSOURCE	Returns the value of a graphic string that caused a conversion condition
ONKEY	Returns the key of a record that raised a condition
ONLOC	Returns the name of the procedure in which a condition occurred
ONSOURCE	Returns the value of a string that caused a conversion condition
ONSUBCODE	Returns an integer value that gives additional information about certain I/O errors
ONWCHAR	Returns the value of a widechar that caused a conversion condition
ONWSOURCE	Returns the value of a widechar string that caused a conversion condition

Date/time built-in functions

These built-in functions return or manipulate date and time information in terms of days, seconds, and character date/time stamps. Some of these built-in functions allow you to specify the date/time patterns to be used. Table 51 on page 377 lists the supported date/time built-in functions. Table 52 on page 378 lists the supported date/time patterns.

The time zone and accuracy for these functions are system dependent.

Lilian format: The Lilian format, named in honor of Luigi Lilio, the creator of the Gregorian calendar, represents a date as the number of days or seconds from the beginning of the Gregorian calendar. This format is useful for performing calculations involving elapsed time.

The Lilian format counts days that have elapsed since October 14, 1582; day one is Friday, October 15, 1582. For example, 16 May 1988 is 148138 Lilian days. The valid range of Lilian days is 1 to 3,074,324 (15 October 1582 to 31 December 9999).

For the number of elapsed seconds, the Lilian format counts elapsed seconds starting at 00:00:00 14 October 1582. For example, 00:00:01 on 15 October 1582 is 86,401 (24*60*60+1) Lilian seconds, and 19:01:01 16 May 1988 is 12,799,191,661 Lilian seconds. The valid range of Lilian seconds is 86,400 to 265,621,679,999.999 (23:59:59:999 31 December 9999) seconds.

Table 51. Date/time built-in functions

Function	Description
DATE	Returns the current date in the pattern YYMMDD
DATETIME	Returns the current date and time in the user-specified pattern or in the default pattern YYYYMMDDHHMISS999
DAYS	Returns the number of days corresponding to a date/time pattern string, or the number of days for today's date
DAYSTODATE	Converts a number of days to a date/time pattern string
DAYSTOSECS	Converts a number of days to a number of seconds
REPATTERN	Takes a value holding a date in one pattern and returns that value converted to a date in a second pattern
SECS	Returns the number of seconds corresponding to a date/time pattern string, or the number of seconds for today's date
SECSTODATE	Converts a number of seconds to a date/time pattern string
SECSTODAYS	Converts a number of seconds to a number of days
TIME	Returns the current time in the pattern HHMISS999
VALIDDATE	Indicates if a string holds a valid date
WEEKDAY	Returns the day of the week corresponding to the current day or specified DAYS value
Y4DATE	Takes a date value with the pattern 'YYMMDD' and returns the date value with the two-digit year widened to a four-digit year
Y4JULIAN	Takes a date value with the pattern 'YYDDD' and returns the date value with the two-digit year widened to a four-digit year
Y4YEAR	Takes a date value with the pattern 'YY' and returns the date value with the two-digit year widened to a four-digit year

Table 52 on page 378 uses the following formats:

YYYY Four-digit year
YY Two-digit year
MM Two-digit month
MMM Three-letter month (Ex: DEC)
Mmm Three-letter month (Ex: Dec)
DD Two-digit day within a given month
DDD Number of days within a given year
HH Number of hours within a given day
MI Number of minutes within a given hour
SS Number of seconds within a given minute
999 Number of milliseconds within a given second

Note: For the three-letter month patterns, the uppercase/lowercase characters must correspond exactly.

The only supported pattern using any of HH, MI, SS or 999 is the pattern 'YYYYMMDDHHMISS999'.

Table 52. Date/time patterns

	Four-digit years	**Two-digit years**
Year first	YYYYMMDDHHMISS999 YYYYMMDD YYYYMMMDD YYYYMmmDD YYYYDDD YYYYMM YYYYMMM YYYYMmm YYYY	YYMMDD YYMMMDD YYMmmDD YYDDD YYMM YYMMM YYMmm YY
Month first	MMDDYYYY MMMDDYYYY MmmDDYYYY MMYYYY MMMYYYY MmmYYYY	MMDDYY MMMDDYY MmmDDYY MMYY MMMYY MmmYY
Day first	DDMMYYYY DDMMMYYYY DDMmmYYYY DDDYYYY	DDMMYY DDMMMYY DDMmmYY DDDYY

When the day is omitted from a pattern, it is assumed to have the value 1. If the month and day are both omitted, they are also assumed to have the value 1.

When using MMM, the date must be written in three uppercase letters; when using Mmm, the date must be written with the first letter in uppercase, and the letters following in lowercase.

Floating-point inquiry built-in functions

The floating-point inquiry built-in functions return information about the floating-point variable arguments that you specify.

Table 53. Floating-point inquiry built-in functions

Function	Description
EPSILON	Returns the spacing around 1
HUGE	Returns the largest positive finite value that a floating-point variable can hold
MAXEXP	Returns the maximum value for an exponent
MINEXP	Returns the minimum value for an exponent
PLACES	Returns the model precision for a floating point value
RADIX	Returns the model base for a floating point value
TINY	Returns the smallest positive value that a floating-point variable can hold

Floating-point manipulation built-in functions

The floating-point manipulation built-in functions perform mathematical operations on floating-point variables that you specify and return the result of the operation.

Table 54. Floating-point manipulation built-in functions

Function	Description
EXPONENT	Returns the exponent part of a floating point value
PRED	Returns the next representable value before a floating-point value
SCALE	Multiplies a floating-point number by an integral power of the radix
SUCC	Returns the next representable value after a floating-point value

Input/output built-in functions

The input and output built-in functions allow you to determine the current state of a file.

Table 55. Input/output built-in functions

Function	Description
COUNT	Returns the number of data items transmitted during the last GET or PUT
ENDFILE	Indicates if a file is open and end-of-file has been reached for it
FILEID	Returns a system token value for a PL/I file constant or variable
FILEOPEN	Indicates if a file is open
FILEREAD	Reads from a file
FILESEEK	Changes the current file position to a new location
FILETELL	Returns a value indicating the current position of a file
FILEWRITE	Writes to a file
LINENO	Returns the current line number associated with a print file
PAGENO	Returns the current page number associated with a print file
SAMEKEY	Indicates if a record is followed by another with the same key

Integer manipulation built-in functions

The integer manipulation built-in functions perform operations on integer variables and return the result of the operation.

Table 56. Integer manipulation built-in functions

Function	Description
IAND	Calculates the bitwise "and" of 2 or more fixed binary values
IEOR	Calculates the bitwise "exclusive-or" of 2 fixed binary values
INOT	Calculates the bitwise "not" of a fixed binary value
IOR	Calculates the bitwise "or" of 2 or more fixed binary values
ISIGNED	Casts an integer to a signed integer without changing its bit pattern
ISLL	Shifts a fixed binary value "logically" to the left
ISRL	Shifts a fixed binary value "logically" to the right
IUNSIGNED	Casts an integer to an unsigned integer without changing its bit pattern
LOWER2	Divides a fixed binary value by an integral power of 2
RAISE2	Multiplies a fixed binary value by an integral power of 2

Mathematical built-in functions

All of these functions operate on floating-point values to produce a floating-point result. Any argument that is not floating-point is converted. The accuracy of these functions is discussed in "Accuracy of mathematical functions" on page 373. Table 57 lists the mathematical built-in functions.

Table 57. Mathematical built-in functions

Function	Description
ACOS	Calculates the arc cosine
ACOSF	Calculates ACOS inline if hardware architecture permits
ASIN	Calculates the arc sine
ASINF	Calculates ASIN inline if hardware architecture permits
ATAN	Calculates the arc tangent
ATAND	Calculates the arc tangent in degrees
ATANF	Calculates ATAN inline if hardware architecture permits
ATANH	Calculates the hyperbolic arc tangent
COS	Calculates the cosine
COSD	Calculates the cosine for a value in degrees
COSF	Calculates COS inline if hardware architecture permits
COSH	Calculates the hyperbolic cosine
ERF	Calculates the error function
ERFC	Calculates the complement of the error function
EXP	Calculates e to a power
EXPF	Calculates EXP inline if hardware architecture permits
GAMMA	Calculates the gamma function
LOG	Calculates the natural logarithm
LOGF	Calculates LOG inline if hardware architecture permits
LOG10	Calculates the base 10 logarithm
LOG10F	Calculates LOG10 inline if hardware architecture permits
LOG2	Calculates the base 2 logarithm
LOGGAMMA	Calculates the log of the gamma function
SIN	Calculates the sine
SIND	Calculates the sine for a value in degrees
SINF	Calculates SIN inline if hardware architecture permits
SINH	Calculates the hyperbolic sine
SQRT	Calculates the square root
SQRTF	Calculates SQRT inline if hardware architecture permits
TAN	Calculates the tangent
TAND	Calculates the tangent for a value in degrees
TANF	Calculates TAN inline if hardware architecture permits
TANH	Calculates the hyperbolic tangent

Miscellaneous built-in functions

The built-in functions do not fit into any of the previous categories are those listed in Table 58.

Table 58. Miscellaneous built-in functions

Function	Description
BYTE	Synonym for CHARVAL
CDS	Returns a FIXED BINARY(31) value that indicates if the old and current values in a *compare double and swap* were equal.
CHARVAL	Returns the character value corresponding to an integer
COLLATE	Returns a character(256) string specifying the collating order
CS	Returns a FIXED BINARY(31) value that indicates if the old and current values in a *compare and swap* were equal.
GETENV	Returns a value representing a specified environment variable
HEX	Returns a character string that is the hex representation of a value
ISMAIN	Indicates if the current procedure is main
OMITTED	Indicates if a parameter was not supplied on a call
PACKAGENAME	Returns the name of the containing package
PLIRETV	Returns the PL/I return code value
PRESENT	Indicates if a parameter was supplied on a call
PROCEDURENAME	Returns the name of the most closely nested procedure
PUTENV	Adds new environment variables or modifies the values of existing environment variables
RANK	Returns the integer value corresponding to a character or widechar
SOURCEFILE	Returns the name of the containing file
SOURCELINE	Returns the number of the containing line
STRING	Returns a string that is the concatenation of all the elements of a string aggregate
SYSTEM	Returns the value returned by a command
UNSPEC	Returns a bit string that is the internal representation of a value
VALID	Indicates if the contents of a variable are valid for its declaration
WCHARVAL	Returns the widechar value corresponding to an integer.

Ordinal-handling built-in functions

The ordinal-handling built-in functions return information about a specified ordinal.

Table 59. Ordinal-handling built-in functions

Function	Description
ORDINALNAME	Returns a character string giving an ordinal's name
ORDINALPRED	Returns the next lower value for an ordinal
ORDINALSUCC	Returns the next higher value for an ordinal

Precision-handling built-in functions

The precision-handling built-in functions allow you to manipulate variables with specified precisions, and they return the value resulting from the operation.

Precision-handling built-in functions

Table 60. Precision-handling built-in functions

Function	Description
ADD	Adds, with a specified precision, two values
BINARY	Converts a value to binary
DECIMAL	Converts a value to decimal
DIVIDE	Divides, with a specified precision, two values
FIXED	Converts a value to fixed
FLOAT	Converts a value to float
MULTIPLY	Multiplies, with a specified precision, two values
PRECISION	Converts a value to specified precision
SIGNED	Converts a value to signed fixed binary
SUBTRACT	Subtracts, with a specified precision, two values
UNSIGNED	Converts a value to unsigned fixed binary

Pseudovariables

Pseudovariables represent receiving fields. They cannot be nested. For example, the following is invalid:

```
unspec(substr(A,1,2)) = '00'B;
```

A pseudovariable can appear only:

- on the left side of an assignment statement
- as the target in a DO-specification and then only if it is one of SUBSTR, REAL, IMAG or UNSPEC
- in the data list of a GET statement or in the STRING option of a PUT statement
- as the string name in a KEYTO or REPLY option

The pseudovariables are:

Table 61. Built-in pseudovariables

Function	Description
ENTRYADDR	Sets an entry variable with the address of the entry to be invoked
IMAG	Assigns the imaginary part of a complex number
ONCHAR	Sets the value of a character that caused a conversion condition
ONGSOURCE	Sets the value of a graphic string that caused a conversion condition
ONSOURCE	Sets the value of a string that caused a conversion condition
REAL	Assigns the real part of a complex number
STRING	Assigns a string that is the concatenation of all the elements of a string aggregate
SUBSTR	Assigns a substring of a string
ONWCHAR	Sets the value of a widechar that caused a conversion condition
ONWSOURCE	Sets the value of a widechar string that caused a conversion condition
TYPE	Assigns a typed structure or union to storage located by a handle
UNSPEC	Assigns a bit string that is the internal representation of a value

Storage control built-in functions

The storage control built-in functions allow you to determine the storage requirements and location of variables, to assign special values to area and locator variables, to perform conversion between offset and pointer values, to obtain the number of generations of a controlled variable, and to reference data and methods of objects and classes. Table 62 lists the storage control built-in functions.

Table 62. Storage control built-in functions

Function	Description
ADDR	Returns the address of a variable
ADDRDATA	Returns the address of the first data byte of a string when applied to a varying string
ALLOCATE	Allocates storage of the specified size in the heap
ALLOCATION	Returns the current number of generations of a controlled variable
ALLOCSIZE	Returns a FIXED BIN(N,0) value giving the amount of storage allocated with a specific pointer
AUTOMATIC	Allocates storage of the specified size in the stack
AVAILABLEAREA	Returns the size of the largest single allocation that can be made in an area
BINARYVALUE	Converts a pointer, offset, or ordinal to an integer
BITLOCATION	Returns the bit offset of a variable within a byte
CHECKSTG	Returns a bit(1) value determining whether allocated storage is uncorrupted
CURRENTSIZE	Returns the current size of a variable
CURRENTSTORAGE	Synonym for CURRENTSIZE
EMPTY	Returns an "empty" area
ENTRYADDR	Returns the address of the routine associated with an entry
HANDLE	Returns a handle to a typed structure or union
LOCATION	Returns the byte offset of a variable within a structure
NULL	Returns a null pointer value
OFFSET	Converts a pointer to an offset
OFFSETADD	Adds an integer to an offset
OFFSETDIFF	Subtracts two offsets
OFFSETSUBTRACT	Subtracts an integer from an offset
OFFSETVALUE	Converts an integer to an offset
POINTER	Converts an offset to a pointer
POINTERADD	Adds an integer to a pointer
POINTERDIFF	Subtracts two pointers
POINTERSUBTRACT	Subtracts an integer from a pointer
POINTERVALUE	Converts an integer or handle to a pointer
SIZE	Returns the maximum size of a variable
STORAGE	Synonym for SIZE
SYSNULL	Returns a system null pointer value
TYPE	Returns the typed structure or union located by a handle

Storage control

Function	Description
UNALLOCATED	Returns a bit(1) value indicating if a specified pointer value is the start of allocated storage
VARGLIST	Returns the address of the first optional parameter passed to a procedure
VARGSIZE	Returns the number of bytes occupied by a byvalue parameter

String-handling built-in functions

The string-handling built-in functions simplify the processing of bit, character, graphic and widechar strings. The string arguments can be represented by an arithmetic expression that will be converted to string either according to data conversion rules or according to the rules given in the function description.

Note: Some of these functions, such as LOWERCASE, TRANSLATE, TRIM and UPPERCASE, support only CHARACTER data.

Table 63. String-handling built-in functions

Function	Description
BIT	Converts a value to bit
BOOL	Performs Boolean operation on 2 bit strings
CENTERLEFT	Returns a string with a value centered (to the left) in it
CENTERRIGHT	Returns a string with a value centered (to the right) in it
CENTRELEFT	Synonym for CENTERLEFT
CENTRERIGHT	Synonym for CENTERRIGHT
CHARACTER	Converts a value to a character string
CHARGRAPHIC	Converts a GRAPHIC string to a mixed character string
COPY	Returns a string consisting of n copies of a string
EDIT	Returns a string consisting of a value converted to a given picture specification
GRAPHIC	Converts a value to graphic
HIGH	Returns a character string consisting of n copies of the highest character in the collating sequence
INDEX	Finds the location of one string within another
LEFT	Returns a string with a value left-justified in it
LENGTH	Returns the current length of a string
LOW	Returns a character string consisting of n copies of the lowest character in the collating sequence
LOWERCASE	Returns a character string with all the characters from A to Z converted to their corresponding lowercase character.
MAXLENGTH	Returns the maximum length of a string
MPSTR	Truncates a string at a logical boundary and returns a mixed character string
REPEAT	Returns a string consisting of n+1 copies of a string
REVERSE	Returns a reversed image of a string
RIGHT	Returns a string with a value right-justified in it

Table 63. String-handling built-in functions (continued)

Function	Description
SEARCH	Searches for the first occurrence of any one of the elements of a string within another string
SEARCHR	Searches for the first occurrence of any one of the elements of a string within another string but the search starts from the right
SUBSTR	Assigns a substring of a string
TALLY	Returns the number of times one string occurs in another
TRANSLATE	Translates a string based on two translation strings
TRIM	Trims specified characters from the left and right sides of a string
UPPERCASE	Returns a character string with all the characters from a to z converted to their corresponding uppercase character.
VERIFY	Searches for first nonoccurrence of any one of the elements of a string within another string
VERIFYR	Searches for first nonoccurrence of any one of the elements of a string within another string but the search starts from the right
WHIGH	Returns a widechar string consisting of n copies of the highest widechar ('ffff'wx).
WIDECHAR	Converts a value to a widechar string
WLOW	Returns a widechar string consisting of n copies of the lowest widechar ('0000'wx).

Subroutines

Built-in subroutines perform miscellaneous operations that do not necessarily return a result as built-in functions do.

Table 64. Built-in subroutines

Function	Description
PLIASCII	Converts from EBCDIC to ASCII
PLIDELETE	Frees the storage associated with a handle
PLICANC	Cancels the automatic restart facility (z/OS only)
PLICKPT	Takes a checkpoint for later restart (z/OS only)
PLIDUMP	Dumps information about currently open files, the calling path to the current location, etc.
PLIEBCDIC	Converts from ASCII to EBCDIC
PLIFILL	Fills n bytes at an address with a specified byte value
PLIFREE	Frees the storage associated with a pointer to heap storage
PLIMOVE	Moves n bytes from one address to another
PLIOVER	Moves n bytes from one address to another, compensating for possible overlap of the source and target
PLIREST	Restarts program execution (z/OS only)
PLIRETC	Sets the PL/I return code value
PLISAXA	Allows you to perform SAX-style parsing of an XML document residing in a buffer in your program
PLISAXB	Allows you to perform SAX-style parsing of an XML document residing in a file

Table 64. Built-in subroutines (continued)

Function	Description
PLISRTA	Allows the use of DFSORT to sort an input file to produce a sorted output file
PLISRTB	Allows the use of DFSORT to sort input records provided by an E15 PL/I exit procedure to produce a sorted output file
PLISRTC	Allows the use of DFSORT to sort an input file to produce sorted records that are processed by an E35 PL/I exit procedure
PLISRTD	Allows the use of DFSORT to sort input records provided by an E15 PL/I exit procedure to produce sorted records that are processed by an E35 PL/I exit procedure

ABS

ABS returns the absolute value of *x*. It is the positive value of *x*.

```
►►──ABS(x)──────────────────────────────────────────────►◄
```

x   Expression.

The mode of the result is REAL. The result has the base, scale, and precision of *x*, except when x is COMPLEX FIXED(p,q). In the latter case, the result is REAL FIXED(min(n,p+1),q) where n is N for DECIMAL and M for BINARY.

ACOS

ACOS returns a real floating-point value that is an approximation of the inverse (arc) cosine in radians of *x*.

```
►►──ACOS(x)─────────────────────────────────────────────►◄
```

x   Real expression, where ABS(x) <= 1.

The result is in the range:

$$0 \leq ACOS(x) \leq \pi$$

and has the base and precision of *x*.

ACOSF

ACOSF is exactly like ACOS except that:
- ACOSF calculates its result inline if hardware architecture permits.
- Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "ACOS."

ADD

ADD returns the sum of *x* and *y* with a precision specified by *p* and *q*. The base, scale, and mode of the result are determined by the rules for expression evaluation.

```
►►──ADD(x,y,p─────────)──────────────────────────────────►◄
               └─,q─┘
```
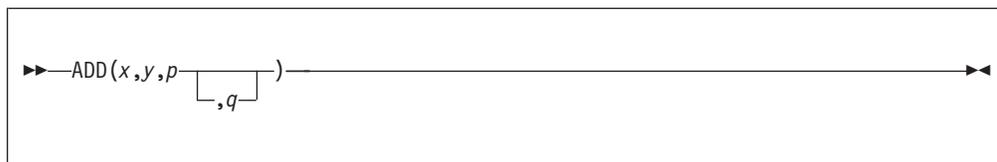
x and y
Expressions.

p Restricted expression. It specifies the number of digits to be maintained throughout the operation.

q Restricted expression specifying the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is the default. For a floating-point result, *q* must be omitted.

ADD can be used for subtraction by prefixing a minus sign to the operand to be subtracted.

ADDR

ADDR returns the pointer value that identifies the generation of *x*.

```
►►──ADDR(x)──────────────────────────────────────────────►◄
```
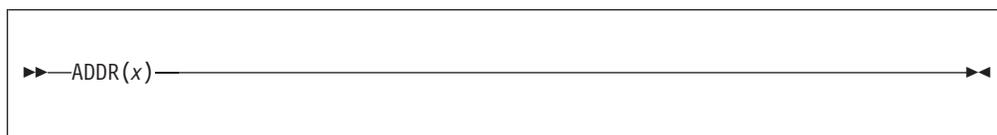
x Reference. It refers to a variable of any data type, data organization, alignment, and storage class except:

- A subscripted reference to a variable that is an unaligned fixed-length bit string
- A reference to a DEFINED or BASED variable or simple parameter, which is an unaligned fixed-length bit string
- A minor structure or union whose first base element is an unaligned fixed-length bit string (except where it is also the first element of the containing major structure or union)
- A major structure or union that has the DEFINED attribute or is a parameter, and that has an unaligned fixed-length bit string as its first element
- A reference that is not to connected storage

If *x* is a reference to:
- An aggregate parameter, it must have the CONNECTED attribute
- An aggregate, the returned value identifies the first element
- A component or cross section of an aggregate, the returned value takes into account subscripting and structure or union qualification
- A varying string, the returned value identifies the 2-byte prefix
- An area, the returned value identifies the control information

- A controlled variable that is not allocated in the current program, the null pointer value is returned
- A based variable, the result is the value of the pointer explicitly qualifying *x* (if it appears), or associated with *x* in its declaration (if it exists), or a null pointer
- A parameter, and a dummy argument has been created, the returned value identifies the dummy argument

ADDRDATA

ADDRDATA returns the pointer value that identifies the generation of *x*.

```
►►──ADDRDATA(x)──────────────────────────────────►◄
```
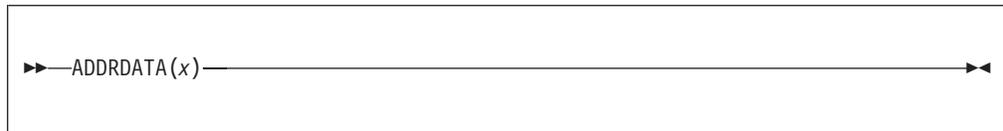
x   Reference.

ADDRDATA behaves the same as the ADDR built-in function except in the following instance:

- When applied to a varying string, ADDRDATA returns the address of the first data byte of the string (rather than of the length field).

ALL

ALL returns a bit string in which each bit is 1 if the corresponding bit in each element of *x* exists and is 1. The length of the result is equal to that of the longest element.
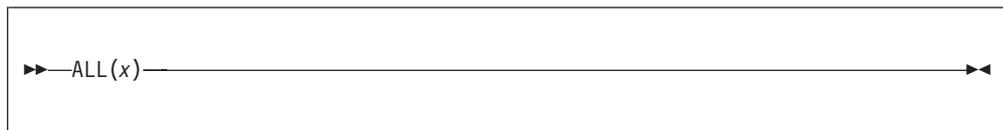
```
►►──ALL(x)───────────────────────────────────────►◄
```

x   Computational array expression. If *x* is not a bit string array, then *x* is converted to a bit string.

ALLOCATE

ALLOCATE allocates storage of size *n* in heap storage and returns the pointer to the allocated storage.

```
►►──ALLOCATE(n)──────────────────────────────────►◄
```
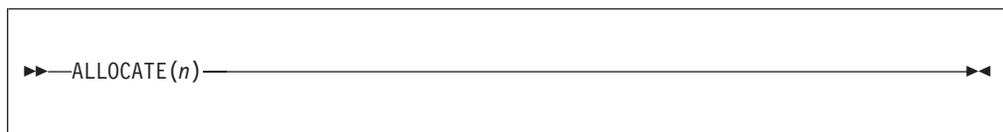
Abbreviation: ALLOC

n   Expression. *n* must be nonnegative. If necessary, *n* is converted to REAL FIXED BINARY(31,0).

If the requested amount of storage is not available, the STORAGE condition is raised.

ALLOCATION

ALLOCATION returns a FIXED BINARY(31,0) specifying the number of generations that can be accessed in the current program for *x*.

```
►►──ALLOCATION(x)──────────────────────────────────────────────────►◄
```
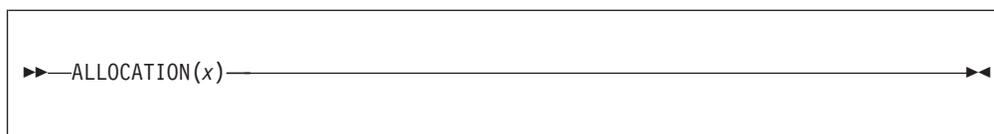
Abbreviation: ALLOCN

x   Level-1 unsubscripted controlled variable.

If *x* is not allocated in the current program, the result is zero.

ALLOCSIZE

ALLOCSIZE returns a FIXED BIN(31,0) value giving the amount of storage allocated with a specified pointer. To use this built-in function, you must also specify the CHECK(STORAGE) compile-time option.

```
►►──ALLOCSIZE(p)───────────────────────────────────────────────────►◄
```
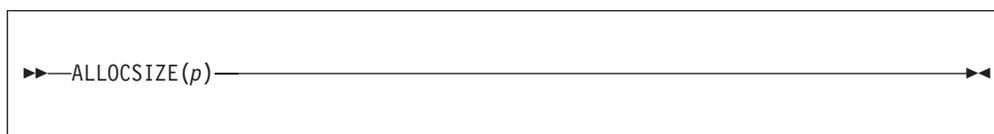
p   Pointer expression.

ALLOCSIZE returns 0 if the pointer does not point to the start of a piece of allocated storage.

ANY

ANY returns a bit string in which each bit is 1 if the corresponding bit in any element of *x* exists and is 1. The length of the result is equal to that of the longest element.
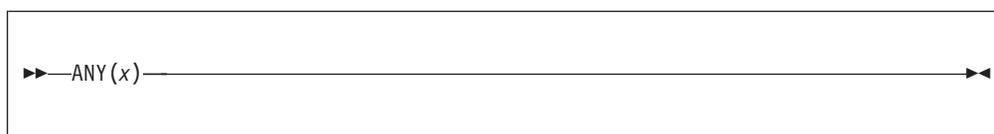
```
►►──ANY(x)─────────────────────────────────────────────────────────►◄
```

x   Computational array expression. If *x* is not a bit string array, then *x* is converted to a bit string.

ASIN

ASIN returns a real floating-point value that is an approximation of the inverse (arc) sine in radians of *x*.

```
►►──ASIN(x)──────────────────────────────────────►◄
```

x   Real expression, where ABS(x) <= 1.

The result is in the range:

$-\pi/2 \leq \text{ASIN}(x) \leq \pi/2$

and has the base and precision of *x*.

ASINF

ASINF is exactly like ASIN except that:
• ASINF calculates its result inline if hardware architecture permits.
• Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
• The accuracy of the result is set by the hardware.

For the definition and syntax, see "ASIN."

ATAN

ATAN returns a floating-point value that is an approximation of the inverse (arc) tangent in radians of *x* or of a ratio *x/y*.

```
►►──ATAN(x──────┬──)──────────────────────────────►◄
               └─,y─┘
```
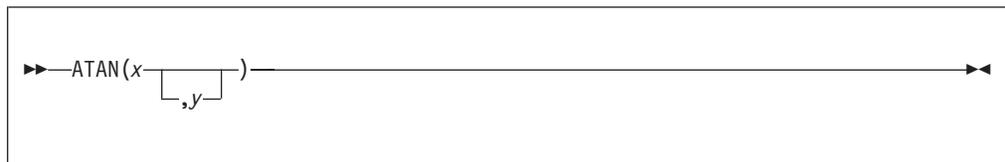
x and y
Expressions.

If *x* alone is specified, the result has the base and precision of *x*, and is in the range:

$-\pi/2 < \text{ATAN}(x) < \pi/2$

If *x* and *y* are specified, each must be real. An error exists if *x* and *y* are both zero. The result for all other values of *x* and *y* has the precision of the longer argument, a base determined by the rules for expressions, and a value given by:

ATAN(x/y)	for y>0
$\pi/2$	for y=0 and x>0
$-\pi/2$	for y=0 and x<0
π + ATAN(x/y)	for y<0 and x>=0
$-\pi$ + ATAN(x/y)	for y<0 and x<0

ATAND

ATAND returns a real floating-point value that is an approximation of the inverse (arc) tangent in degrees of *x* or of a ratio *x/y*.

```
►►──ATAND(x──┬──────┬──)──────────────────────────────────────►◄
             └─,y──┘
```

x and y
    Expressions.

    If *x* alone is specified it must be real. The result has the base and precision of *x*, and is in the range:

```
-90 < ATAND(x) < 90
```

    If *x* and *y* are specified, each must be real. The value of the result is given by:

```
(180/π) * ATAN(x,y)
```

For argument requirements and attributes of the result see "ATAN" on page 390.

ATANF

ATANF is exactly like ATAN except that:
• ATANF calculates its result inline if hardware architecture permits.
• Only one real argument is allowed.
• Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
• The accuracy of the result is set by the hardware.

For the definition and syntax, see "ATAN" on page 390.

ATANH

ATANH returns a floating-point value that has the base, mode, and precision of *x*, and is an approximation of the inverse (arc) hyperbolic tangent of *x*.

```
►►──ATANH(x)──────────────────────────────────────────────────►◄
```

x   Expression. ABS(x)<1.

The result has a value given by:
```
LOG((1 + x)/(1 - x))/2
```

AUTOMATIC

AUTOMATIC allocates storage of size *n* automatic storage and returns the pointer to the allocated storage.

```
►►──AUTOMATIC(n)────────────────────────────────────────────────►◄
```

Abbreviation: AUTO

n Expression. *n* must be nonnegative. If necessary, *n* is converted to REAL FIXED BINARY(31,0).

The storage acquired cannot be explicitly freed; the storage is automatically freed when the block terminates.

AVAILABLEAREA

AVAILABLEAREA returns a FIXED BINARY(31,0) value. The value returned by AVAILABLEAREA is the size of the largest single allocation that can be obtained from the area *x*.

```
►►──AVAILABLEAREA(x)─────────────────────────────────────────────►◄
```
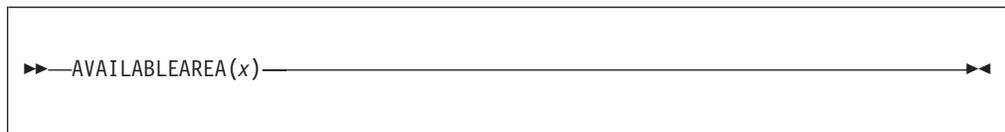
x A reference with the AREA attribute.

Example

```
dcl Uarea area(1000);
dcl Pz ptr;
dcl C99z char(99) varyingz based(Pz);
dcl (SizeBefore, SizeAfter) fixed bin(31);
SizeBefore = availablearea(Uarea);        /* returns 1000    */
Alloc C99z in(Uarea);
SizeAfter = availablearea(Uarea);         /* returns 896     */
dcl C9 char(896) based(Pz);
Alloc C9 in(Uarea);
```

BINARY

BINARY returns the binary value of *x*, with a precision specified by *p* and *q*. The result has the mode and scale of *x*.

```
►►──BINARY(x─┬──────────┬──)────────────────────────────────────►◄
             └─,p─┬─────┤
                  └─,q─┘
```
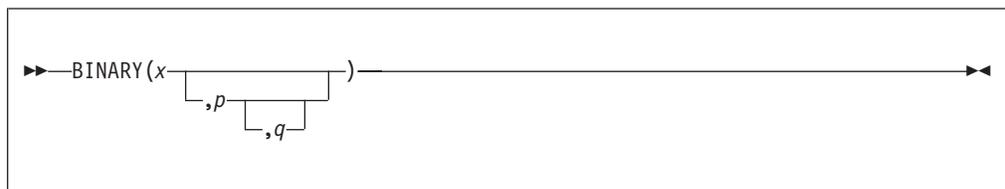
Abbreviation: BIN

x Expression.

p　Restricted expression. Specifies the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

q　Restricted expression. It specifies the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is the default. For a floating-point result, *q* must be omitted.

If both *p* and *q* are omitted, the precision of the result is determined from the rules for base conversion.

BINARYVALUE

BINARYVALUE returns a FIXED BINARY(31,0) value that is the converted value of *x*; *x* can be a pointer, offset, or ordinal.

```
►►──BINARYVALUE(x)────────────────────────────────────────►◄
```
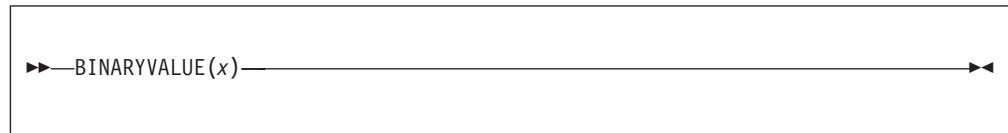
Abbreviation: BINVALUE

x　Expression.

BIT

BIT returns a result that is the bit value of *x*, and has a length specified by *y*.

```
►►──BIT(x─────────)──────────────────────────────────────►◄
         └─,y─┘
```
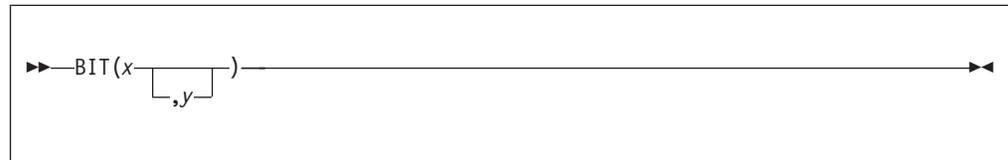
x　Expression.

y　Expression. If necessary, *y* is converted to a real fixed-point binary value. If *y* is omitted, the length is determined by the rules for type conversion. If $y = 0$, the result is the null bit string. *y* must not be negative.

BITLOCATION

BITLOCATION returns a FIXED BINARY(31,0) result that is the location of bit *x* within the byte that contains *x*. The value returned is always between 0 and 7 ($0 \leq$ value ≤ 7).

```
►►──BITLOCATION(x)──────────────────────────────────────►◄
```

Abbreviation: BITLOC

x　Reference of type unaligned bit. If *x* does not have type unaligned bit, a value of 0 is returned.

x must not be subscripted.

BITLOCATION can be used in restricted expressions, with the following limitations. If BITLOC(x) is used to set:
- The extent of a variable *y* that must have constant extents, or
- The value of a variable *y* that must have a constant value,

then *x* must be declared before *y*.

For examples, see "LOCATION" on page 428.

BOOL

BOOL returns a bit string that is the result of the Boolean operation *z*, on *x* and *y*. The length of the result is equal to that of the longer operand, *x* or *y*.

```
►►──BOOL(x,y,z)──────────────────────────────────►◄
```

x and y
> Expressions. *x* and *y* are converted to bit strings, if necessary. If *x* and *y* are of different lengths, the shorter is padded on the right with zeros to match the longer.

z
> Expression. *z* is converted to a bit string of length 4, if necessary. When a bit from *x* is matched with a bit from *y*, the corresponding bit of the result is specified by a selected bit of *z*, as follows:

x	y	Result
0	0	bit 1 of z
0	1	bit 2 of z
1	0	bit 3 of z
1	1	bit 4 of z

BYTE

BYTE is a synonym for CHARVAL. For more information, refer to "CHARVAL" on page 398.

CDS

CDS returns a FIXED BINARY(31) value that indicates if the old and current values in a *compare double and swap* were equal.

```
►►──CDS(p,q,x)──────────────────────────────────►◄
```
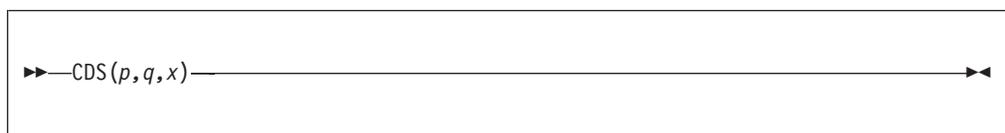
p
Address of the old FIXED BINARY(63) value.

q
Address of the current FIXED BINARY(63) value.

x
The new FIXED BINARY(63) value.

CDS compares the ″current″ and ″old″ values. If they are equal, the ″new″ value is copied over the ″current″, and a value of 0 is returned. If they are unequal, the ″current″ value is copied over the ″old″, and a value of 1 is returned.

On z/OS, the CDS built-in function implements the CDS instruction. For a detailed description of this function, read the appendices in the *Principles of Operations* manual.

On Intel, the CDS built-in function uses the Intel cmpxchg8 instruction in the same manner that the CS built-in function uses the cmpxchg4 instruction.

CEIL

CEIL determines the smallest integer value greater than or equal to *x*, and assigns this value to the result.

```
►►─CEIL(x)────────────────────────►◄
```
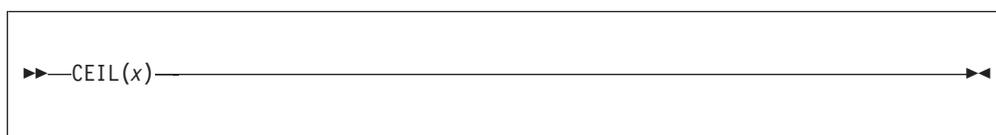
x Real expression.

The result has the mode, base, scale, and precision of *x*, except when *x* is fixed-point with precision (p,q). The precision of the result is then given by:

$$(min(N,max(p-q+1,1)),0)$$

where N is the maximum number of digits allowed.

CENTERLEFT

CENTERLEFT returns a string that is the result of inserting string *x* in the center (or one position to the left of center) of a string with length y and padded on the left and on the right with the character *z* as needed. Specifying a value for *z* is optional.

```
►►─┬─CENTERLEFT─┬─(x,y─┬────┬─)──────────►◄
   └─CENTRELEFT─┘      └─,z─┘
```
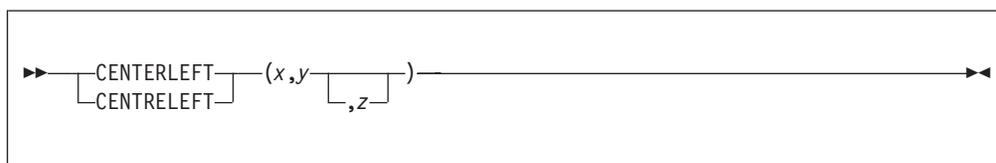
Abbreviation: CENTER

x Expression that is converted to character.

y Expression that is converted to FIXED BINARY(31,0).

z Optional expression. If specified, *z* must be CHARACTER(1) NONVARYING type.

Example

```
dcl Source char value('Feel the Power');
dcl Target20 char(20);
dcl Target21 char(21);

Target20 = center (Source, length(Target20), '*');
          /* '***Feel the Power***' - exactly centered */
```

```
Target21 = center (Source, length(Target21), '*');
          /* '***Feel the Power****' - leaning left! */
```

If *z* is omitted, a blank is used as the padding character.

CENTRELEFT

Abbreviation: CENTRE

CENTRELEFT is a synonym for CENTERLEFT.

CENTERRIGHT

CENTERRIGHT returns a string that is the result of inserting string *x* in the center (or one position to the right of center) of a string with length *y* and padded on the left and on the right with the character *z* as needed. Specifying a value for *z* is optional.

```
►►──┬─CENTERRIGHT─┬──(x,y─┬────┬──)───────────────────────►◄
    └─CENTRERIGHT─┘       └─,z─┘
```
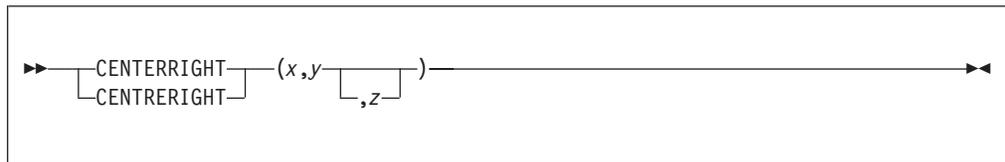
x   Expression that is converted to character.

y   Expression that is converted to FIXED BINARY(31,0).

z   Optional expression. If specified, *z* must be CHARACTER(1) NONVARYING type.

Example

```
dcl Source char value('Feel the Power');
dcl Target20 char(20);
dcl Target21 char(21);

Target20 = centerright (Source, length(Target20), '*');
          /* '***Feel the Power***' - exactly centered */

Target21 = centerright (Source, length(Target21), '*');
          /* '****Feel the Power***' - leaning right! */
```

If *z* is omitted, a blank is used as the padding character.

CENTRERIGHT

CENTRERIGHT is a synonym for CENTERRIGHT.

CHARACTER

CHARACTER returns the character value of *x*, with a length specified by *y*. CHARACTER also supports conversion from graphic to character type.

```
►►──CHARACTER(x──────────)───────────────────────────►◄
                 └─,y─┘
```
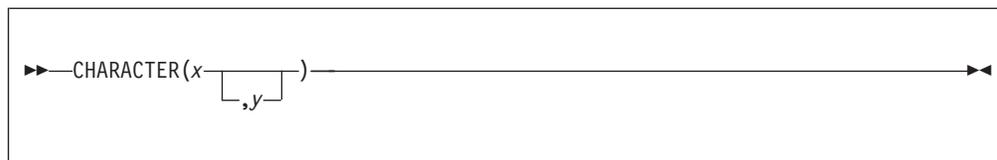
Abbreviation: CHAR

x   Expression.

  x must have a computational type.

  When *x* is nongraphic, CHARACTER returns *x* converted to character.

  When *x* is GRAPHIC, CHARACTER returns *x* converted to SBCS characters. If a DBCS character cannot be translated to an SBCS equivalent, the CONVERSION condition is raised.

  The values of *x* are not checked.

y   Expression. If necessary, y is converted to a real fixed-point binary value.

  If *y* is omitted, the length is determined by the rules for type conversion.

  y cannot be negative.

  If *y* = 0, the result is the null character string.

Example

Conversion from graphic to character:

```
dcl X graphic(6);
dcl A char (6);
A = char(X);
```

For X with value	Intermediate Result	A is assigned
.A.B.C.D.E.F	ABCDEF	ABCDEF

CHARGRAPHIC

CHARGRAPHIC converts a GRAPHIC (DBCS) string *x* to a mixed character string with a length specified by *y*.

```
►►──CHARGRAPHIC(x──────────)─────────────────────────►◄
                    └─,y─┘
```
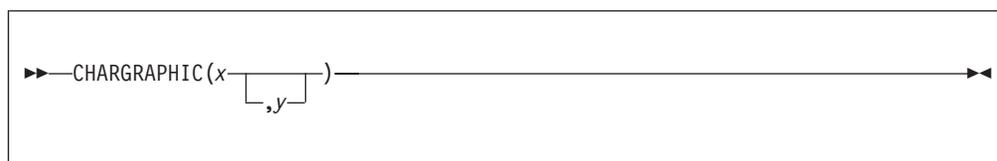
Abbreviation: CHARG

x   Expression.

  x must be a GRAPHIC string. CHARACTER returns *x* converted to a mixed character string.

y Expression. If necessary, y is converted to a real fixed-point binary value.

If *y* is omitted, the length is determined by the rules for type conversion.

y cannot be negative.

If *y* = 0, the result is the null character string.

The following rules apply:
- If *y* = 1, the result is a character string of 1 blank.
- If *y* is greater than the length needed to contain the character string, the result is padded with SBCS blanks.
- If *y* is less than the length needed to contain the character string, the result is truncated. The integrity is preserved by truncating after a graphic, and appending an SBCS blank if necessary, to complete the length of the string.

Example 1

Conversion from graphic to character, where *y* is long enough to contain the result:

```
dcl X graphic(6);
dcl A char (12);
A = char(X,12);
```

For X with value	Intermediate Result	A is assigned
.A.B.C.D.E.F	.A.B.C.D.E.F	.A.B.C.D.E.F

Example 2

Conversion from graphic to character, where *y* is too short to contain the result:

```
dcl X graphic(6);
dcl A char (12);
A = char(X,11);
```

For X with value	Intermediate Result	A is assigned
.A.B.C.D.E.F	.A.B.C.D.E.F	.A.B.C.D.Eb

CHARVAL

CHARVAL returns the CHARACTER(1) value corresponding to an integer.

```
►►──CHARVAL──(──n──)────────────────────►◄
```
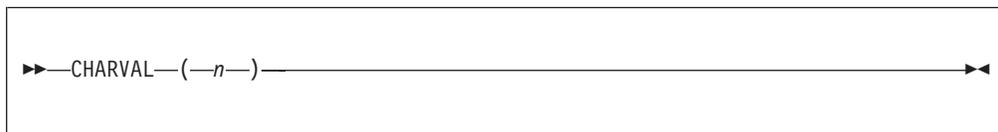
n Expression converted to UNSIGNED FIXED BIN(8) if necessary.

CHARVAL(n) has the same bit value as n (that is, UNSPEC(CHARVAL(n)) is equal to UNSPEC(n)), but it has the attributes CHARACTER(1).

CHARVAL is the inverse of RANK (when applied to character).

CHECKSTG

CHECKSTG returns a bit(1) value which indicates whether a specified pointer value is the start of a piece of uncorrupted allocated storage. If no pointer value is supplied, CHECKSTG determines whether all allocated storage is uncorrupted. To use this built-in function, you must also specify the CHECK(STORAGE) compile-time option.

```
►►──CHECKSTG(──┬────┬──)──────────────────────────────────────►◄
               └─p──┘
```
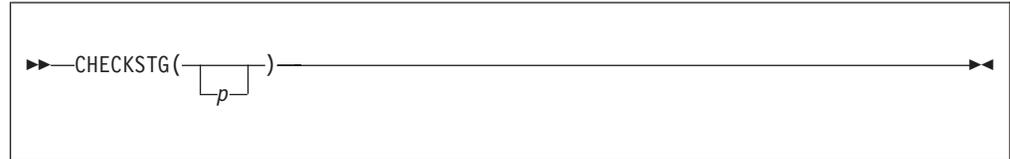
p Pointer expression.

When an allocation is made, it is followed by eight extra bytes which are set to 'ff'x. The allocation is considered *uncorrupted* if those bytes have not been altered.

The pointer expression must point to storage allocated for a BASED variable.

COLLATE

COLLATE returns a CHARACTER(256) string comprising the 256 possible CHARACTER(1) values one time each in the collating order.

```
►►──COLLATE────┬──────┬──────────────────────────────────────►◄
               └─()───┘
```
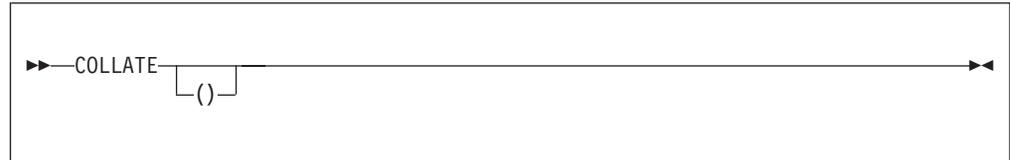
COMPARE

COMPARE returns a FIXED BINARY(31,0) value that is:
* Zero, if the z bytes at the addresses x and y are identical
* Negative, if the z bytes at x are less than those at y
* Positive, if the z bytes at x are greater than those at y

```
►►──COMPARE(x,y,z)────────────────────────────────────────────►◄
```
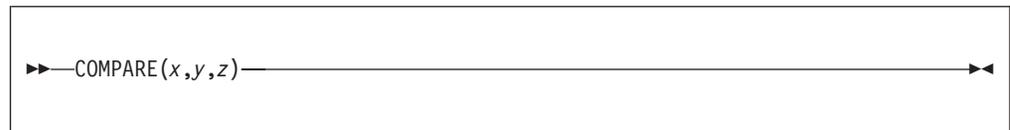
x and y
   Expressions. Both must have the POINTER or OFFSET type. If OFFSET, the expression must be declared with the AREA qualification.

z Expression that is converted to FIXED BINARY(31,0).

Example

```
dcl Result fixed bin;
dcl 1 Str1,
     2 B fixed bin(31),
     2 C pointer,
     2 * union,
```

```
             3 D char(4),
             3 E fixed bin(31),
             3 *,
               4 * char(3),
               4 F fixed bin(8) unsigned,
          2 * char(0);
    dcl 1 Template nonasgn static,
          2 * fixed bin(31) init(16),     /* ''X */
          2 * pointer init(null()),
          2 * char(4) init(''),
          2 * char(0);

    call plimove(addr(Str1), addr(Template), stg(Str1));
    Result = compare(addr(Str1), addr(Template), stg(Str1));   /*  0 */
    D = 'DSA ';
    Result = compare(addr(Str1), addr(Template), stg(Str1));   /*  1 */
    B = 15;      /* '00000F00'X */
    D = 'DSA ';
    Result = compare(addr(Str1), addr(Template), stg(Str1));   /* -1 */
```

COMPLEX

COMPLEX returns the complex value $x + y$I.

```
►►──COMPLEX(x,y)──────────────────────────────────►◄
```
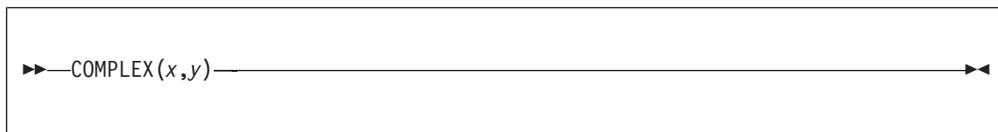
Abbreviation: CPLX

x and y
   Real expressions.

   If x and y differ in base, the decimal argument is converted to binary. If they
   differ in scale, the fixed-point argument is converted to floating-point. The result
   has the common base and scale.

If fixed-point, the precision of the result is given by the following:

```
(min(N,max(p1-q1,p2-q2)+max(q1,q2)),max(q1,q2))
```

In this example, (p1,q1) and (p2,q2) are the precisions of x and y, respectively, and
N is the maximum number of digits allowed.

After any necessary conversions have been performed, if the arguments are
floating-point, the result has the precision of the longer argument.

CONJG

CONJG returns the conjugate of x, that is, the value of the expression with the sign
of the imaginary part reversed.

```
►►──CONJG(x)──────────────────────────────────────►◄
```
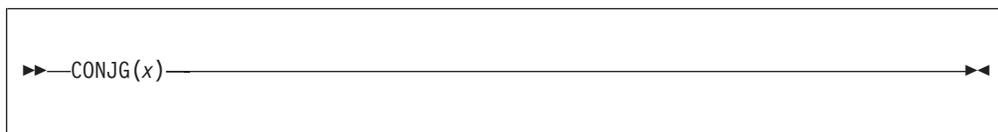
x   Expression.

If *x* is real, it is converted to complex. The result has the base, scale, mode, and precision of *x*.

COPY

COPY returns a string consisting of *y* concatenated copies of the string *x*.
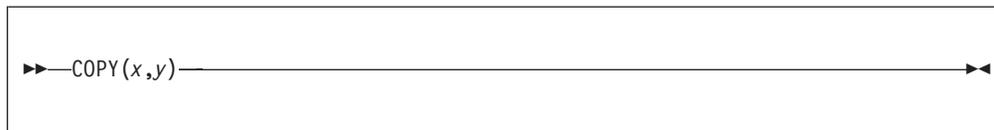
```
►►──COPY(x,y)──────────────────────────────────────────────────►◄
```

x Expression.

 x must have a computational type and should have a string type. If not, it is converted to character.

y An integer expression with a nonnegative value. It specifies the number of repetitions. It must have a computational type and is converted to FIXED BINARY(31,0).

 If *y* is zero, the result is a null string.

Considering the following code:

```
copy('Walla ',1)        /* returns 'Walla ' */

repeat('Walla ',1)      /* returns 'Walla  Walla ' */
```

In the preceding example, `repeat(x,n)` is equivalent to `copy(x,n+1)`.

COS

COS returns a floating-point value that has the base, precision, and mode of *x*, and is an approximation of the cosine of *x*.

```
►►──COS(x)──────────────────────────────────────────────────────►◄
```
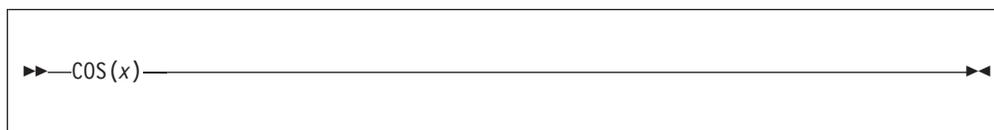
x Expression with a value in radians.

COSD

COSD returns a real floating-point value that has the base and precision of *x*, and is an approximation of the cosine of *x*.

```
►►──COSD(x)─────────────────────────────────────────────────────►◄
```

x Real expression with a value in degrees.

COSF

COSF is exactly like COS except that:
- COSF calculates its result inline if hardware architecture permits.
- The argument must be a real expression.
- The maximum supported absolute value of the argument is set by the hardware.
- Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "COS" on page 401.

COSH

COSH returns a floating-point value that has the base, precision, and mode of x, and is an approximation of the hyperbolic cosine of x.

```
►►──COSH(x)──────────────────────────────────────────────►◄
```
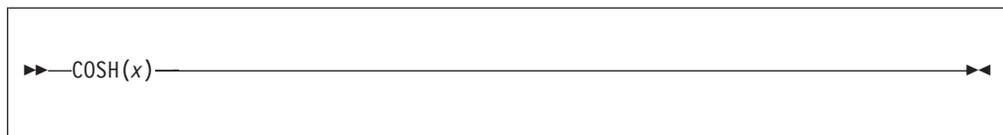
x   Expression.

COUNT

COUNT returns an unscaled REAL FIXED BINARY value specifying the number of data items transmitted during the last GET or PUT operation on x.

```
►►──COUNT(x)─────────────────────────────────────────────►◄
```
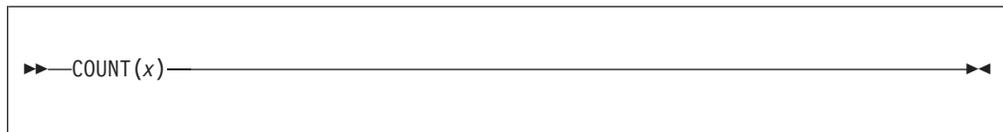
x   File-reference. The file must be open and have the STREAM attribute.

The count of transmitted items for a GET or PUT operation on x is initialized to zero before the first data item is transmitted, and is incremented by one after the transmission of each data item in the list. If x is not open in the current program, a value of zero is returned.

If an ON-unit or procedure is entered during a GET or PUT operation, and within that ON-unit or procedure, a GET or PUT operation is executed for x, the value of COUNT is reset for the new operation. It is restored when the original GET or PUT is continued.

The BIFPREC compiler option determines the precision of the result returned.

CS returns a FIXED BINARY(31) value that indicates if the old and current values in a *compare and swap* were equal.

```
▶▶──CS(p,q,x)─────────────────────────────────────────────────────▶◀
```
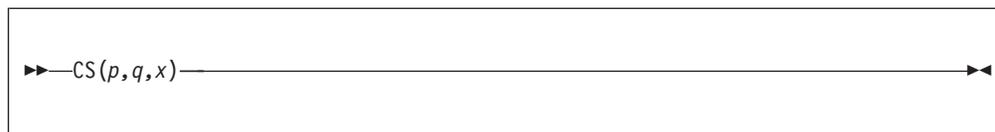
p Address of the old FIXED BINARY(31) value.

q Address of the current FIXED BINARY(31) value.

x The new FIXED BINARY(31) value.

CS compares the ″current″ and ″old″ values. If they are equal, the ″new″ value is copied over the ″current″, and a value of 0 is returned. If they are unequal, the ″current″ value is copied over the ″old″, and a value of 1 is returned.

So, CS could be implemented as the following PL/I function, but then it would not be atomic at all. :

```
cs: proc( old_Addr, current_Addr, new )
 returns( fixed bin(31) byvalue )
 options( byvalue );

 dcl old_Addr     pointer;
 dcl current_Addr pointer;
 dcl new          fixed bin(31);

 dcl old          fixed bin(31) based(old_addr);
 dcl current      fixed bin(31) based(current_addr);

 if current = old then
   do;
     current = new;
     return( 0 );
   end;
 else
   do;
     old = current;
     return( 1 );
   end;
end;
```

On z/OS, the CS built-in function implements the CS instruction. For a detailed description of this function, read the appendices in the *Principles of Operations* manual.

On Intel, the CDS built-in function uses the Intel cmpxchg4 instruction. The cmpxchg4 instruction takes the address of a ″current″ value, a ″new″ value and an ″old″ value. It returns the original ″current″ value and updates it with the ″new″ value only if it equaled the ″old″ value.

So, on Intel, the CS built-in function is implemented via the following inline function:

```
cs: proc( old_Addr, current_Addr, new )
 returns( fixed bin(31) byvalue )
 options( byvalue );

 dcl old_Addr     pointer;
 dcl current_Addr pointer;
```

```
      dcl new          fixed bin(31);

      dcl old          fixed bin(31) based(old_addr);
      dcl current      fixed bin(31) based(current_addr);

      if cmpxchg4( current_Addr, new, old ) = old then
        do;
          return( 0 );
        end;
      else
        do;
          old = current;
          return( 1 );
        end;
    end;
```

CURRENTSIZE

CURRENTSIZE returns a FIXED BINARY(31,0) value giving the
implementation-defined storage, in bytes, required by *x*.

```
►►──CURRENTSIZE(x)──────────────────────────────────────────►◄
```
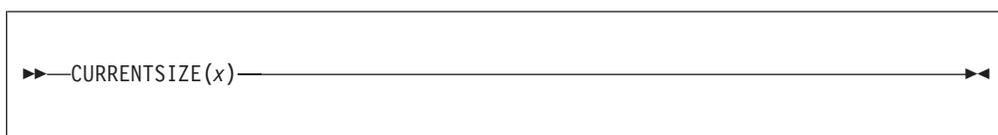
x   A variable of any data type, data organization, and storage class except:

- A BASED, DEFINED, parameter, subscripted, or structure or union base-element variable that is an unaligned fixed-length bit string
- A minor structure or union whose first or last base element is an unaligned fixed-length bit string (except where it is also the first or last element of the containing major structure or union)
- A major structure or union that has the BASED, DEFINED, or parameter attribute, and which has an unaligned fixed-length bit string as its first or last element
- A variable not in connected storage

The value returned by CURRENTSIZE(x) is defined as the number of bytes that would be transmitted in the following circumstances:

```
  declare F file record output
          environment(scalarvarying);
  write file(F) from(S);
```

If *x* is a scalar varying-length string, the returned value includes the length-prefix of the string and the number of currently-used bytes. It does not include any unused bytes in the string.

If *x* is a scalar area, the returned value includes the area control bytes and the current extent of the area. It does not include any unused bytes at the end of the area.

If *x* is an aggregate containing areas or varying-length strings, the returned value includes the area control bytes, the maximum sizes of the areas, the length prefixes of the strings, and the number of bytes in the maximum lengths of the strings. The exception to this rule is:

If *x* is a structure or union whose last element is a nondimensioned area, the returned value includes that area's control bytes and the current extent of that area. It does not include any unused bytes at the end of that area.

The CURRENTSIZE built-in function must not be used on a BASED variable with adjustable extents if that variable has not been allocated.

For examples of the CURRENTSIZE built-in function, refer to the "SIZE" on page 473.

CURRENTSTORAGE

Abbreviation: CSTG

CURRENTSTORAGE is a synonym for CURRENTSIZE. For more information, refer to "CURRENTSIZE" on page 404.

DATAFIELD

DATAFIELD is in context in a NAME condition ON-unit (or any of its dynamic descendants). It returns a character string whose value is the contents of the field that raised the condition. It is also in context in an ON-unit (or any of its dynamic descendants) for an ERROR or FINISH condition raised as part of the implicit action for the NAME condition.

```
►►──DATAFIELD──────────────────────────────────────────────►◄
             └─()─┘
```
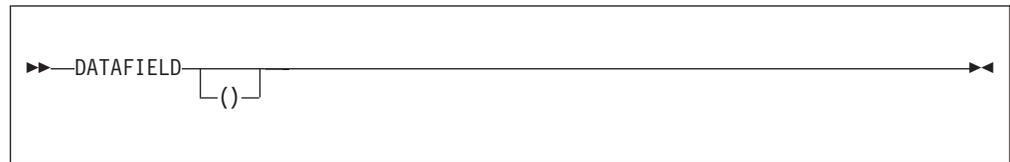
If the string that raised the condition contains DBCS identifiers, GRAPHIC data, or mixed character data, DATAFIELD returns a mixed character string.

If DATAFIELD is used out of context, a null string is returned.

DATE

DATE returns a nonvarying character(6) string containing the date in the format, YYMMDD.

```
►►──DATE──────────────────────────────────────────────►◄
        └─()─┘
```
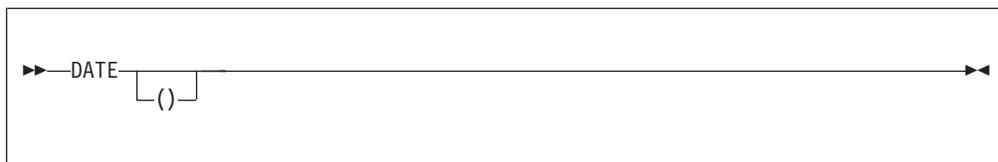
DATETIME

DATETIME returns a character string timestamp of today's date in either the default or a user-specified format.

```
►►──DATETIME──────────────────────────────────────────►◄
          └─(──┬────┬──)─┘
               └─y─┘
```
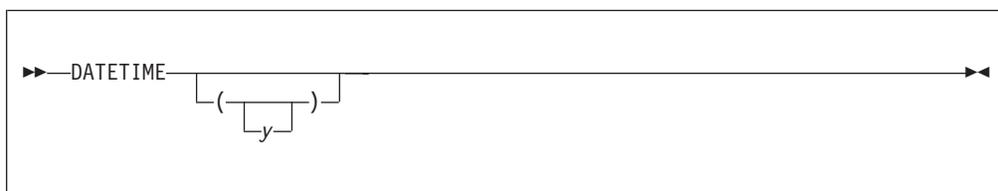
y Expression

If present, it specifies the date/time pattern in which the date is returned. If *y* is missing, it is assumed to be the default date/time pattern 'YYYYMMDDHHMISS999'.

Refer to Table 52 on page 378 for the allowed patterns.

y must have computational type and should have character type. If not, it is converted to character.

See "DAYS" for an example of using DATETIME.

DAYS

DAYS returns a FIXED BINARY(31,0) value which is the number of days (in Lilian format) corresponding to the date *d*.

```
►►──DAYS──────────────────────────────────────────────►◄
        └─(──┬───────────────┬──)─┘
             └─d─┘ └─,p─┬──────┘
                        └─,w─┘
```
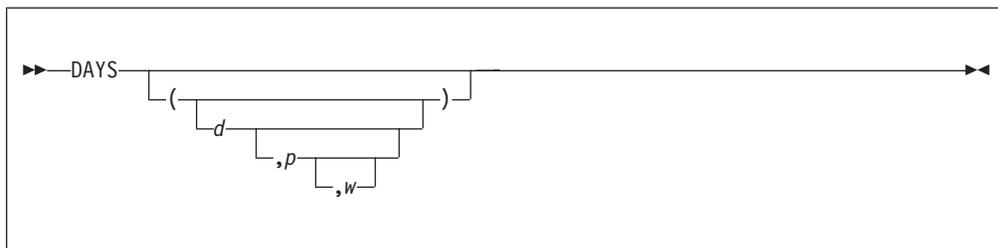
d String expression representing a date. If omitted, it is assumed to be the value returned by DATETIME().

The value for *d* must have computational type and should have character type. If not, *d* is converted to character.

p One of the supported date/time patterns. If omitted, it is assumed to be the value 'YYYYMMDDHHMISS9999'.

p must have computational type and should have character type. If not, it is converted to character.

w An integer expression that defines a century window to be used to handle any two-digit year formats.

- If the value is positive, such as 1950, it is treated as a year.
- If negative or zero, the value specifies an offset to be subtracted from the current, system-supplied year.
- If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

Example

```
dcl Date_format value ('MMDDYYYY') char;
dcl Todays_date char(length(Date_format));
dcl Sep2_1993 char(length(Date_format));
dcl Days_of_July4_1993 fixed bin(31);
dcl Msg char(100) varying;
dcl Date_due char(length(Date_format));

Todays_date = datetime(date_format);          /* e.g. 06161993  */

Days_of_July4_1993 = days('07041993','MMDDYYYY');
Sep2_1993 = daystodate(days_of_July4_1993 + 60, Date_format);
             /*  09021993    */

Date_due = daystodate(days() + 60, Date_format);
       /* assuming today is July 4, 1993, this would be Sept. 2, 1993  */

Msg = 'Please pay amount due on or before ' ||
        substr(Date_due, 1, 2) || '/' ||
        substr(Date_due, 3,2)  || '/' ||
        substr(Date_due, 5);
```

The allowed patterns are listed in Table 52 on page 378. For an explanation of Lilian format, see "Date/time built-in functions" on page 376.

DAYSTODATE

DAYSTODATE returns a nonvarying character string containing the date in the form *p* that corresponds to *d* days (in Lilian format).
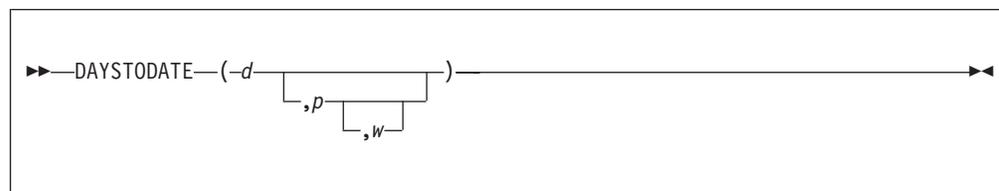
d The number of days (in Lilian format). *d* must have a computational type and is converted to FIXED BINARY(31,0) if necessary.

p One of the supported date/time patterns.

If omitted, *p* is assumed to be the default date/time pattern 'YYYYMMDDHHMISS999' (same as the default format returned by DATETIME).

w An integer expression that defines a century window to be used to handle any two-digit year formats.

- If the value is positive, such as 1950, it is treated as a year.

- If negative or zero, the value specifies an offset to be subtracted from the current, system-supplied year.
- If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The allowed patterns are listed in Table 52 on page 378. For an explanation of Lilian format, see "Date/time built-in functions" on page 376.

See "DAYS" on page 406 for an example using DAYSTODATE.

DAYSTOSECS

DAYSTOSECS returns a FLOAT BINARY(53) value that is the number of seconds corresponding to the number of days *x*.
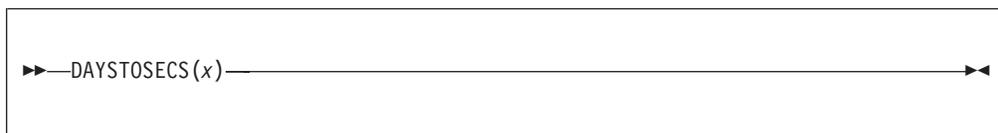
```
►►──DAYSTOSECS(x)──────────────────────────────────────►◄
```

x   Expression.

     x must have a computational type and is converted to FIXED BINARY(31,0) if necessary.

DAYSTOSECS(x) is the same as *x**(24*60*60).

DECIMAL

DECIMAL returns the decimal value of *x*, with a precision specified by *p* and *q*. The result has the mode and scale of *x*.

```
►►──DECIMAL(x─────────────)──────────────────────────►◄
            └─,p──────┘
                └─,q─┘
```
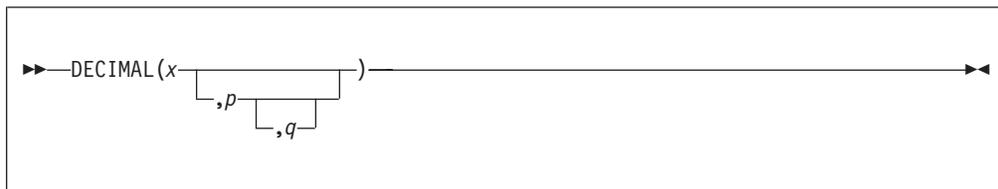
Abbreviation: DEC

x   Reference.

p   Restricted expression specifying the number of digits to be maintained throughout the operation.

q   Restricted expression specifying the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is assumed. For a floating-point result, *q* must be omitted.

If both *p* and *q* are omitted, the precision of the result is determined from the rules for base conversion.

DIMENSION

DIMENSION returns a FIXED BINARY(31,0) value specifying the current extent of dimension *y* of *x*.
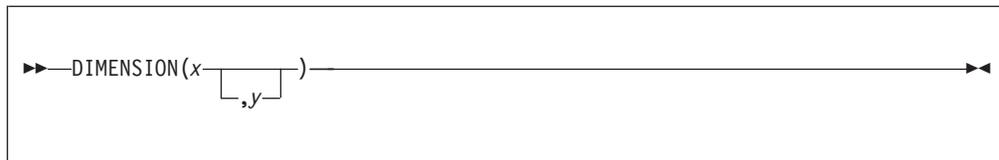
```
►►──DIMENSION(x───────)──────────────────────────►◄
              └─,y─┘
```

Abbreviation: DIM

x   Array reference. *x* must not have less than *y* dimensions.

y   Expression specifying a particular dimension of *x*. If necessary, *y* is converted to a FIXED BINARY(31,0). *y* must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

   y can be omitted only if the array is one-dimensional.

If *y* exceeds the number of dimensions of *x*, the DIMENSION function returns an undefined value.

Using LBOUND and HBOUND instead of DIMENSION is recommended.

DIVIDE

DIVIDE returns the quotient of *x/y* with a precision specified by *p* and *q*. The base, scale, and mode of the result follow the rules for expression evaluation.

```
►►──DIVIDE(x,y,p───────)──────────────────────────►◄
               └─,q─┘
```
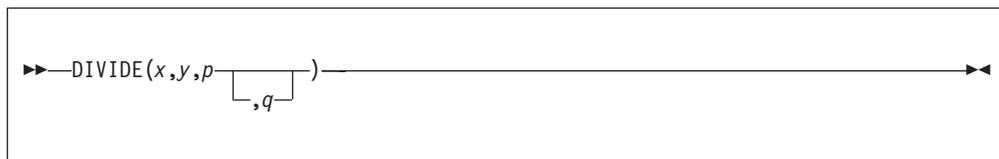
x   Expression.

y   Expression. If *y* = 0, the ZERODIVIDE condition is raised.

p   Restricted expression specifying the number of digits to be maintained throughout the operation.

q   Restricted expression specifying the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is the default. For a floating-point result, *q* must be omitted.

EDIT

EDIT returns a character string of length LENGTH(y). Its value is equivalent to what would result if *x* were assigned to a variable declared with the picture specification given by *y*.

For the valid picture characters, refer to Chapter 15, "Picture specification characters," on page 315.

```
►►──EDIT(x,y)──────────────────────────────────────────────────────►◄
```

x   Expression

   x must have computational type.

y   String expression.

   y must have character type and must contain picture characters that are valid for a PICTURE data item. If *y* does not contain a valid picture specification, the ERROR condition is raised.

Example

```
dcl pic1 char(9) init ('ZZZZZZZZ9');
dcl pic2 char(5) init ('ZZ9V.99');
dcl num fixed dec (9) init (123456789);
z = edit (num, pic1);                      /* '123456789'      */
z = edit (num, pic2);                      /*    '789.00'      */
z = edit (num, substr(pic1,8));            /*        '89'      */
z = edit (num, substr(pic2,1,4));          /*      '789.'      */
z = edit (num, substr(pic1,7,3));          /*       '789'      */
z = edit (num, substr(pic2,3,4));          /*      '9.00'      */
z = edit ('1', substr(pic1,7,3));          /*      '  1'       */
z = edit ('PL/I', 'AAXA');                 /*      'PL/I'       */
z = edit ('PL/I', 'AAAA');                 /* raises conversion */
```

If *x* cannot be edited into the picture specification given by *y*, the conditions raised are those that would be raised if *x* were assigned to a PICTURE data item which has the same picture specification contained in *y*.

EMPTY

EMPTY returns an area of zero extent. It can be used to free all allocations in an area.

```
►►──EMPTY──────────────────────────────────────────────────────────►◄
          └─()─┘
```

The value of this function is assigned to an area variable when the variable is allocated. Consider this example:

```
declare A area,
        I based (P),
        J based (Q);

allocate I in(A), J in (A);
A = empty();

/* Equivalent to:  free I in (A), J in (A); */
```

ENDFILE

ENDFILE returns a '1'B when the end of the file is reached; '0'B if the end is not reached. If the file is not open, the ERROR condition is raised.

```
►►──ENDFILE(x)────────────────────────────────────────────────►◄
```
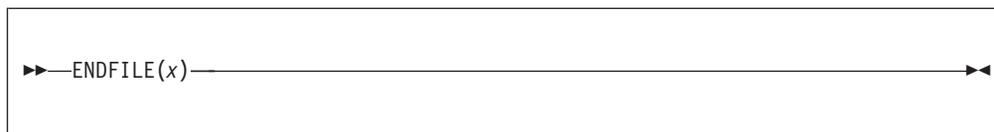
x File reference.

ENDFILE can be used to detect the end-of-file condition for bytestream files; for example, files that require the use of the FILEREAD built-in function.

ENTRYADDR

ENTRYADDR returns a pointer value that is the address of the first executed instruction if the entry *x* is invoked. The entry *x* must represent a non-nested procedure.

```
►►──ENTRYADDR(x)──────────────────────────────────────────────►◄
```
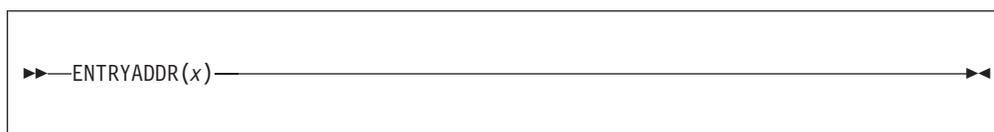
x Entry reference.

If *x* is a fetchable entry constant, it must be fetched before ENTRYADDR is executed.

ENTRYADDR pseudovariable

The ENTRYADDR pseudovariable initializes an entry variable, *x*, with the address of the entry to be invoked.

```
►►──ENTRYADDR(x)──────────────────────────────────────────────►◄
```
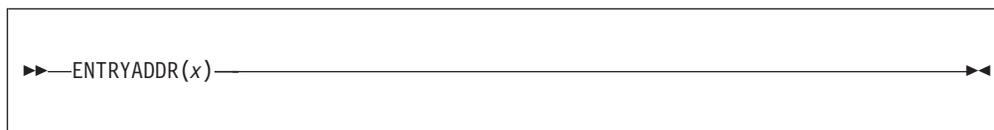
x Entry reference.

Note: If the address supplied to the ENTRYADDR variable is the address of an internal procedure, the results are unpredictable.

EPSILON

EPSILON returns a floating-point value that is the spacing between *x* and the next positive number when *x* is 1. It has the base, mode, and precision of *x*.

```
►►──EPSILON(x)──────────────────────────────────────────────►◄
```
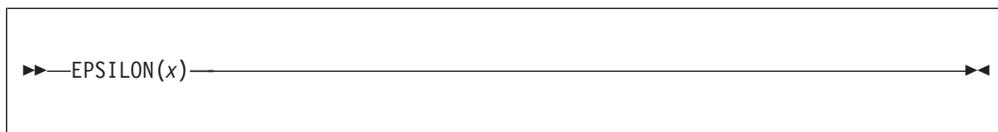
x   REAL FLOAT expression.

EPSILON(x) is a constant and can be used in restricted expressions.

ERF

ERF returns a real floating-point value that is an approximation of the error function of *x*.

```
►►──ERF(x)───────────────────────────────────────────────────►◄
```

x   Real expression.

The result has the base and precision of *x*, and a value given by:
$(2/ \sqrt{(\pi)}) \int_{0}^{x} EXP(-(t^2))dt$

ERFC

ERFC returns a real floating-point value that is an approximation of the complement of the error function of *x*.

```
►►──ERFC(x)──────────────────────────────────────────────────►◄
```
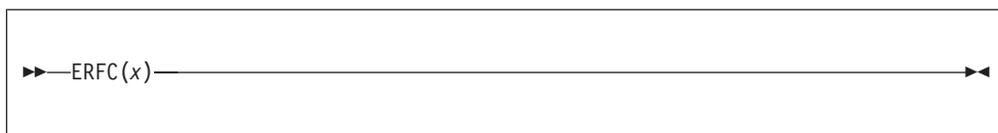
x   Real expression.

The result has the base and precision of *x*, and a value given by:
```
 1 - ERF(x)
```

EXP

EXP returns a floating-point value that is an approximation of the base, e, of the natural logarithm system raised to the power *x*.

```
►►──EXP(x)───────────────────────────────────────────────────►◄
```
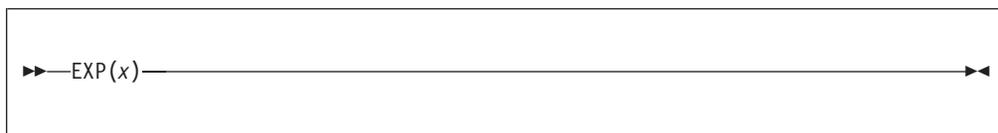
x   Expression.

The result has the base, mode, and precision of *x*.

EXPF

EXPF is exactly like EXP except that:
- EXPF calculates its result inline if hardware architecture permits.
- The argument must be a real expression.
- Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "EXP" on page 412.

EXPONENT

EXPONENT returns a FIXED BINARY(31,0) value that is the exponent part of x.

```
►►──EXPONENT(x)────────────────────────────────────►◄
```

x   Expression. *x* must be declared as REAL FLOAT.

EXPONENT(x) is not the "mathematical" exponent of *x*. If $x = 0$, EXPONENT(x) = 0. For other values of *x*, EXPONENT(x) is the unique number *e* such that:

$$radix(x)^{(e-1)} <= abs(x) < radix(x)^{e}$$

Consequently, EXPONENT(1e0) equals 1 and not 0.

FILEDDINT

FILEDDINT returns a FIXED BIN(31) value that is attribute *c* pertaining to file *x*.
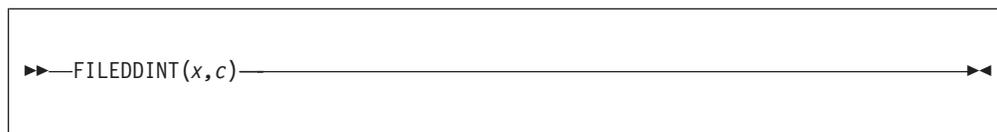
```
►►──FILEDDINT(x,c)─────────────────────────────────►◄
```

x   File reference.

c   Character string that holds the attribute to be queried.

When using FILEDDINT, the following are valid values for *c*:

bufsize	keyloc
delay	recsize
filesize	retry
keylen	

The ERROR condition with oncode 1010 is raised when the attribute is invalid for the file being queried.

FILEDDTEST

FILEDDTEST returns a FIXED BIN(31) value that holds the value 1 if the attribute *c* applies to file *x*. Otherwise, a value of 0 is returned.
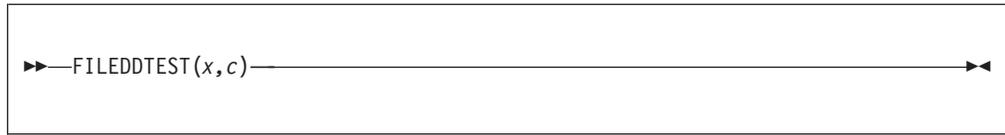
```
►►──FILEDDTEST(x,c)──────────────────────────────────►◄
```

x   File reference.

c   Character string that holds the attribute to be queried.

When using FILEDDTEST, the following are valid values for *c*:

append	graphic
bkwd	lrmskip
ctlasa	print
delimit	prompt
descendkey	scalarvarying
genkey	skip0

The ERROR condition with oncode 1010 is raised when the attribute is invalid for the file being queried.

FILEDDWORD

FILEDDWORD returns a character string that is the value of attribute *c* pertaining to file *x*.

```
►►──FILEDDWORD(x,c)──────────────────────────────────►◄
```

x   File reference.

c   Character string that holds the attribute to be queried.

When using FILEDDWORD, the following are valid values for *c*:

access	organization
amthd	putpage
action	share
charset	type
filename	typef

When you specify the *filename* option, the compiler returns the fully-qualified path name of the file.
• ACCESS returns SEQUENTIAL or DIRECT.
• ACTION returns INPUT, OUTPUT, or UPDATE.
• ORGANIZATION returns CONSECUTIVE, RELATIVE, REGIONAL(1) or INDEXED.
• TYPE returns RECORD or STREAM.

- TYPEF returns the type of the native file.

The ERROR condition with oncode 1010 is raised when the attribute is invalid for the file being queried.

FILEID

FILEID returns a FIXED BIN(31) value that is the system token for a PL/I file constant or variable.

```
►►──FILEID(x)──────────────────────────────────────►◄
```

x   PL/I file constant or variable.

This token should not be used for any purpose which could be accomplished by a PL/I statement.

FILEOPEN

FILEOPEN returns '1'B if the file *x* is open and '0'B if the file is not open.

```
►►──FILEOPEN(x)────────────────────────────────────►◄
```

x   File reference.

FILEREAD

FILEREAD attempts to read *z* storage units (bytes) from file *x* into location *y*. It returns the number of storage units actually read.
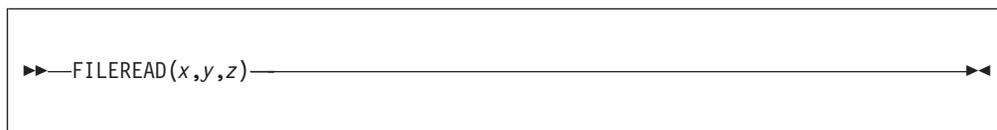
```
►►──FILEREAD(x,y,z)────────────────────────────────►◄
```

x   Reference with type FILE.

y   Expression with type POINTER or OFFSET. If the type is OFFSET, the expression must be an OFFSET variable declared with the AREA attribute.

z   Expression with computational type that is converted to FIXED BIN(31,0).

FILEREAD can read only TYPE(U) files.

FILESEEK

FILESEEK changes the current file position associated with file *x* to a new location within the file. The next operation on the file takes place at the new location. FILESEEK is equivalent to the fseek function in C.

```
►►──FILESEEK(x,y,z)──────────────────────────────────►◄
```

x Reference with type FILE.

y A FIXED BINARY(31) value that indicates the number of positions the file pointer is to be moved relative to *z*.

z A FIXED BINARY(31) value that indicates the origin from which the file pointer is to be moved. The following values are valid:
-1   Beginning of the file
0   Current position of the file pointer
1   End of the file.

FILESEEK can be used only on TYPE(U) files.

FILETELL

FILETELL returns a FIXED BINARY(31) value indicating the current position of the file *x*. The value returned is an offset relative to the beginning of the file. FILETELL is equivalent to the ftell function in C.

```
►►──FILETELL(x)──────────────────────────────────────►◄
```

x Reference with type FILE.

FILETELL can be used only on TYPE(U) files.

FILEWRITE

FILEWRITE attempts to write *z* storage units (bytes) to file *x* from location *y* It returns the number of storage units actually written.

```
►►──FILEWRITE(x,y,z)──────────────────────────────────►◄
```

x Reference with type FILE.

y Expression with type POINTER or OFFSET. If the type is OFFSET, the expression must be an OFFSET variable declared with the AREA attribute.

z Expression with computational type that is converted to FIXED BIN(31,0).

FILEWRITE can write only to TYPE(U) files.

FIXED

FIXED returns the fixed-point value of *x*, with a precision specified by *p* and *q*. The result has the base and mode of *x*.

```
►►──FIXED(x─────┬──────────┬──)──────────────────────────────►◄
                └─,p─┬─────┘
                     └─,q─┘
```
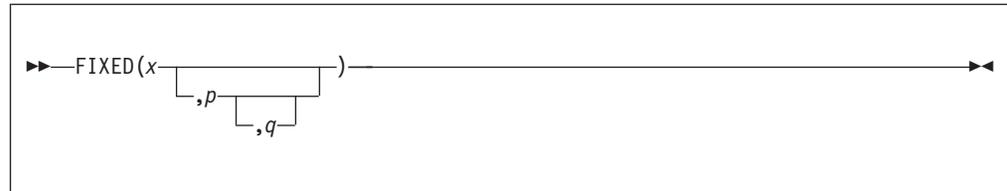
x Expression.

p Restricted expression that specifies the total number of digits in the result. It must not exceed the implementation limit.

q Restricted expression that specifies the scaling factor of the result. If *q* is omitted, a scaling factor of zero is assumed.

If both *p* and *q* are omitted, the precision of the result is determined from the rules for base conversion.

FLOAT

FLOAT returns the approximate floating-point value of *x*, with a precision specified by *p*. The result has the base and mode of *x*.

```
►►──FLOAT(x─────┬──────┬──)──────────────────────────────►◄
                └─,p─┘
```
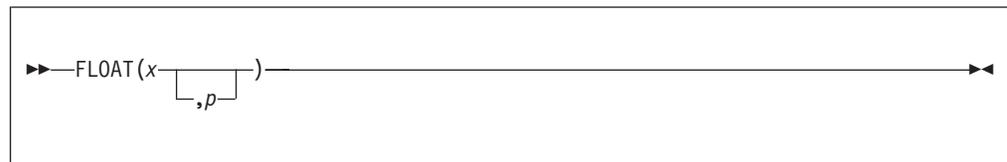
x Expression.

p Restricted expression that specifies the minimum number of digits in the result.

   If *p* is omitted, the precision of the result is determined from the rules for base conversion.

FLOOR

FLOOR determines the largest integer value less than or equal to *x*, and assigns this value to the result.

```
►►──FLOOR(x)──────────────────────────────────────────────►◄
```
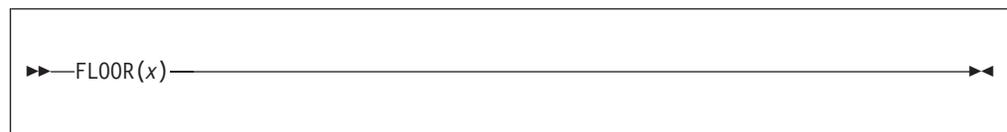
x Real expression.

The mode, base, scale, and precision of the result match the argument. Except when *x* is fixed-point with precision (*p,q*), the precision of the result is given by:

```
(min(n,max(p-q+1,1)),0)
```

where n is the maximum number of digits allowed and is N for FIXED DECIMAL or M for FIXED BINARY.

GAMMA

GAMMA is an approximation of the gamma of *x*, as given by the following equation:

$$\mathrm{gamma}(x) = \int_0^\infty (u^{x-1})(e^{-x})\,du$$

GAMMA returns a floating-point value that has the base, mode, and precision of *x*.

```
►►──GAMMA(x)──────────────────────────────────────►◄
```

x Real expression. The value of *x* must be greater than zero.

GETENV

GETENV returns a character value representing a specified environment variable.

```
►►──GETENV(x)──────────────────────────────────────►◄
```
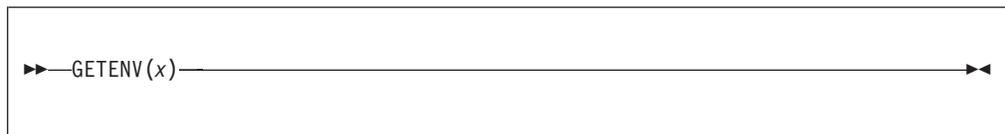
x Expression naming an environment variable.

GRAPHIC

GRAPHIC can be used to explicitly convert character (or mixed character) data to GRAPHIC data. All other data first converts to character, and then to the GRAPHIC data type.

GRAPHIC returns the graphic value of *x*, with a length in graphic symbols specified by *y*.

Characters convert to graphics. The content of *x* is checked for validity during conversion, using the same rules as for checking graphic and mixed character constants.

```
►►──GRAPHIC(x──┬──────┬──)──────────────────────────►◄
               └─,y──┘
```
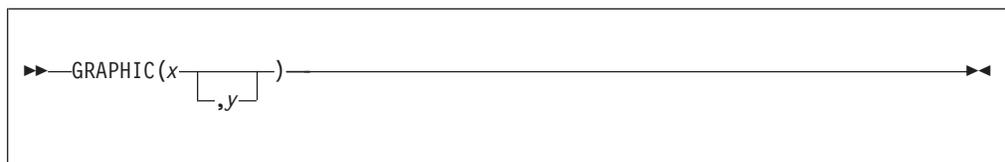
x Expression. When *x* is GRAPHIC, it is subject to a length change, with applicable padding or truncation. When *x* is nongraphic, it is converted to character, if necessary. SBCS characters are converted to equivalent DBCS characters.

y Expression. If necessary, *y* is converted to a real fixed-point binary value. If *y* is omitted, the length is determined by the rules for type conversion.

y must not be negative.

If $y = 0$, the result is the null graphic string.

The following rules apply:

- If y is greater than the length needed to contain the graphic string, the result is padded with graphic blanks.
- If y is less than the length needed to contain the graphic string, the result is truncated.

Example 1

Conversion from CHARACTER to GRAPHIC, where the target is long enough to contain the result:

```
dcl X char (11) varying;
dcl A graphic (11);
A = graphic(X,8);
```

For X with values	Intermediate Result	A is assigned
ABCDEFGHIJ	.A.B.C.D.E.F.G.H.I.J	.A.B.C.D.E.F.G.H.b.b.b
123	.1.2.3	.1.2.3.b.b.b.b.b.b.b.b
123A.B.C	.1.2.3.A.B.C	.1.2.3.A.B.C.b.b.b.b.b

where .b is a DBCS blank.

Example 2

Conversion from CHARACTER to GRAPHIC, where the target is too short to contain the result:

```
dcl X char (10) varying;
dcl A graphic (8);
A = graphic(X);
```

For X with value	Intermediate Result	A is assigned
ABCDEFGHIJ	.A.B.C.D.E.F.G.H.I.J	.A.B.C.D.E.F.G.H

HANDLE

HANDLE returns a handle to the typed structure *x*.

►►──HANDLE(*x*)──────────────────────────►◄
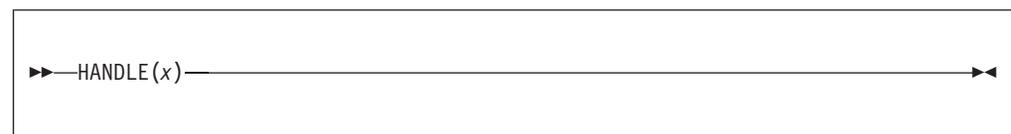
x   Typed structure.

HBOUND

HBOUND returns a FIXED BINARY(31,0) value specifying the current upper bound of dimension *y* of *x*.
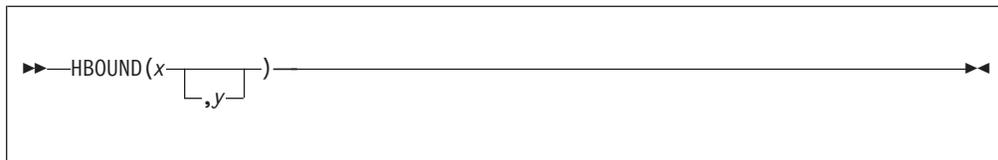
```
►►──HBOUND(x──────)──────────────────────────────────►◄
              └─,y─┘
```

x   Array reference. *x* must not have less than *y* dimensions.

y   Expression specifying a particular dimension of *x*. If necessary, *y* is converted to FIXED BINARY(31,0). *y* must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

    y can be omitted only if the array is one-dimensional.

HEX

HEX returns a character string that is the hexadecimal representation of the storage that contains *x*.

```
►►──HEX(x──────)──────────────────────────────────────►◄
           └─,z─┘
```
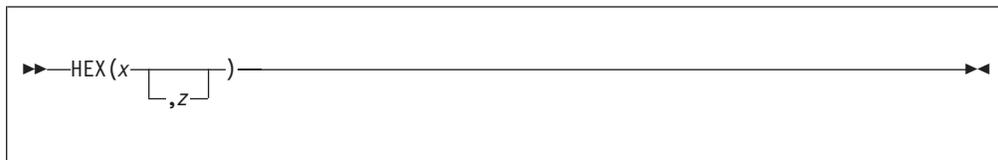
HEX(x) returns a character string of length 2 * size(x).

HEX(x,z) returns a character string that contains *x* with the character *z* inserted between every set of eight characters in the output string. Its length is 2 * size(x) + ((size(x) - 1)/4).

x   Expression that represents any variable. The whole number of bytes that contain *x* is converted to hexadecimal.

z   Expression. If specified, *z* must have the type CHARACTER(1) NONVARYING.

Example 1

```
dcl Sweet char(5) init('Sweet');
dcl Sixteen fixed bin(31) init(16);
dcl XSweet char(size(Sweet)*2+(size(Sweet)-1)/4);
dcl XSixteen char(size(Sixteen)*2+(size(Sixteen)-1)/4);

XSweet = hex(Sweet,'-');
        /* '53776565-74' */

XSweet = heximage(addr(Sweet),length(Sweet),'-');
        /* '53776565-74' */

XSixteen = hex(Sixteen,'-');
        /* '10000000' - bytes NOT reversed */

XSixteen = heximage(addr(Sixteen),stg(Sixteen),'-');
        /* '00000010' - bytes reversed */
```

Example 2

```
dcl X fixed bin(15) littleendian;
dcl Y fixed bin(15) bigendian;

X = 258;                                    /* stored as '0201'B4 */
Y = 258;                                    /* stored as '0102'B4 */

display (hex(X));                           /* displays 0102 */
display (hex(Y));                           /* displays 0102 */

display (heximage( addr(X), stg(X) ));      /* displays 0201 */
display (heximage( addr(Y), stg(Y) ));      /* displays 0102 */
```

Note: This function does not return an exact image of *x* in storage. If an exact image is required, use the HEXIMAGE built-in function.

HEXIMAGE

HEXIMAGE returns a character string that is the hexadecimal representation of the storage at a specified location.

```
►►──HEXIMAGE(p,n──────)──────────────────────────────────►◄
                └─,z─┘
```
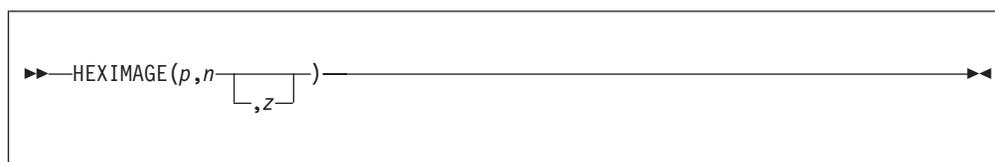
HEXIMAGE(p,n) returns a character string that is the hexadecimal representation of *n* bytes of storage at location *p*. Its length is $2 * n.$.

HEXIMAGE(p,n,z) returns a character string that is the hexadecimal representation of *n* bytes of storage at location *p* with character *z* inserted between every set of eight characters in the output string. Its length is $(2 * n) + ((n - 1)/4)$.

p Restricted expression that must have a locator type (POINTER or OFFSET). If *p* is OFFSET, it must have the AREA attribute.

n Expression. *n* must have a computational type and is converted to FIXED BINARY(31,0).

z If specified, *z* must have the type CHARACTER(1) NONVARYING.

For examples of the HEXIMAGE built-in function, see "HEX" on page 420.

HIGH

HIGH returns a character string of length *x*, where each character is the highest character in the collating sequence (hexadecimal FF).
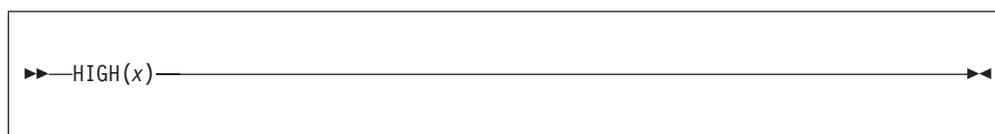
```
►►──HIGH(x)──────────────────────────────────────────────►◄
```

x Expression. If necessary, *x* is converted to a positive real fixed-point binary value. If $x = 0$, the result is the null character string.

HUGE

HUGE returns a floating-point value that is the largest positive value *x* can assume. It has the base, mode, and precision of *x*.

```
►►──HUGE(x)──────────────────────────────────────────────────────►◄
```

x   Expression. *x* must have the attributes REAL FLOAT.

HUGE(x) is a constant and can be used in restricted expressions.

IAND

IAND returns the logical AND of its arguments

```
                  ┌─,─┐
►►──IAND(─x,─▼─y─┴─)───────────────────────────────────────────►◄
```

x and y
   Expressions that must have a computational type.

If any argument is not REAL FIXED BIN(p,0), then it is converted to SIGNED REAL FIXED BIN(M,0).

If any argument is SIGNED, then any UNSIGNED arguments are converted to SIGNED.

The result is REAL FIXED BIN(max(p1,p2,...), 0). It is UNSIGNED if all the arguments are UNSIGNED.

IEOR

IEOR returns the logical exclusive-OR of *x* and *y*. The result is unsigned if all arguments are unsigned.
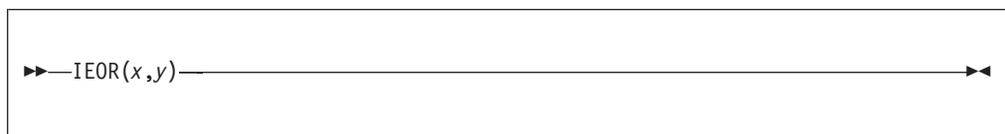
```
►►──IEOR(x,y)─────────────────────────────────────────────────►◄
```

x and y
   Expressions that must have a computational type.

If any argument is not REAL FIXED BIN(p,0), then it is converted to SIGNED REAL FIXED BIN(M,0).

If any argument is SIGNED, then any UNSIGNED arguments are converted to SIGNED.

The result is REAL FIXED BIN(max(p1,p2,...), 0). It is UNSIGNED if all the arguments are UNSIGNED.

IMAG

IMAG returns the coefficient of the imaginary part of *x*. The mode of the result is real and has the base, scale, and precision of *x*.

```
►►──IMAG(x)──────────────────────────────────────────►◄
```
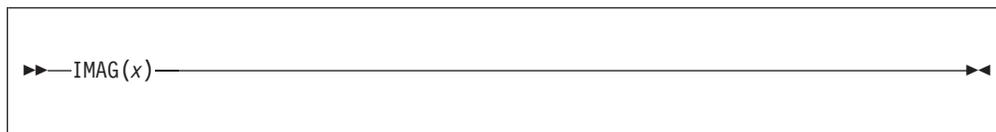
x   Expression. If *x* is real, it is converted to complex.

IMAG pseudovariable

The IMAG pseudovariable assigns a real value or the real part of a complex value to the coefficient of the imaginary part of *x*.

```
►►──IMAG(x)──────────────────────────────────────────►◄
```
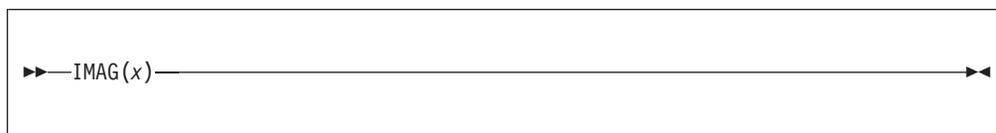
x   Complex reference.

INDEX

INDEX returns an unscaled REAL FIXED BINARY value indicating the starting position within *x* of a substring identical to *y*. You can also specify the location within *x* where processing begins.

```
►►──INDEX(x,y─────────)──────────────────────────────►◄
              └─,n─┘
```
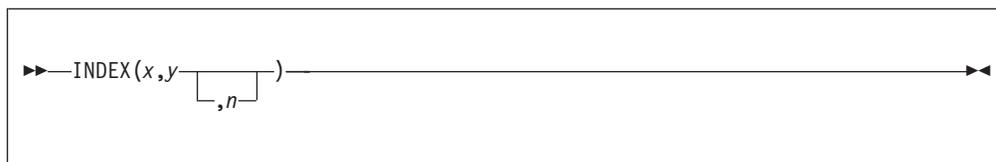
x   String-expression to be searched.

y   Target string-expression of the search.

n   *n* specifies the location within *x* at which to begin processing. It must have a computational type and is converted to FIXED BINARY(31,0).

If *y* does not occur in *x*, or if either *x* or *y* have zero length, the value zero is returned.

If *n* is less than 1 or if *n* is greater than 1 + length(x), the STRINGRANGE condition will be raised, and the result will be 0.

The BIFPREC compiler option determines the precision of the result returned.

Example

```
dcl tractatus char
      value( 'Wovon man nicht sprechen kann, ' ‖
             'darueber muss man schweigen.' );

dcl pos fixed bin init(1);

pos = index( tractatus, 'man', pos+1 ); /* pos = 07 */

pos = index( tractatus, 'man', pos+1 ); /* pos = 46 */

pos = index( tractatus, 'man', pos+1 ); /* pos = 00 */
```

INOT

INOT returns the logical NOT of *x*.

```
►►──INOT(x)──────────────────────────────────────────►◄
```
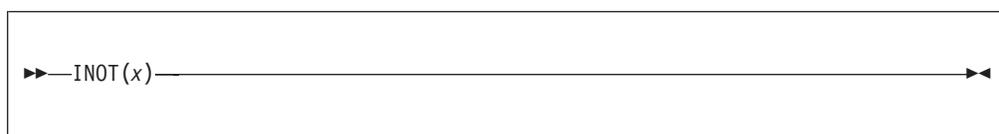
x   Expression. *x* must have a computational type.

If *x* is REAL FIXED BIN(p,0), then the result is REAL FIXED BIN(p,0) and it is UNSIGNED if *x* is UNSIGNED. Otherwise, *x* is converted to SIGNED REAL FIXED BIN(M,0) and the result has the same attributes.

Although INOT(x) has the opposite sign of *x*, INOT(x) is not the same as –*x*.

Examples

```
inot(0)        /*  produces −1  */
inot(−1)       /*  produces  0  */
inot(+1)       /*  produces −2  */
```

IOR

IOR returns the logical OR of its arguments.
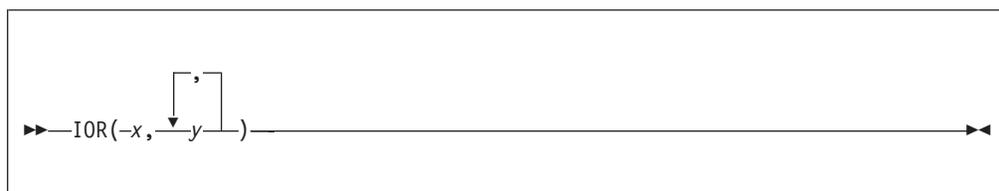
```
         ┌─,─┐
►►──IOR(─x,─▼─y─┴─)──────────────────────────────────────►◄
```

x and y
    Expressions that must have a computational type.

If any argument is not REAL FIXED BIN(p,0), then it is converted to SIGNED REAL FIXED BIN(M,0).

If any argument is SIGNED, then any UNSIGNED arguments are converted to SIGNED.

The result is REAL FIXED BIN(max(p1,p2,...), 0). It is UNSIGNED if all the arguments are UNSIGNED.

ISIGNED

ISIGNED(x) returns the result of casting *x* to a signed integer value without changing its bit pattern.
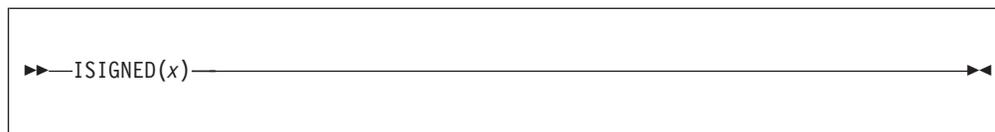
```
►►──ISIGNED(x)───────────────────────────────────────────────────────────►◄
```

x Expression. *x* must have a computational type.

If *x* is not an integer, i.e. if *x* is not REAL FIXED BIN with zero scale factor, then it is converted to REAL FIXED BIN(M,0).

ISIGNED(*x*) returns, for integer *x*, a value with the same bit pattern as *x* but the attributes SIGNED FIXED BIN(p).

If *x* is UNSIGNED, p is given by:

If precision(*x*) = 8, 16, 32 or 64, then p = precision(*x*) - 1 else p = precision(*x*)

If *x* is SIGNED, p is equal to the precision of *x*.

Examples

```
ISIGNED('ff_ff_ff_ff'xu) equals the SIGNED FIXED BIN(31) value -1.
```

ISLL

ISLL(x,n) returns the result of logically shifting *x* to the left by *n* places, and padding on the right with zeroes.

```
►►──ISLL(x,n)─────────────────────────────────────────────────────────────►◄
```

x Expression. *x* must have a computational type.

n Expression. *n* must have a computational type.

If *x* is REAL FIXED BIN(p,0) and:
- *x* is SIGNED, then the result is SIGNED REAL FIXED BIN(M,0).
- *x* is UNSIGNED, the result is UNSIGNED REAL FIXED BIN(M+1,0).

Otherwise, *x* is converted to SIGNED REAL FIXED BIN(M,0) and the result has the same attributes.

The result is undefined if *n* is negative or if *n* is greater than M.

Note: Unlike RAISE2(x,n), ISLL(x,n) can have a different sign than *x* does.

Examples

```
isll(+6,1)              /* produces 12   */
isll(2147483645,1)      /* produces -6   */
```

ISMAIN

ISMAIN()) returns a '1'B if the procedure in which it is invoked has the
OPTIONS(MAIN) attribute. Otherwise it returns a '0'B.

```
►►──ISMAIN──(──)───────────────────────────────────►◄
```

ISRL

ISRL(x,n) returns the result of logically shifting *x* to the right by *n* places, and
padding on the left with zeroes.

```
►►──ISRL(x,n)──────────────────────────────────────►◄
```

x   Expression. *x* must have a computational type.

n   Expression. *n* must have a computational type.

If *x* is REAL FIXED BIN(p,0) and:
- *x* is SIGNED, then the result is SIGNED REAL FIXED BIN(p,0).
- *x* is UNSIGNED, the result is UNSIGNED REAL FIXED BIN(p,0).

Otherwise, *x* is converted to SIGNED REAL FIXED BIN(M,0) and the result has the
same attributes.

The result is undefined if *n* is negative or if *n* is greater than M.

If *x* is nonnegative, ISRL(x,n) is equivalent to LOWER2(x,n); if *x* is negative,
ISRL(x,n) is positive, unless *n*=0.

Examples

```
isrl(+6,1)            /*  produces 3          */
isrl(-6,1)            /*  produces 2147483645  */
```

IUNSIGNED

IUNSIGNED(x) returns the result of casting *x* to an unsigned integer value without
changing its bit pattern.

```
►►──IUNSIGNED(x)───────────────────────────────────►◄
```

x   Expression. *x* must have a computational type.

If *x* is not an integer, i.e. if *x* is not REAL FIXED BIN with zero scale factor, then it is
converted to REAL FIXED BIN(M,0).

IUNSIGNED(*x*) returns, for integer *x*, a value with the same bit pattern as *x* but the attributes UNSIGNED FIXED BIN(p).

If *x* is SIGNED, p is given by:

If precision(*x*) = 7, 15, 31 or 63, then p = precision(*x*) + 1 else p = precision(*x*)

If *x* is UNSIGNED, p is equal to the precision of *x*.

Examples

```
IUNSIGNED('ff_ff_ff_ff'xn) equals the largest UNSIGNED FIXED BIN(32) value.
```

LBOUND

LBOUND returns a FIXED BINARY (31,0) value specifying the current lower bound of dimension *y* of *x*.
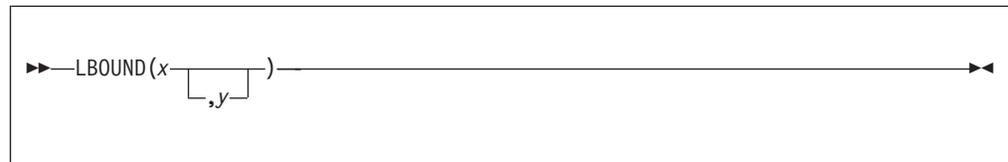
```
►►──LBOUND(x────────)──────────────────────────────►◄
              └─,y─┘
```

x   Array reference. *x* must not have less than *y* dimensions.

y   Expression specifying a particular dimension of *x*. If necessary, *y* is converted to FIXED BINARY(31,0). The value for *y* must be greater than or equal to 1. and if *y* is not supplied, it defaults to 1.

The value for *y* can be omitted only if the array is one-dimensional.

LEFT

LEFT returns a string that is the result of inserting string *x* at the left end of a string with length *n* and padded on the right with the character *z* as needed.
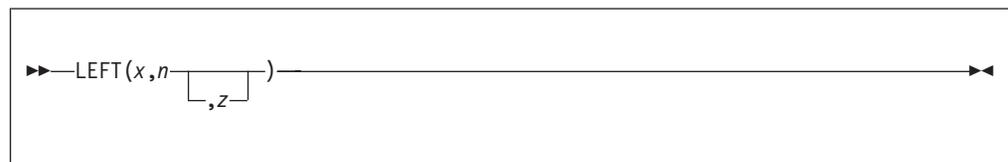
```
►►──LEFT(x,n────────)──────────────────────────────►◄
              └─,z─┘
```

x   Expression. *x* must have a computational type and should have a character type. If not, it is converted to CHARACTER.

n   Expression. *n* must have a computational type and should have a character type. If *n* does not have the attributes FIXED BINARY(31,0), it is converted to them.

z   Expression. If specified, *z* must have the type CHARACTER(1) NONVARYING type.

Example

```
dcl Source char value('One Hundred SCIDS Marks');
dcl Target char(30);

Target = left (Source, length(Target), '*');
        /* 'One Hundred SCIDS Marks*******'           */
```

If *z* is omitted, a blank is used as the padding character.

LENGTH

LENGTH returns an unscaled REAL FIXED BINARY value specifying the current length of *x*.

```
►►──LENGTH(x)──────────────────────────────────────────────►◄
```

x String-expression. If *x* is binary, it is converted to bit string; otherwise, any other conversion required is to character string.

For an example of the LENGTH built-in function, refer to "MAXLENGTH" on page 433.

The BIFPREC compiler option determines the precision of the result returned.

LINENO

LINENO returns an unscaled REAL FIXED BINARY specifying the current line number of *x*.

```
►►──LINENO(x)──────────────────────────────────────────────►◄
```

x File-reference.

The file must be open and have the PRINT attribute. If the file is not open or does not have the PRINT attribute, '0'B is returned.

The BIFPREC compiler option determines the precision of the result returned.

LOCATION

LOCATION returns a FIXED BINARY(31,0) value specifying the byte location of *x* within the level-1 structure or union that has member *x*.

```
►►──LOCATION(x)────────────────────────────────────────────►◄
```
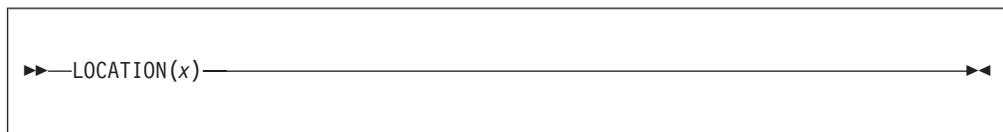
Abbreviation: LOC

x Structure or union member name. If *x* is not a member of a structure or union, a value of 0 is returned. If *x* has the BIT attribute, the value returned by LOCATION is the location of the byte that contains *x*.

The value for *x* must not be subscripted.

LOCATION can be used in restricted expressions, with a limitation. The value for *x* must be declared before *y* if LOC(x) is used to set either of the following:
*   The extent of a variable *y* that must have constant extents
*   The value of a variable *y* that must have a constant value.

Example

```
dcl 1 Table static,
      2 Tab2loc fixed bin(15) nonasgn init(loc(Tab2)),
                /* location is 0; gets initialized to 8  */
      2 Tab3loc fixed bin(15) nonasgn init(loc(Tab3)),
                /* location is 2; gets initialized to 808 */
      2 Length fixed bin nonasgn init(loc(End)),
                /* location is 4 */
      2 * fixed bin,
      2 Tab2(20,20)    fixed bin,
                /* location is 8 */
      2 Tab3(20,20)    fixed bin,
                /* location is 808 */

      2 F2_loc fixed bin nonasgn init(loc(F2)),
                /* location is 1608; gets initialized to  1612 */
      2 F2_bitloc fixed bin nonasgn init(bitloc(F2)),
                /* location is 1610; gets initialized to 1 */

      2 Flags,        /* location is 1612 */
        3 F1 bit(1),
        3 F2 bit(1),  /* bitlocation is 1 */
        3 F3 bit(1),
      2 Bits(16) bit, /* location is 1613 */
      2 End char(0);
```

LOG

LOG returns a floating-point value that is an approximation of the natural logarithm (the logarithm to the base e) of *x*. It has the base, mode, and precision of *x*.

```
►►──LOG(x)──────────────────────────────────────────────────────►◄
```
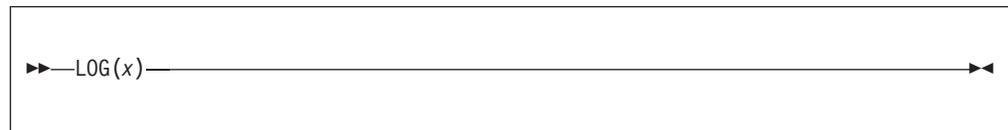
x   Expression. *x* must be greater than zero.

LOGF

LOGF is exactly like LOG except that:
*   LOGF calculates its result inline if hardware architecture permits.
*   The argument must be a real expression.
*   Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
*   The accuracy of the result is set by the hardware.

For the definition and syntax, see "LOG."

LOGGAMMA

LOGGAMMA returns a floating-point value that is an approximation of the log of gamma of *x*. The gamma of *x* is given by the following equation:

$$\text{gamma}(x) = \int_0^\infty (u^{x-1})(e^{-x})\,du$$

LOGGAMMA

LOGGAMMA has the base, mode, and precision of *x*.

```
►►──LOGGAMMA(x)────────────────────────────────────────────────►◄
```
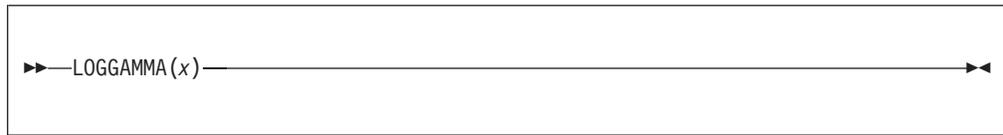
x Real expression. The value of *x* must be greater than 0.

LOG2

LOG2 returns a real floating-point value that is an approximation of the binary logarithm (the logarithm to the base 2) of *x*. It has the base and precision of *x*.

```
►►──LOG2(x)────────────────────────────────────────────────────►◄
```
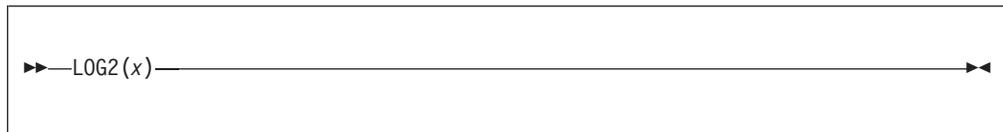
x Real expression. The value of *x* must be greater than zero.

LOG10

LOG10 returns a real floating-point value that is an approximation of the common logarithm (the logarithm to the base 10) of *x*. It has the base and precision of *x*.

```
►►──LOG10(x)───────────────────────────────────────────────────►◄
```
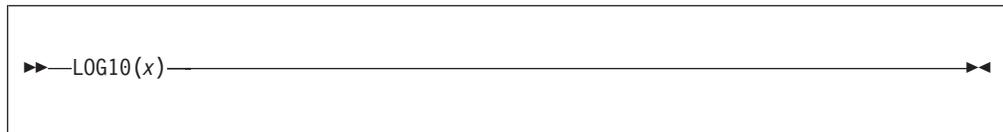
x Real expression. It must be greater than zero.

LOG10F

LOG10F is exactly like LOG10 except that:
- LOG10F calculates its result inline if hardware architecture permits.
- The argument must be a real expression.
- Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "LOG10."

LOW

LOW returns a character string of length *x*, where each character is the lowest character in the collating sequence (hexadecimal 00).
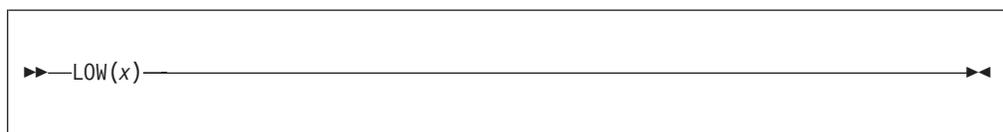
```
►►──LOW(x)─────────────────────────────────────────────────────►◄
```

x Expression. If necessary, *x* is converted to a positive real fixed-point binary value. If *x* = 0, the result is the null character string.

LOWERCASE

LOWERCASE returns a character string with all the alphabetic characters from A to Z converted to their lowercase equivalent.

```
►►──LOWERCASE(x)────────────────────────────────────────►◄
```
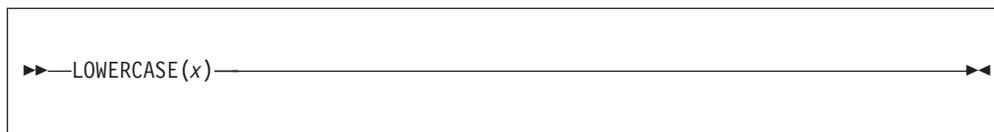
x   Expression. If necessary, *x* is converted to character.

LOWERCASE(x) is equivalent to

```
TRANSLATE( x,
           'abcdefghijklmnopqrstuvwxyz',
           'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  )
```

LOWER2

LOWER2(x,n) returns the value:

$$floor(x * (2^{-n}))$$

LOWER2(x,n) returns the value:

$$floor(x * (2^{-n}))$$

```
►►──LOWER2(x,n)──────────────────────────────────────────►◄
```

Note: LOWER2(x,n) is equivalent to the assembler SRA(x,n).

x   Expression. *x* must have a computational type.

n   Expression. *n* must have a computational type.

If *x* is SINGED REAL FIXED BIN(p,0), then the result has the same attributes. Otherwise, *x* is converted to SIGNED REAL FIXED BIN(M,0) and the result has the same attributes.

The result is undefined if *n* is negative or if *n* is greater than M.

Examples

```
lower2 (+6,1)                    /*  Produces 3   */

lower2 (-6,1)                    /*  Produces -3  */

lower2 (-7,1)                    /*  Produces -4  */
```

MAX

MAX returns the largest value from a set of two or more expressions.
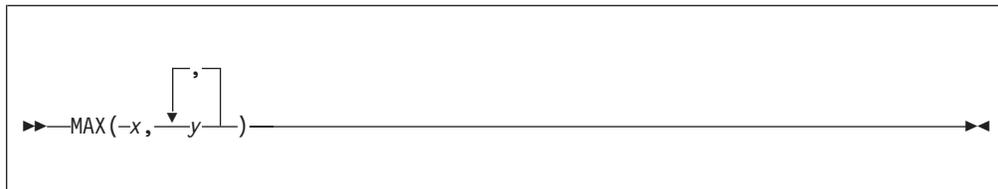
x and y
  Expressions.

All the arguments must be real. The result is real, with the common base and scale of the arguments.

If the arguments are fixed-point with precisions:

```
(p1,q1),(p2,q2),...,(pn,qn)
```

then the precision of the result is given by:

```
(min(N,max(p1-q1,p2-q2,...,pn-qn) + max(q1,q2,...,qn)),max(q1,q2,...,qn))
```

where N is the maximum number of digits allowed.

If the arguments are floating-point with precisions:

```
p1,p2,p3,...pn
```

then the precision of the result is given by:

```
max(p1,p2,p3,...pn)
```

MAXEXP

MAXEXP returns a FIXED BINARY(31,0) value that is the maximum value that EXPONENT(x) could assume.
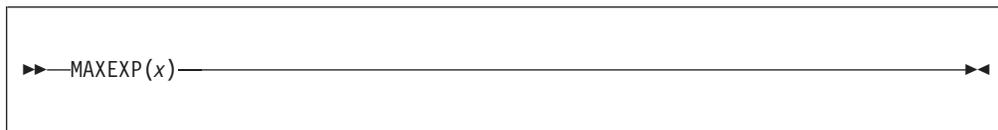
x   Expression. *x* must have the REAL and FLOAT attributes.

Example (Intel Values)

```
maxexp(x) = 00128      for x float bin(p), p <= 21
maxexp(x) = 01024      for x float bin(p), 21 < p <= 53
maxexp(x) = 16384      for x float bin(p), 53 < p

maxexp(x) = 00128      for x float dec(p), p <= 6
maxexp(x) = 01024      for x float dec(p), 6 < p <= 16
maxexp(x) = 16384      for x float dec(p), 16 < p
```

Example (AIX Values)

```
maxexp(x) = 0128       for x float bin(p), p <= 21
maxexp(x) = 1024       for x float bin(p), 21 < p <= 53
maxexp(x) = 1024       for x float bin(p), 53 < p
```

```
maxexp(x) = 0128      for x float dec(p), p <= 6
maxexp(x) = 1024      for x float dec(p), 6 < p <= 16
maxexp(x) = 1024      for x float dec(p), 16 < p
```

Example (z/OS Hexdecimal Values)

```
maxexp(x) = 63        for x float bin(p), p <= 21
maxexp(x) = 63        for x float bin(p), 21 < p <= 53
maxexp(x) = 63        for x float bin(p), 53 < p

maxexp(x) = 63        for x float dec(p), p <= 6
maxexp(x) = 63        for x float dec(p), 6 < p <= 16
maxexp(x) = 63        for x float dec(p), 16 < p
```

Example (z/OS IEEE Values)

```
maxexp(x) = 128       for x float bin(p), p <= 21
maxexp(x) = 1024      for x float bin(p), 21 < p <= 53
maxexp(x) = 16384     for x float bin(p), 53 < p

maxexp(x) = 128       for x float dec(p), p <= 6
maxexp(x) = 1024      for x float dec(p), 6 < p <= 16
maxexp(x) = 16384     for x float dec(p), 16 < p
```

MAXEXP(x) is a constant and can be used in restricted expressions.

MAXLENGTH

MAXLENGTH returns the maximum length of a string.
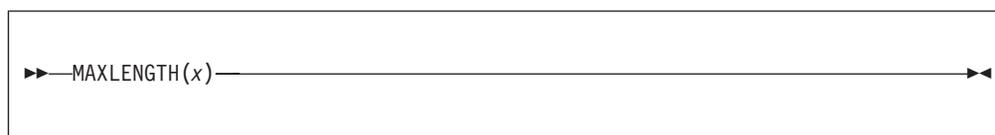
```
►►──MAXLENGTH(x)──────────────────────────────────────►◄
```

x Expression. *x* must have a computational type and should have a string type. If not, it is converted to character.

Example

```
dcl x char(20);
dcl y char(20) varying;

x, y = '';

x = copy( '*', length(x) );     /* fills x with '*'   */
y = copy( '*', length(y) );     /* leaves y unchanged */

x = copy( '-', maxlength(x) ); /* fills x with '-'   */
y = copy( '-', maxlength(y) ); /* fills y with '-'   */
```

Note that the first assignment to *y* leaves it unchanged because *length(y)* will return zero when it is used in the code snippet above (since *y* is VARYING and was previously set to ").

However, the second assignment to *y* fills it with 20 – signs because *maxlength(y)* will return 20 (the declared length of *y*).

MEMINDEX

MEMINDEX returns an unscaled REAL FIXED BINARY value indicating the starting position within a buffer of a specified substring.

With 3 arguments, the function's syntax is :

```
►►──MEMINDEX──(──p──,──n──,──x──)────────────────────────────────►◄
```
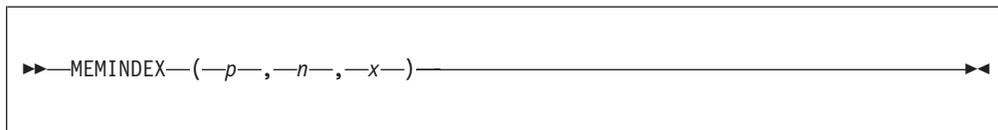
p   Address of buffer to be searched

n   Length of buffer to be searched.

x   String-expression to use as the target of the search.

With 4 arguments, the function's syntax is

```
►►──MEMINDEX──(──p──,──n──,──q──,──m──)──────────────────────────►◄
```

p   Address of first buffer to be searched.

n   Length of first buffer to be searched.

q   Address of second buffer to use as the target of the search.

m   Length of second buffer to use as the target of the search.

The buffer lengths must have a computational type and will be converted to FIXED BINARY(31,0).

The buffer lengths must be nonnegative

With 3 arguments, the target string-expression must have type CHARACTER (including PICTURE), GRAPHIC or WIDECHAR. The buffer length is interpreted as the number of units of that string type.

With 4 arguments, the buffer lengths specify a number of bytes and the search performed is a character search.

For a VARYING or VARYINGZ string *X* and string *Y*, the function MEMINDEX(ADDRDATA(X), LENGTH(X), Y) will return the same value as INDEX(X, Y).

Example

```
dcl cb(128*1024) char(1);
dcl wb(128*1024) widechar(1);
dcl pos fixed bin(31);
/* 128K bytes searched for the character string 'test' */
pos = memindex( cb, stg(cb), 'test' );
/* 256K bytes searched for the string 'test' as widechar */
pos = memindex( wb, stg(wb), wchar('<') );
```

MEMSEARCH

MEMSEARCH returns an unscaled REAL FIXED BINARY value specifying the first position (from the left) in a buffer at which `any` character, graphic or widechar in a given string appears.

```
►►──MEMSEACRH──(──p──,──n──,──x──)──────────────────────►◄
```

p Address of buffer to be searched.

n Length of buffer to be searched.

x String-expression.

The buffer length must have a computational type and will be converted to FIXED BINARY(31,0).

The buffer length must be nonnegative.

The string-expression *x* must have type CHARACTER (including PICTURE), GRAPHIC or WIDECHAR. The buffer length is interpreted as the number of units of that string type.

The address *p* and the length *n* specify the ″string″ in which to search for any character, graphic or widechar that appears in *x*.

If either the buffer length *n* is zero or *x* is the null string, the result is zero.

If *x* does not occur in the buffer, the result is zero.

Example

```
dcl cb(128*1024) char(1);
dcl wb(128*1024) widechar(1);
dcl pos fixed bin(31);

/* 128K bytes searched from the left for a numeric */
pos = memsearch( cb, stg(cb), '012345789' );

/* 256K bytes searched from the left for a widechar '0' or '1' */
pos = memsearch( wb, stg(wb), '0030_0031'wx );
```

MESEARCHR

MEMSEARCHR returns an unscaled REAL FIXED BINARY value specifying the first position (from the right) in a buffer at which `any` character, graphic or widechar in a given string appears

```
►►──MEMSEACRHR──(──p──,──n──,──x──)──────────────────────►◄
```

p Address of buffer to be searched.

n Length of buffer to be searched.

x String-expression.

The buffer length must have a computational type and will be converted to FIXED BINARY(31,0).

The buffer length must be nonnegative.

The string-expression *x* must have type CHARACTER (including PICTURE), GRAPHIC or WIDECHAR. The buffer length is interpreted as the number of units of that string type.

The address *p* and the length *n* specify the ″string″ in which to search for any character, graphic or widechar that appears in *x*.

If either the buffer length *n* is zero or *x* is the null string, the result is zero.

If *x* does not occur in the buffer, the result is zero.

Example

```
dcl cb(128*1024) char(1);
dcl wb(128*1024) widechar(1);
dcl pos fixed bin(31);

/* 128K bytes searched from the right for a numeric */
pos = memsearchr( cb, stg(cb), '012345789' );

/* 256K bytes searched from the right for a widechar '0' or '1' */
pos = memsearchr( wb, stg(wb), '0030_0031'wx );
```

MEMVERIFY

MEMVERIFY returns an unscaled REAL FIXED BINARY value specifying the position in a buffer of the first (from the left) character, graphic or widechar that is *not* in a specified string.

```
►►──MEMVERIFY──(──p──,──n──,──x──)──────────────────────────►◄
```

p Address of buffer to be searched.

n Length of buffer to be searched.

x String-expression.

The buffer length must have a computational type and will be converted to FIXED BINARY(31,0).

The buffer length must be nonnegative.

The string-expression *x* must have type CHARACTER (including PICTURE), GRAPHIC or WIDECHAR. The buffer length is interpreted as the number of units of that string type.

The address *p* and the length *n* specify the ″string″ in which to search for any character, graphic or widechar that does not appear in *x*.

If either the buffer length *n* is zero or *x* is the null string, the result is zero.

If all the characters, graphics or widechars in the buffer do appear in *x*, the result is zero.

Example

```
 dcl cb(128*1024) char(1);
dcl wb(128*1024) widechar(1);
dcl pos fixed bin(31);

/* 128K bytes searched from the left for a non-numeric */
pos = memverify( cb, stg(cb), '012345789' );

/* 256K bytes searched from the left for the a non-blank widechar */
pos = memverify( wb, stg(wb), '0020'wx );
```

MEMVERIFYR

MEMVERIFYR returns an unscaled REAL FIXED BINARY value specifying the position in a buffer of the first (from the right) character, graphic or widechar that is *not* in a specified string.

```
►►──MEMVERIFYR──(──p──,──n──,──x──)──────────────────────────────────────►◄
```

p Address of buffer to be searched.

n Length of buffer to be searched.

x String-expression.

The buffer length must have a computational type and will be converted to FIXED BINARY(31,0).

The buffer length must be nonnegative.

The string-expression *x* must have type CHARACTER (including PICTURE), GRAPHIC or WIDECHAR. The buffer length is interpreted as the number of units of that string type.

The address *p* and the length *n* specify the ″string″ in which to search for any character, graphic or widechar that does not appear in *x*.

If either the buffer length *n* is zero or *x* is the null string, the result is zero.

If all the characters, graphics or widechars in the buffer do appear in *x*, the result is zero.

Example

```
 dcl cb(128*1024) char(1);
 dcl wb(128*1024) widechar(1);
 dcl pos fixed bin(31);

 /* 128K bytes searched from the right for a non-numeric */
 pos = memverify( cb, stg(cb), '012345789' );
```

```
                    /* 256K bytes searched from the right for the a non-blank widechar */
                    pos = memverify( wb, stg(wb), '0020'wx );
```

MIN

MIN returns the smallest value from a set of one or more expressions.
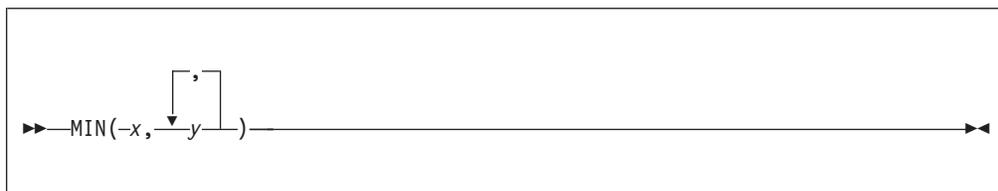
```
►►──MIN(─x,──┬─y─┬──)─────────────────────────────────────►◄
            └─,◄─┘
```

x and y
    Expressions.

All the arguments must be real. The result is real with the common base and scale of the arguments.

The precision of the result is the same as that described in "MAX" on page 432.

MINEXP

MINEXP returns a FIXED BINARY(31,0) value that is the minimum value that EXPONENT(x) could assume.

```
►►──MINEXP(x)─────────────────────────────────────────────►◄
```
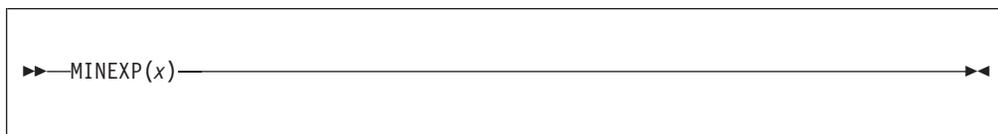
x   Expression. *x* must have the REAL and FLOAT attributes.

MINEXP(x) is a constant and can be used in restricted expressions.

Example (Intel Values)

```
minexp(x) = -00125    for x float bin(p), p <= 21
minexp(x) = -01021    for x float bin(p), 21 < p <= 53
minexp(x) = -16831    for x float bin(p), 53 < p

minexp(x) = -00125    for x float dec(p), p <= 6
minexp(x) = -01021    for x float dec(p), 6 < p <= 16
minexp(x) = -16831    for x float dec(p), 16 < p
```

Example (AIX Values)

```
minexp(x) = -0125     for x float bin(p), p <= 21
minexp(x) = -1021     for x float bin(p), 21 < p <= 53
minexp(x) = -0968     for x float bin(p), 53 < p

minexp(x) = -0125     for x float dec(p), p <= 6
minexp(x) = -1021     for x float dec(p), 6 < p <= 16
minexp(x) = -0968     for x float dec(p), 16 < p
```

Example (z/OS Hexadecimal Values)

```
minexp(x) = -64    for x float bin(p), p <= 21
minexp(x) = -64    for x float bin(p), 21 < p <= 53
minexp(x) = -50    for x float bin(p), 53 < p

minexp(x) = -64    for x float dec(p), p <= 6
minexp(x) = -64    for x float dec(p), 6 < p <= 16
minexp(x) = -50    for x float dec(p), 16 < p
```

Example (z/OS IEEE Values)

```
maxexp(x) = -125    for x float bin(p), p <= 21
maxexp(x) = -1021   for x float bin(p), 21 < p <= 53
maxexp(x) = -16381  for x float bin(p), 53 < p

maxexp(x) = -125    for x float dec(p), p <= 6
maxexp(x) = -1021   for x float dec(p), 6 < p <= 16
maxexp(x) = -16381  for x float dec(p), 16 < p
```

MOD

MOD returns the smallest nonnegative value, R, such that:

```
(x - R)/y = n
```

In this example, the value for n is an integer value. That is, R is the smallest nonnegative value that must be subtracted from x to make it divisible by y.

```
►►──MOD(x,y)──────────────────────────────────────────►◄
```

x   Real expression.

y   Real expression. If $y = 0$, the ZERODIVIDE condition is raised.

The result, R, is real with the common base and scale of the arguments. If the result is floating-point, the precision is the greater of those of x and y. If the result is fixed-point, the precision is given by the following:

```
(min(n,p2-q2+max(q1,q2)),max(q1,q2))
```

In this example, (p1,q1) and (p2,q2) are the precisions of x and y, respectively, and n is N for FIXED DECIMAL or M for FIXED BINARY.

If x and y are fixed-point with different scaling factors, the argument with the smaller scaling factor is converted to the larger scaling factor before R is calculated. If the conversion fails, the result is unpredictable.

The following example contrasts the MOD and REM built-in functions.

For information on the REM built-in function, see "REM" on page 464.

Example

```
rem( +10, +8 ) = 2
mod( +10, +8 ) = 2

rem( +10, -8 ) = 2
mod( +10, -8 ) = 2
```

MOD

```
rem( -10, +8 ) = -2
mod( -10, +8 ) = 6

rem( -10, -8 ) = -2
mod( -10, -8 ) = 6
```

MPSTR

MPSTR truncates a string at a logical boundary and returns a mixed character string. It does not truncate a double-byte character between bytes. The length of the returned string is equal to the length of the expression *x*, or to the value specified by *y*. The processing of the string is determined by the rules selected by the expression *r*, as described below.

```
►►──MPSTR(x,r─┬────────┬──)──────────────────────────────►◄
              └──,y──┘
```
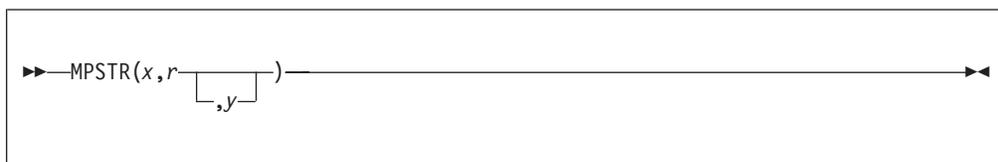
x   Expression that yields the character string result. The value of *x* is converted to character if necessary.

r   Expression that yields a character result. The expression cannot be GRAPHIC and is converted to character if necessary.

The expression *r* specifies the rules to be used for processing the string. The characters that can be used in *r* and the rules for them are as follows:

V or v
   Validates the mixed string *x* and returns a mixed string.

S or s
   Removes any null DBCS strings, creates a new string, and returns a mixed string.

If both V and S are specified, V takes precedence over S, regardless of the order in which they were specified.

If S is specified without V, the string x is assumed to be a valid string. If the string is not valid, undefined results occur.

Note: The parameter *r* is ignored on Intel and AIX.

y   Expression. If necessary, *y* is converted to a real fixed-point binary value. If *y* is omitted, the length is determined by the rules for type conversion. The value of *y* cannot be negative. If *y* = 0, the result is the null character string. If *y* is greater than the length needed to contain *x*, the result is padded with blanks. If *y* is less than the length needed to contain *x*, the result is truncated by discarding excess characters from the right (if they are SBCS characters), or by discarding as many DBCS characters (2-byte pairs) as needed.

MULTIPLY

MULTIPLY returns the product of *x* and *y*, with a precision specified by *p* and *q*. The base, scale, and mode of the result are determined by the rules for expression evaluation.

```
►►──MULTIPLY(x,y,p───────)─────────────────────────────────────────────►◄
                    └─,q─┘
```

x and y
> Expressions.

p Restricted expression that specifies the number of digits to be maintained throughout the operation.

q Restricted expression that specifies the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is assumed. For a floating-point result, *q* must be omitted.

NULL

NULL returns the null pointer value. The null pointer value does not identify any generation of a variable. The null pointer value can be assigned to and compared with handles. The null pointer value can be converted to OFFSET by assignment of the built-in function value to an offset variable.

```
►►──NULL───────────────────────────────────────────────────────────────►◄
        └─()─┘
```

OFFSET

OFFSET returns an offset value derived from a pointer reference *x* and relative to an area *y*. If x is the null pointer value, the null offset value is returned.

```
►►──OFFSET──(──x──,──y──)───────────────────────────────────────────────►◄
```

x Pointer reference. It must identify a generation of a based variable within the area *y*, or be the null pointer value.

y Area reference.

If *x* is an element reference, *y* must be an element variable.

OFFSETADD

OFFSETADD returns the sum of the arguments.

```
►►──OFFSETADD(x,y)──────────────────────────────────────────────────────►◄
```

x Expression. *x* must be specified as OFFSET.

> **y** Expression. *y* must have a computational type and is converted to FIXED
> BINARY(31,0).

OFFSETDIFF

OFFSETDIFF returns a FIXED BINARY(31,0) value that is the arithmetic difference
between the arguments.

```
►►──OFFSETDIFF(x,y)──────────────────────────────────────►◄
```
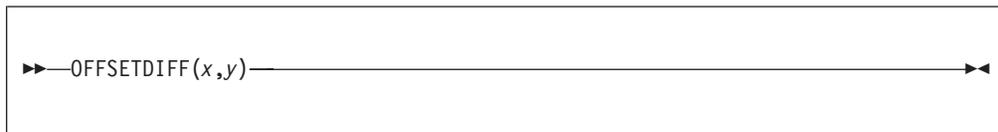
x and y
   Expressions. Both must be specified as OFFSET.

OFFSETSUBTRACT

OFFSETSUBTRACT is equivalent to OFFSETADD(x,-y).

```
►►──OFFSETSUBTRACT(x,y)──────────────────────────────────►◄
```
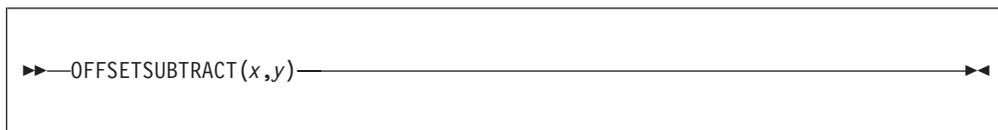
> **x** Expressions. *x* must be specified as OFFSET.

> **y** Expression. *y* must have a computational type and is converted to FIXED
> BINARY(31,0).

OFFSETVALUE

OFFSETVALUE returns an offset value that is the converted value of *x*.

```
►►──OFFSETVALUE(x)───────────────────────────────────────►◄
```

> **x** Expression. *x* must have a computational type and is converted to FIXED
> BINARY(31,0).

OMITTED

OMITTED returns a BIT(1) value that is '1'B if the parameter named *x* was omitted
in the invocation to its containing procedure.

```
►►──OMITTED(x)───────────────────────────────────────────►◄
```
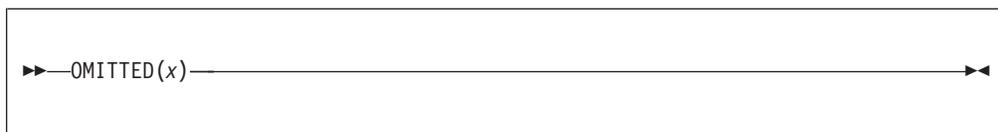
> **x** Level-1 unsubscripted parameter with the BYADDR attribute.

ONCHAR

ONCHAR returns a character(1) string containing the character that caused the CONVERSION condition to be raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for the CONVERSION condition.

```
►►──ONCHAR──────────────────────────────────────────────►◄
           └─()─┘
```
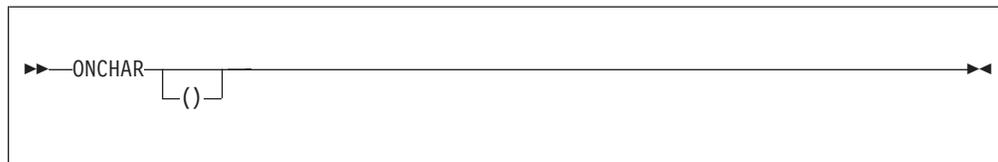
If the ONCHAR built-in function is used out of context, a blank is returned.

ONCHAR pseudovariable

The ONCHAR pseudovariable sets the current value of the ONCHAR built-in function. The element value assigned to the pseudovariable is converted to a character value of length 1. The new character is used when the conversion is attempted again. (See conversions in Chapter 5, "Data conversion," on page 73.)

```
►►──ONCHAR──────────────────────────────────────────────►◄
           └─()─┘
```
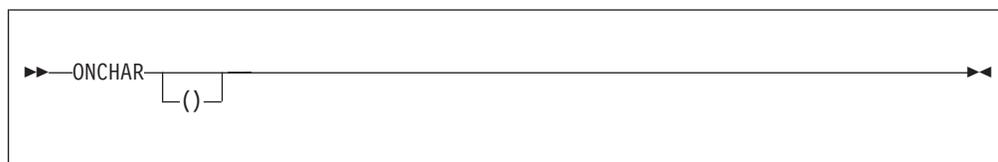
The pseudovariable must not be used out of context.

ONCODE

The ONCODE built-in function provides a fixed-point binary value that depends on the cause of the last condition. ONCODE can be used to distinguish between the various circumstances that raise a particular condition—for instance, the ERROR condition. For codes corresponding to the conditions and errors detected, refer to the specific condition.

ONCODE returns a real fixed-point binary value that is the condition code. It is in context in any ON-unit or its dynamic descendant. All condition codes are defined in *Messages and Codes*.

```
►►──ONCODE──────────────────────────────────────────────►◄
           └─()─┘
```
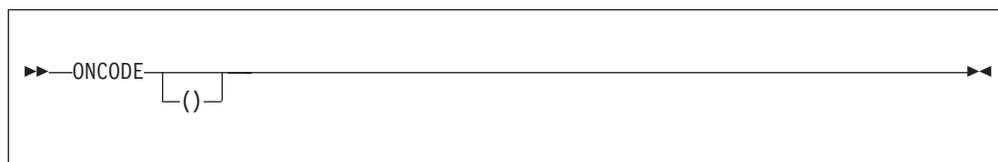
If ONCODE is used out of context, zero is returned.

ONCONDCOND

ONCONDCOND returns a nonvarying character string whose value is the name of the condition for which a CONDITION condition is raised. If the name is a DBCS name, it will be returned as a mixed character string. It is in context in the following circumstances:

- In a CONDITION ON-unit, or any of its dynamic descendants
- In an ANYCONDITION ON-unit that traps a CONDITION condition, or any dynamic descendants of such an ON-unit.

```
►►──ONCONDCOND──────────────────────────────►◄
              └─()─┘
```
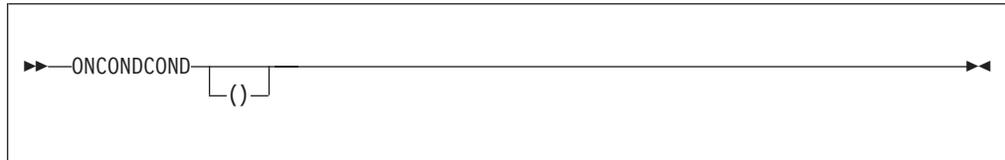
If ONCONDCOND is used out of context, a null string is returned.

ONCONDID

ONCONDID (short for ON-condition identifier) returns a FIXED BINARY(31,0) value that identifies the condition being handled by an ON-unit. It is in context in any ON-unit or one of its dynamic descendants.

```
►►──ONCONDID──────────────────────────────────►◄
           └─()─┘
```
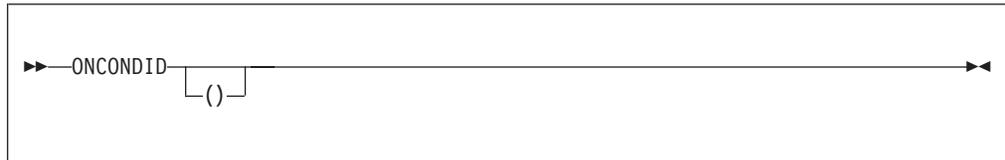
The values returned by ONCONDID are given in the following DECLARE statement:

```
 declare (   condid_area               value(1),
             condid_attention          value(2),
             condid_condition          value(3),
             condid_conversion         value(4),
             condid_endfile            value(5),
             condid_endpage            value(6),
             condid_error              value(7),
             condid_finish             value(8),
             condid_fixedoverflow      value(9),
             condid_invalidop          value(10),
             condid_key                value(11),
             condid_name               value(12),
             condid_overflow           value(13),
             condid_record             value(14),
             condid_size               value(15),
             condid_storage            value(16),
             condid_stringrange        value(17),
             condid_stringsize         value(18),
             condid_subscriptrange     value(19),
             condid_transmit           value(20),
             condid_undefinedfile      value(21),
             condid_underflow          value(22),
             condid_zerodivide         value(23)
          ) fixed bin(31);
```

If ONCONDID is used out of context, a value of zero is returned.

ONCOUNT

ONCOUNT returns an unscaled REAL FIXED BINARY value specifying the number of conditions that remain to be handled when an ON-unit is entered. (See "Multiple conditions" on page 338.) It is in context in any ON-unit, or any dynamic descendant of an ON-unit.

```
►►──ONCOUNT───────────────────────────────────►◄
            └─()─┘
```
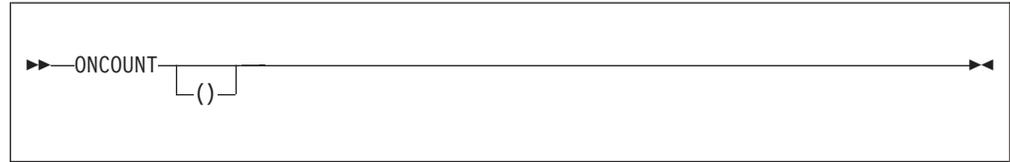
If ONCOUNT is used out of context, zero is returned.

The BIFPREC compiler option determines the precision of the result returned.

ONFILE

ONFILE returns a character string whose value is the name of the file for which an input or output condition is raised. If the name is a DBCS name, it is returned as a mixed character string. It is in context in an ON-unit (or any of its dynamic descendants) for an input or output condition, or for the ERROR or FINISH condition raised as the implicit action for an input or output condition.

```
►►──ONFILE──────────────────────────────────►◄
          └─()─┘
```
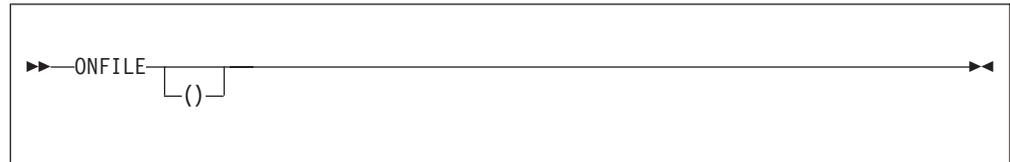
If ONFILE is used out of context, a null string is returned.

ONGSOURCE

ONGSOURCE returns a graphic string containing the DBCS character that caused the CONVERSION condition to be raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for a CONVERSION condition.

```
►►──ONGSOURCE────────────────────────────────►◄
              └─()─┘
```
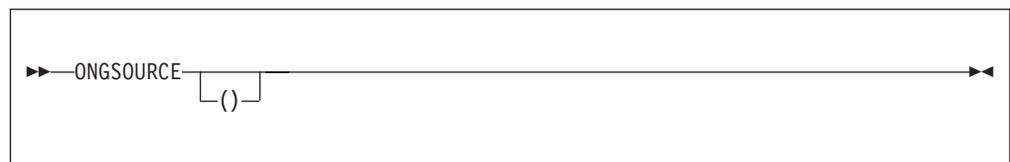
If the ONGSOURCE built-in function is used out of context, a null GRAPHIC string is returned.

ONGSOURCE pseudovariable

The ONGSOURCE pseudovariable sets the current value of the ONGSOURCE built-in function. The element value assigned to the pseudovariable is converted graphic. The string is used when the conversion is attempted again.

```
►►──ONGSOURCE──────────────────────────────────────►◄
             └─()─┘
```
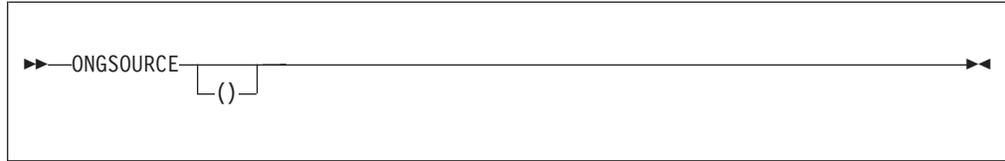
The pseudovariable must not be used out of context.

ONKEY

ONKEY returns a character string whose value is the key of the record that raised an input/output condition. For indexed files, if the key is GRAPHIC, the string is returned as a mixed character string. ONKEY is in context for the following:

- An ON-unit, or any of its dynamic descendants
- Any input/output condition, except ENDFILE
- The ERROR or FINISH condition raised as implicit action for an input/output condition.

ONKEY is always set for operations on a KEYED file, even if the statement that raised the condition does not specified the KEY, KEYTO, or KEYFROM options.

```
►►──ONKEY──────────────────────────────────────────►◄
        └─()─┘
```
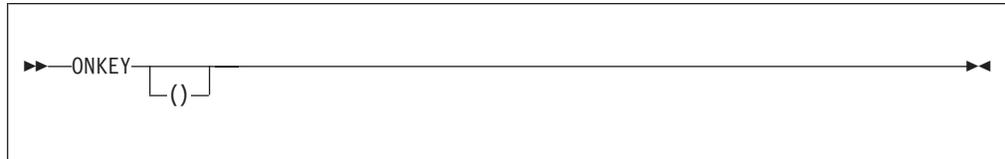
The result of specifying ONKEY is:

- For any input/output condition (other than ENDFILE), or for the ERROR or FINISH condition raised as implicit action for these conditions, the result is the value of the recorded key from the I/O statement causing the error.
- For relative data sets, the result is a character string representation of the relative record number. If the key was incorrectly specified, the result is the last 8 characters of the source key. If the source key is less than 8 characters, it is padded on the right with blanks to make it 8 characters. If the key was correctly specified, the character string consists of the relative record number in character form padded on the left with blanks, if necessary.
- For a REWRITE statement that attempts to write an updated record on to an indexed data set when the key of the updated record differs from that of the input record, the result is the value of the embedded key of the input record.

If ONKEY is used out of context, a null string is returned.

ONLOC

ONLOC returns a character string whose value is the name of the entry-point used for the current invocation of the procedure in which a condition was raised.

ONLOC always returns the leftmost name of a multiple label specification, regardless of which name appears in the CALL or GOTO statement.

If the name is a DBCS name, it is returned as a mixed-character string. It is in context in any ON-unit, or in any of its dynamic descendants.

```
►►──ONLOC──────────────────────────────────────────────────►◄
         └─()─┘
```
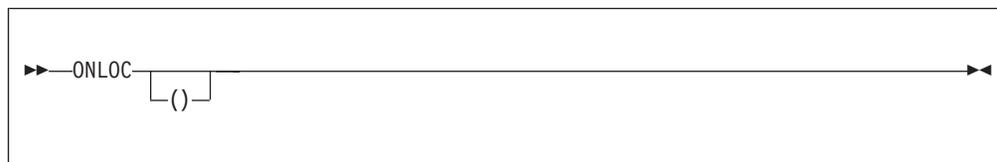
If ONLOC is used out of context, a null string is returned.

ONSOURCE

ONSOURCE returns a character string whose value is the contents of the field that was being processed when the CONVERSION condition was raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for a CONVERSION condition.

```
►►──ONSOURCE────────────────────────────────────────────────►◄
           └─()─┘
```
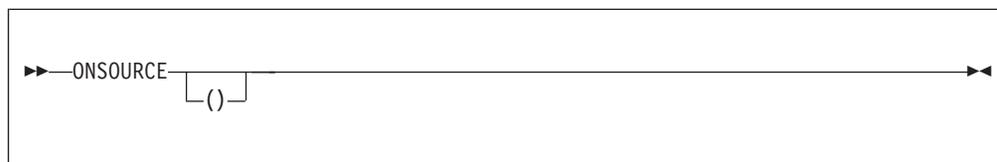
If ONSOURCE is used out of context, a null string is returned.

ONSOURCE pseudovariable

The ONSOURCE pseudovariable sets the current value of the ONSOURCE built-in function. The element value assigned to the pseudovariable is converted to a character string and, if necessary, is padded on the right with blanks or truncated to match the length of the field that raised the CONVERSION condition. The string is used when the conversion is attempted again.

```
►►──ONSOURCE────────────────────────────────────────────────►◄
           └─()─┘
```

When conversion is retried, the string assigned to the pseudovariable is processed as a single data item. For this reason, the error correction process must not assign a string containing more than one data item when the conversion occurs during the execution of a GET LIST or GET DATA statement. The presence of blanks or commas in the string could raise CONVERSION again.

The pseudovariable must not be used out of context.

ONSUBCODE

ONSUBCODE returns a FIXED BINARY(31,0) value that gives more information about an I/O error that occurred. This corresponds to the SUBCODE1 values documented for messages IBM0236I and IBM0265I. These values are defined in *Messages and Codes*.

```
►►──ONSUBCODE()──────────────────────────────────────────────►◄
```
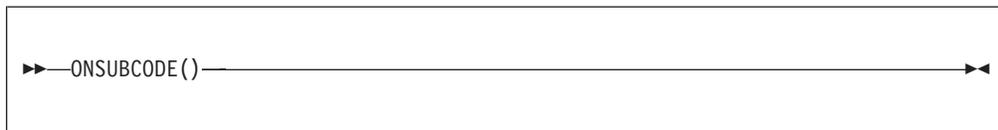
ONWCHAR

ONWCHAR returns a widechar(1) string containing the widechar that caused the CONVERSION condition to be raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for the CONVERSION condition.

```
►►──ONWCHAR──────────────────────────────────────────────────►◄
            └─()─┘
```
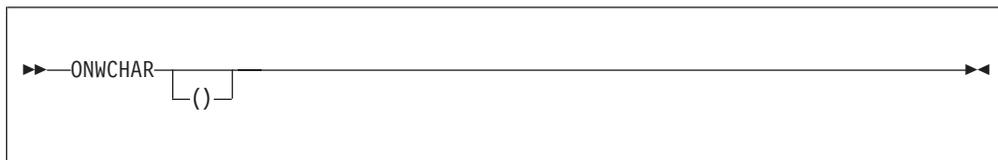
If the ONWCHAR built-in function is used out of context, a widechar blank is returned.

ONWCHAR pseudovariable

The ONWCHAR pseudovariable sets the current value of the ONWCHAR built-in function. The element value assigned to the pseudovariable is converted to a widechar value of length 1. The new widechar is used when the conversion is attempted again. (See conversions in Chapter 5, "Data conversion," on page 73.)

```
►►──ONWCHAR──────────────────────────────────────────────────►◄
            └─()─┘
```
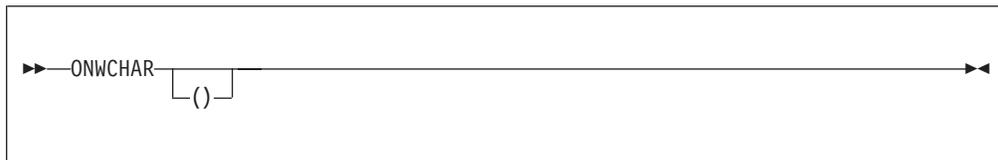
The pseudovariable must not be used out of context.

ONWSOURCE

ONWSOURCE returns a widechar string whose value is the contents of the field that was being processed when the CONVERSION condition was raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for a CONVERSION condition.

```
►►──ONWSOURCE──────────────────────────────────────────────────►◄
             └─()─┘
```

If ONWSOURCE is used out of context, a null string is returned.

ONWSOURCE pseudovariable

The ONWSOURCE pseudovariable sets the current value of the ONWSOURCE built-in function. The element value assigned to the pseudovariable is converted to a widechar string and, if necessary, is padded on the right with widecahr blanks or truncated to match the length of the field that raised the CONVERSION condition. The string is used when the conversion is attempted again.

```
►►──ONWSOURCE──────────────────────────────────────────────────►◄
             └─()─┘
```

When conversion is retried, the string assigned to the pseudovariable is processed as a single data item. For this reason, the error correction process must not assign a string containing more than one data item when the conversion occurs during the execution of a GET LIST or GET DATA statement. The presence of blanks or commas in the string could raise CONVERSION again.

The pseudovariable must not be used out of context.

ORDINALNAME

ORDINALNAME returns a nonvarying character string that is the member of the set associated with the ordinal *x*.

```
►►──ORDINALNAME(x)──────────────────────────────────────────────►◄
```

x   Reference. It must have ordinal type.

ORDINALs cannot be used in computational expressions and cannot be converted to character, but ORDINALNAME provides a way to obtain a displayable value for an ORDINAL and can be very useful in debugging.

ORDINALPRED

ORDINALPRED returns an ordinal that is the next lower value that the ordinal *x* could assume.

```
►►──ORDINALPRED(x)──────────────────────────────────────────►◄
```
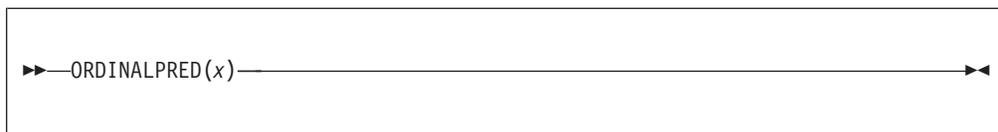
x   Reference. It must have ordinal type.

The returned ordinal has the same type as ordinal *x*.

ORDINALSUCC

ORDINALSUCC returns an ordinal that is the next higher value the ordinal *x* could assume.

```
►►──ORDINALSUCC(x)──────────────────────────────────────────►◄
```
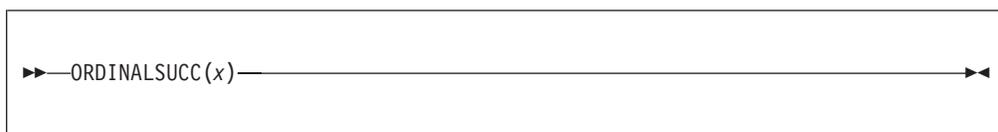
x   Reference. It must have ordinal type.

The returned ordinal has the same type as ordinal *x*.

PACKAGENAME

PACKAGENAME returns a nonvarying character string containing the name of the package in which it is invoked. If there is no package in the current compilation unit, PACKAGENAME returns the name of the outermost procedure.

```
►►──PACKAGENAME──────────────────────────────────────────────►◄
              └─()─┘
```
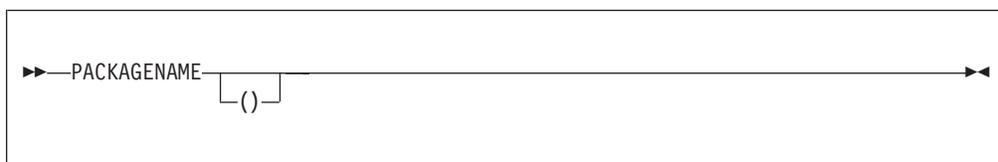
PAGENO

PAGENO returns an unscaled REAL FIXED BINARY31 value that is the current page number associated with file *x*.

```
►►──PAGENO(x)────────────────────────────────────────────────►◄
```
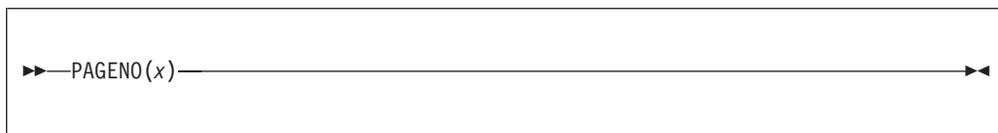
x   An open PRINT file.

If the file is not a PRINT file, the ERROR condition is raised.

The BIFPREC compiler option determines the precision of the result returned.

PLACES

PLACES returns a FIXED BINARY(31,0) value that is the model-precision used to represent the floating-point expression *x*.

```
►►──PLACES(x)──────────────────────────────────────►◄
```
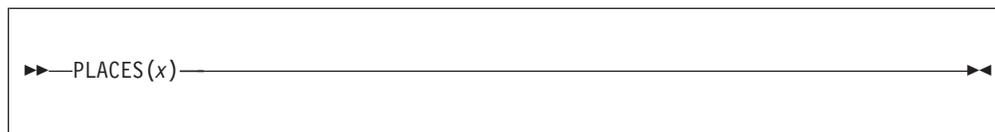
x   Expression. *x* must be declared REAL FLOAT.

PLACES(x) is a constant and can be used in restricted expressions.

Example (Intel Values)

```
places(x) = 24        for x float bin(p), p <= 21
places(x) = 53        for x float bin(p), 21 < p <= 53
places(x) = 64        for x float bin(p), 53 < p

places(x) = 24        for x float dec(p), p <= 6
places(x) = 53        for x float dec(p), 6 < p <= 16
places(x) = 64        for x float dec(p), 16 < p
```

Example (AIX Values)

```
places(x) = 024        for x float bin(p), p <= 21
places(x) = 053        for x float bin(p), 21 < p <= 53
places(x) = 106        for x float bin(p), 53 < p

places(x) = 024        for x float dec(p), p <= 6
places(x) = 053        for x float dec(p), 6 < p <= 16
places(x) = 106        for x float dec(p), 16 < p
```

Example (z/OS Hexadecimal Values)

```
places(x) = 6         for x float bin(p), p <= 21
places(x) = 14        for x float bin(p), 21 < p <= 53
places(x) = 28        for x float bin(p), 53 < p

places(x) = 6         for x float dec(p), p <= 6
places(x) = 14        for x float dec(p), 6 < p <= 16
places(x) = 28        for x float dec(p), 16 < p
```

Example (z/OS IEEE Values)

```
places(x) = 24        for x float bin(p), p <= 21
places(x) = 53        for x float bin(p), 21 < p <= 53
places(x) = 113       for x float bin(p), 53 < p

places(x) = 24        for x float dec(p), p <= 6
places(x) = 53        for x float dec(p), 6 < p <= 16
places(x) = 113       for x float dec(p), 16 < p
```

PLIASCII

PLIASCII converts Z storage units (bytes) at location y from EBCDIC to ASCII at location x. The storage at location x and y must not overlap unless they specify the same location.

```
►►──PLIASCII(x,y,z)──────────────────────────────────────────────►◄
```

x and y
> Expressions with type POINTER or OFFSET. If the type is OFFSET, the expression must be an OFFSET variable declared with the AREA attribute.

z   Expression with computational type that is converted to FIXED BIN(31,0).

PLICANC

This built-in subroutine allows you to cancel the automatic restart facility.

```
►►──PLICANC──────────────────────────────────────────────────────►◄
            └──()──┘
```

For more information about using PLICANC, see the Programming Guide.

PLICKPT

This built-in subroutine allows you to take a checkpoint for later restart.

```
►►──PLICKPT(argument───────────)──────────────────────────────────►◄
                    └──,argument──┘
```

For more information about using PLICKPT, see the Programming Guide.

PLIDELETE

This built-in subroutine frees the storage associated with the handle *x*.

```
►►──PLIDELETE(x)──────────────────────────────────────────────────►◄
```

x   Handle expression.

PLIDELETE(x) is the best way to free the storage associated with a handle; this storage is usually acquired by the NEW type function.

CALL PLIDELETE(x) is equivalent to CALL PLIFREE(PTRVALUE(x)).

PLIDUMP

This built-in subroutine allows you to obtain a formatted dump of selected parts of storage used by your program.

```
►►──PLIDUMP(argument─────────────)───────────────────────────►◄
              └─,argument─┘
```
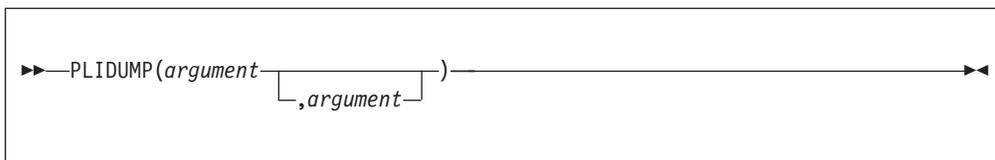
For more information about using PLIDUMP, see the Programming Guide.

PLIEBCDIC

PLIEBCDIC converts Z storage units (bytes) at location y from ASCII to EBCDIC at location x. The storage at location x and y must not overlap unless they specify the same location.

```
►►──PLIEBCDIC(x,y,z)─────────────────────────────────────────►◄
```
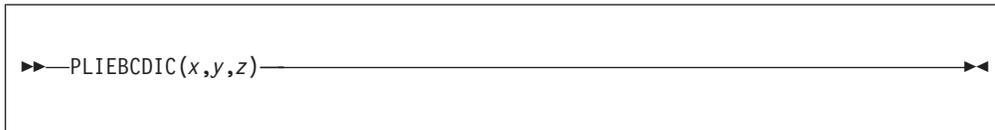
x and y
Expressions with type POINTER or OFFSET. If the type is OFFSET, the expression must be an OFFSET variable declared with the AREA attribute.

z   Expression with computational type that is converted to FIXED BIN(31,0).

PLIFILL

This built-in subroutine moves *z* copies of the byte *y* to the location *x* without any conversions, padding, or truncation.

```
►►──PLIFILL(x,y,z)───────────────────────────────────────────►◄
```
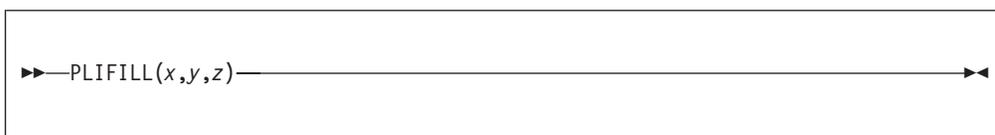
x   Expression. *x* must be declared POINTER or OFFSET. If it is OFFSET, x must be declared with the AREA attribute.

y   Must be declared CHARACTER(1) NONVARYING.

z   Expression that is converted to FIXED BINARY(31,0).

Example

```
dcl 1 Str1,
      2 B  fixed bin(31),
      2 C  pointer,
      2 * union,
        3 D  char(4),
        3 E  fixed bin(31),
        3 *,
          4 * char(3),
```

```
            4 F fixed bin(8) unsigned,
        2 * char(0)
             initial call plifill( addr(Str1), '00'x, stg(Str1) );
```

PLIFREE

This built-in subroutine frees the heap storage associated with the pointer *p* that was allocated using the ALLOCATE built-in function.

```
►►─PLIFREE(p)──────────────────────────────────────────►◄
```

p   Locator expression.

PLIFREE is the opposite of ALLOCATE (ALLOC).

PLIMOVE

This built-in subroutine moves *z* storage units (bytes) from location *y* to location *x*, without any conversions, padding, or truncation. Unlike the PLIOVER built-in subroutine, storage at locations *x* and *y* is assumed to be unique. If storage overlaps, unpredictable results can occur.
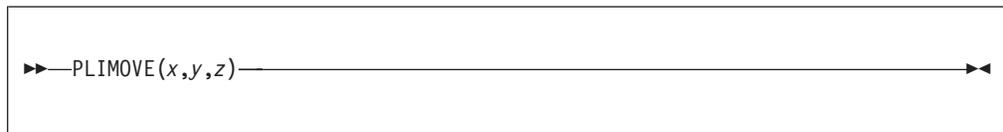
```
►►─PLIMOVE(x,y,z)──────────────────────────────────────►◄
```

x and y
   Expressions declared as POINTER or OFFSET. If the type is OFFSET, *x* or *y* must be declared with the AREA attribute.

z   Expression. *z* must have a computational type and is converted to FIXED BINARY(31,0).

Example

```
dcl 1 Str1,
      2 B  fixed bin(31),
      2 C  pointer,
      2 * union,
        3 D  char(4),
        3 E  fixed bin(31),
        3 *,
          4 * char(3),
          4 F fixed bin(8) unsigned,
      2 * char(0);
dcl 1 Template nonasgn static,
      2 * fixed bin(31) init(200),
      2 * pointer init(null()),
      2 * char(4) init(''),
      2 * char(0);

call plimove(addr(Str1), addr(Template), stg(Str1));
```

PLIOVER

This built-in subroutine moves *z* storage units (bytes) from location *y* to location *x*, without any conversions, padding, or truncation. Unlike the PLIMOVE built-in subroutine, the storage at locations *x* and *y* can overlap.
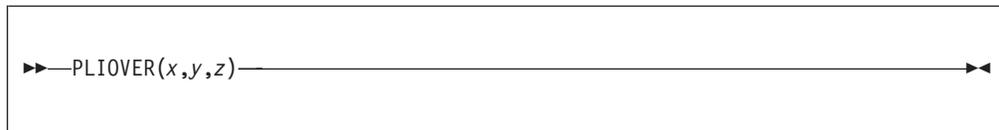
```
►►──PLIOVER(x,y,z)──────────────────────────────────────────►◄
```

x and y
Expressions declared as POINTER or OFFSET. If the type is OFFSET, *x* or *y* must be declared with the AREA attribute.

z Expression. *z* must have a computational type and is converted to FIXED BINARY(31,0).

Usage of PLIOVER is the same as PLIMOVE, with the exception that storage for *x* and *y* can overlap (see "PLIMOVE" on page 454).

PLIREST

This built-in subroutine allows you to restart program execution.

```
►►──PLIREST─────────────────────────────────────────────────►◄
          └─()─┘
```
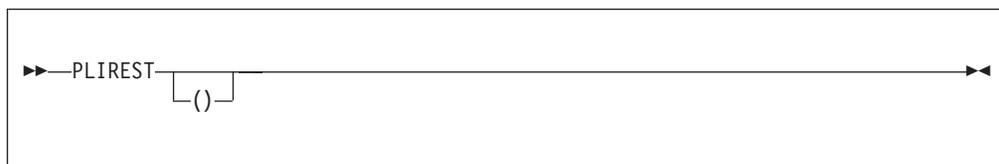
For more information about using PLIREST, see the Programming Guide.

PLIRETC

This built-in subroutine allows you to set a return code that can be examined by the program that invoked this PL/I program or by another PL/I procedure via the PLIRETV built-in function.

```
►►──PLIRETC(x)──────────────────────────────────────────────►◄
```
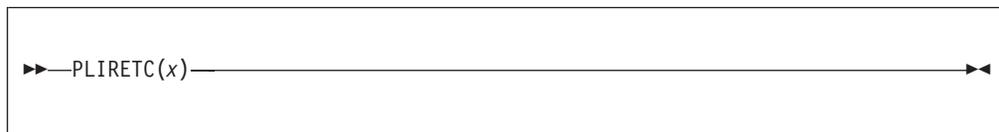
x An expression yielding a FIXED BINARY(31,0) return code.

PLIRETV

PLIRETV returns a FIXED BINARY(31,0) value that is the PL/I return code.

```
►►──PLIRETV──────────────►◄
         └─()─┘
```
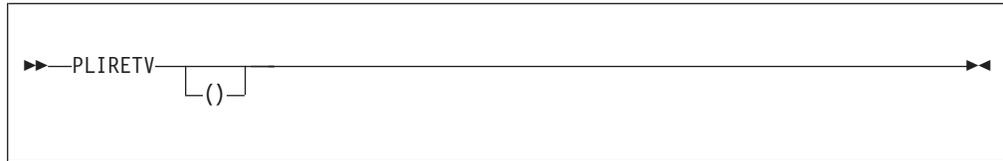
The value of the PL/I return code is the most recent value specified by a CALL
PLIRETC statement.

PLISAXA

This built-in subroutine allows you to perform SAX-style parsing of an XML
document residing in a buffer in your program.

```
►►──PLISAXA(e,p,x,n──────)──────────────────────►◄
                 └─,c─┘
```
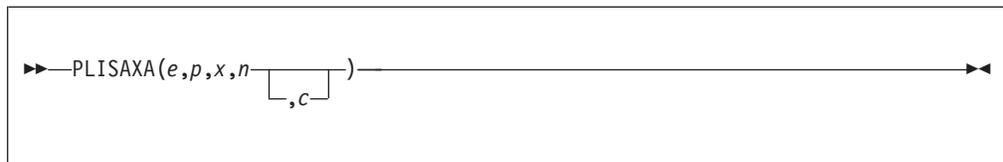
e   An event structure

p   A pointer value or ″token″ that will be passed back to the parsing events

x   The address of the buffer containing the input XML

n   The number of bytes of data in that buffer

c   A numeric expression specifying the purported codepage of that XML

Note that if the XML is contained in a CHARACTER VARYING or a WIDECHAR
VARYING string, then the ADDRDATA built-in function should be used to obtain the
address of the first data byte.

Also note that if the XML is contained in a WIDECHAR string, the value for the
number of bytes is twice the value returned by the LENGTH built-in function.

For more information, see the Programming Guide.

PLISAXB

This built-in subroutine allows you to perform SAX-style parsing of an XML
document residing in a file.

```
►►──PLISAXB(e,p,x──────)──────────────────────►◄
                └─,c─┘
```
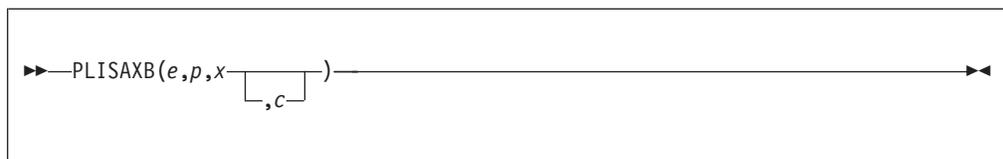
e   An event structure

p   A pointer value or ″token″ that will be passed back to the parsing events

> **x** A character string expression specifying the input file
>
> **c** A numeric expression specifying the purported codepage of that XML
>
> For more information, see the Programming Guide.

PLISRTA

This built-in subroutine allows you to sort an input file to produce a sorted output file.

```
                     ┌─,◄─────┐
►►──PLISRTA(───▼─argument──┴──)──────────────────────►◄
```
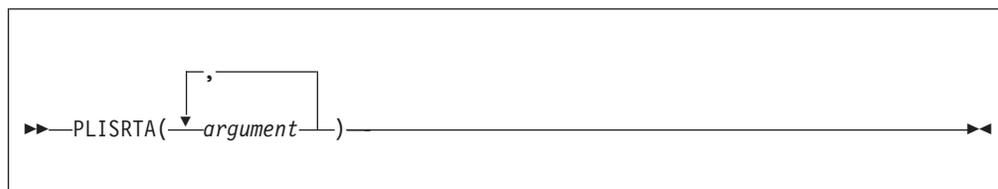
For more information, see the Programming Guide.

PLISRTB

This built-in subroutine allows you to sort input records provided by an E15 PL/I exit procedure to produce a sorted output file.

```
                     ┌─,◄─────┐
►►──PLISRTB(───▼─argument──┴──)──────────────────────►◄
```
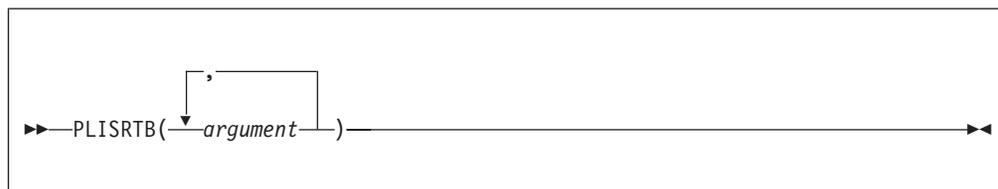
For more information, see the Programming Guide.

PLISRTC

This built-in subroutine allows you to sort an input file to produce sorted records that are processed by an E35 PL/I exit procedure.

```
                     ┌─,◄─────┐
►►──PLISRTC(───▼─argument──┴──)──────────────────────►◄
```
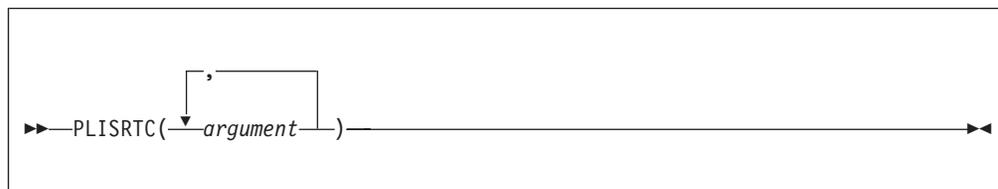
For more information, see the Programming Guide.

PLISRTD

This built-in subroutine allows you to sort input records provided by an E15 PL/I exit procedure to produce sorted records that are processed by an E35 PL/I exit procedure.

```
         ┌─────,─────┐
►►──PLISRTD(──▼─argument──┘──)──────────────────────────►◄
```

For more information, see the Programming Guide.

POINTER

POINTER returns a pointer value that identifies the generation specified by an offset reference *x*, in an area specified by *y*. If *x* is the null offset value, the null pointer value is returned.

```
►►──POINTER(x,y)────────────────────────────────────────►◄
```
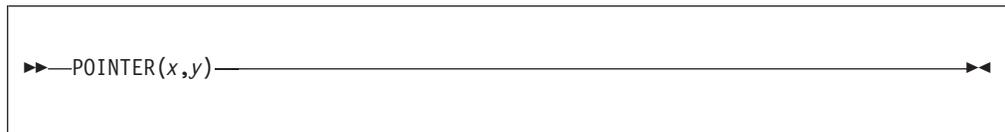
Abbreviation: PTR

x   Offset reference. It can be the null offset value. If it is not, *x* must identify a generation of a based variable, but not necessarily in *y*. If it is not in *y*, the generation must be equivalent to a generation in *y*.

y   Area reference.

Generations of based variables in different areas are equivalent if, up to the allocation of the latest generation, the variables have been allocated and freed the same number of times as each other.

POINTERADD

POINTERADD returns a pointer value that is the sum of its arguments.

```
►►──POINTERADD(x,y)─────────────────────────────────────►◄
```
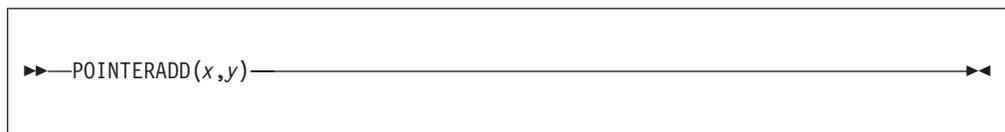
Abbreviation: PTRADD

x   Pointer expression.

y   Expression that must have a computational type and is converted to FIXED BINARY(31,0).

POINTERADD can be used as a locator for a based variable.

POINTERADD can be used for subtraction by prefixing the operand to be subtracted with a minus sign.

There is no need to use POINTERADD to increment a pointer - you can simply increment the pointer as you would an integer. For example, there is no need to write:

```
     p = pointeradd(p,2);
```

Instead, you could write either of the following equivalent statements:

```
p = p + 2;
p += 2;
```

However, POINTERADD can be useful in dereferencing the storage at a location offset from a pointer, as in the following example:

```
dcl x fixed bin(31), b based fixed bin(31);
x = pointeradd(p,2)->b;
```

Note, however, since a locator in PL/I must be a reference, you cannot write

```
x = (p + 2)->b;
```

POINTERDIFF

POINTERDIFF returns a FIXED BINARY(31,0) result that is the difference between the two pointers *x* and *y*.

```
►►──POINTERDIFF(x,y)──────────────────────────────────────►◄
```
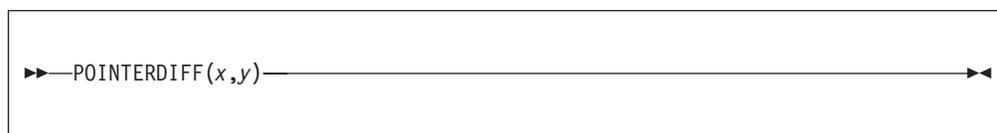
Abbreviation: PTRDIFF

x and y
Expressions declared as POINTER.

POINTERSUBTRACT

POINTERSUBTRACT is equivalent to POINTERADD(x,-y).

```
►►──POINTERSUBTRACT(x,y)──────────────────────────────────►◄
```
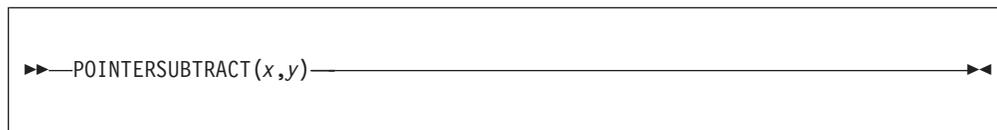
Abbreviation: PTRSUBTRACT

x   Must be a pointer expression.

y   Expression that must have a computational type and is converted to FIXED BINARY(31,0).

POINTERVALUE

POINTERVALUE returns a pointer value that is the converted value of *x*.

```
►►──POINTERVALUE(x)───────────────────────────────────────►◄
```
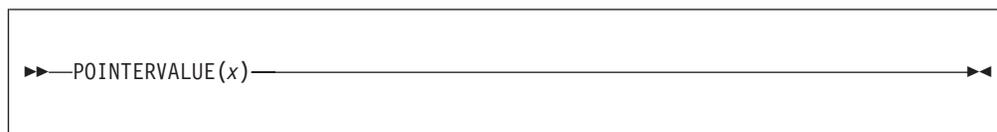
Abbreviation: PTRVALUE

> **x** Expression that must have either the HANDLE attribute, or have a computational type. If *x* has a computational type, it is converted to FIXED BINARY(31,0).
>
> POINTERVALUE(x) can be used to initialize static pointer variables if *x* is a constant.

POLY

POLY returns a floating-point value that is an approximation of a polynomial formed from an one-dimensional array expressions x. The returned value has the same attributes as the first argument. The syntax for POLY is:

```
►►──POLY──(──x──,──y──)──────────────────────────────────►◄
```
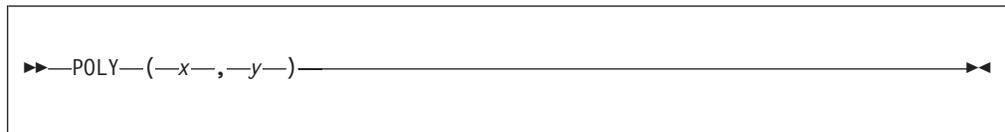
> **x** An array expression.
>
> **y** An element expression.

x must be REAL FLOAT and **y** is converted to the attributes of x, if necessary.

If x has lower bound 0 and upper bound n, the result is a classic polynomial of degree n in y with coefficients given by x, i.e. the result is

```
x(0) + x(1)*y + x(2)*y**2 + ... + x(n)*y**n
```

In the general case, where x has lower bound m and upper bound n, the result is the polynomial

```
x(m) + x(m+1)*y + x(m+2)*y**2 + ... + x(n)*y**(n-m)
```

PRECISION

PRECISION returns the value of *x*, with a precision specified by *p* and *q*. The base, mode, and scale of the returned value are the same as that of *x*.

```
►►──PRECISION(x,p────────)──────────────────────────────►◄
              └─,q─┘
```
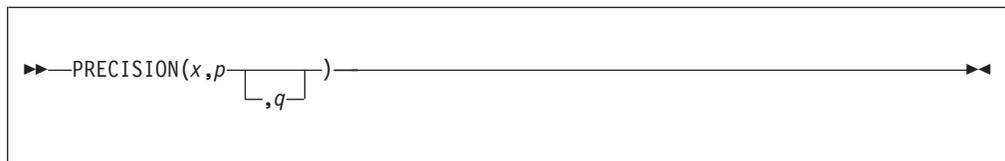
Abbreviation: PREC

> **x** Expression.
>
> **p** Restricted expression. *p* specifies the number of digits that the value of the expression *x* is to have after conversion.
>
> **q** Restricted expression. It specifies the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is assumed. For a floating-point result, *q* must be omitted.

PRED

PRED returns a floating-point value that is the biggest representable number smaller than *x*. It has the base, mode, and precision of x. OVERFLOW will be raised if there is no such number.

```
►►──PRED(x)──────────────────────────────────────────────►◄
```
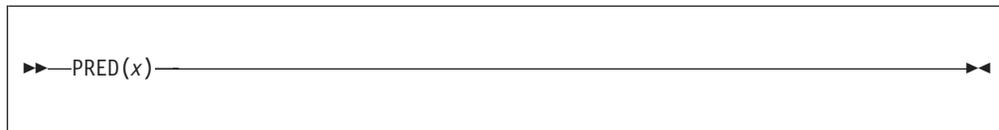
x   REAL FLOAT expression.

PRED(TINY(X)) will return zero and will not raise UNDERFLOW.

PRESENT

PRESENT(x) returns a BIT(1) value that is '1'B if the parameter *x* was present in the invocation of its containing procedure.

```
►►──PRESENT(x)───────────────────────────────────────────►◄
```
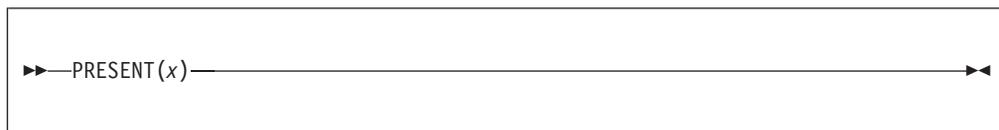
x   Level-1 unsubscripted BYADDR parameter.

PROCEDURENAME

PROCEDURENAME() returns a nonvarying character string containing the name of the procedure in which this built-in function is invoked.

```
►►──PROCEDURENAME──(──)──────────────────────────────────►◄
```
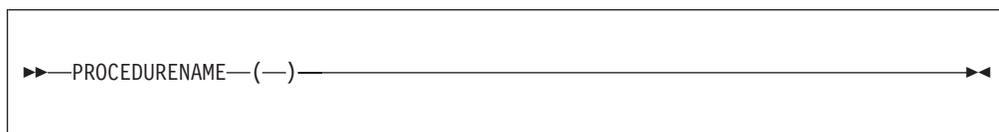
Abbreviation: PROCNAME

PROCEDURENAME always returns the leftmost name of a multiple label specification, regardless of which name appears in the CALL or GOTO statement.

PROD

PROD returns the product of all the elements in *x*.
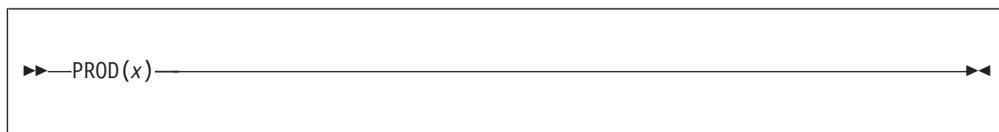
```
►►──PROD(x)──────────────────────────────────────────────►◄
```

x   Array expression. If the elements of *x* are strings, they are converted to fixed-point integer values.

If the elements of *x* are not fixed-point integer values or strings, they are converted to floating-point and the result is floating-point.

The result has the precision of *x*, except that the result for fixed-point integer values and strings is fixed-point with precision (n,0), where *n* is the maximum number of digits allowed. The base and mode match the converted argument *x*.

PUTENV

PUTENV works the same as the C putenv function. This function adds new environment variables or modifies the values of existing environment variables.

```
►►──PUTENV(string)──────────────────────────────────────────────►◄
```

string
    A character string of the form *envvarname=value*.

PUTENV returns true ('1'B) if successful and false ('0'B) otherwise.

RADIX

RADIX returns a FIXED BINARY(31,0) value that is the model-base used to represent the floating-point expression *x*.

```
►►──RADIX(x)─────────────────────────────────────────────────────►◄
```

x   REAL FLOAT expression.

RADIX(x) is 2 (except for hexadecimal on z/OS where it is 16) and can be used in restricted expressions.

RAISE2

RAISE2(x,n) returns the value x*(2**n)

```
►►──RAISE2(x,n)──────────────────────────────────────────────────►◄
```

Note: RAISE2(x,n) is equivalent to the assembler SLA(x,n).

x   Expression. *x* must have a computational type.

n   Expression. *n* must have a computational type.

x is converted to SIGNED REAL FIXED BIN(M,0) and the result has the same attributes.

It is undefined if *n* is negative or if *n* is greater than M.

Example

```
        raise2(6,1)                     /*  produces 12  */
```

RANDOM

RANDOM returns a FLOAT BINARY(53) random number generated using *x* as the given seed. If *x* is omitted, the random number generated is based on the seed provided by the last RANDOM invocation with a seed, or on a default initial seed of 1 if RANDOM has not previously been invoked with a seed.

```
►►──RANDOM──┬──────┬──────────────────────────────────────►◄
            └─(x)──┘
```
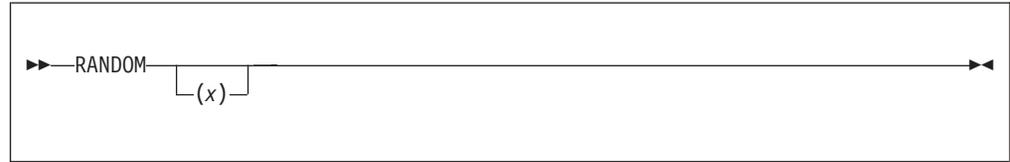
x Expression. *x* must have a computational type and should have an arithmetic type. If *x* is numeric, it must be real. If *x* is not specified FIXED BINARY(31,0), it is converted.

Unless 0 < *x* < 2,147,483,646, the ERROR condition is raised.

The values generated by RANDOM are uniformly distributed between 0 and 1, with 0 < random(x) < 1. They are generated as follows using the multiplicative congruential method:

```
seed(x) = mod(950706376 * seed(x - 1), 2147483647)
random(x) = seed(x) / 2147483647
```

The seed is maintained at the program level and not within each thread in a multithreading application.

RANK

RANK returns the integer value corresponding to a character or widechar.

```
►►──RANK(x)────────────────────────────────────────────────►◄
```
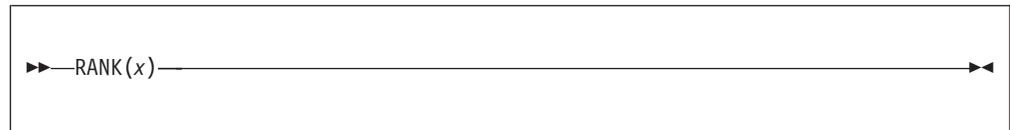
x Must have the attributes CHAR (1) NONVARYING or WCHAR (1) NONVARYING.

If x is character, RANK(x) is defined as index(collate(),x)-1, and Rank is the inverse of BYTE.

If x is widechar, RANK(x) is equal to UNSPEC(y) where y is x stored in bigendian format.

REAL

REAL returns the real part of *x*. The result has the base, scale, and precision of *x*.

```
►►──REAL(x)────────────────────────────────────────────────►◄
```
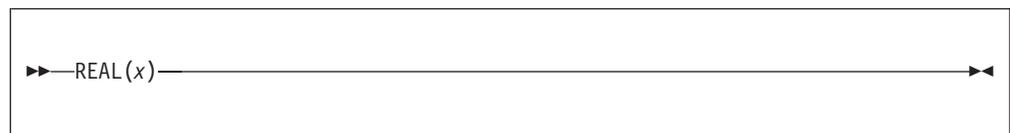
x Expression. If *x* is real, it is converted to complex.

REAL pseudovariable

The REAL pseudovariable assigns a real value or the real part of a complex value to the real part of *x*.

```
►►──REAL(x)──────────────────────────────────────────────►◄
```

x Complex reference.

REM

REM returns the remainder of *x* divided by *y*. This can be calculated by:

```
x - y * trunc(x/y)
```

```
►►──REM(x,y)──────────────────────────────────────────────►◄
```

x and y
Expressions. *x* and *y* must be computational and can be arithmetic.

For examples that contrast the REM and MOD built-in functions, refer to "MOD" on page 439.

REPATTERN

Takes a value holding a date in one pattern and returns that value converted to a date in a second pattern.

```
►►──REPATTERN(d─,p─,q──────┬─)──────────────────────────────►◄
                          └─,w─┘
```

d A string expression representing a date. The length of *d* must be at least as large as the length of the source pattern *q*. If *d* is larger, any excess characters must be formed by leading blanks.

   d must have a computational type and should have character type. If not, it is converted to character.

p The target pattern; must be one of the supported date/time patterns.

q The source pattern; must be one of the supported date/time patterns.

w Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The returned value has the attributes CHAR(m) NONVARYING where *m* is the length of the target patter *p*.

The allowed patterns are listed in Table 52 on page 378. For an explanation of Lilian format, see "Date/time built-in functions" on page 376.

REPATTERN('990101','YYYYMMDD','YYMMDD', 1950) returns '19990101'
REPATTERN('000101','YYYYMMDD','YYMMDD', 1950) returns '20000101'
REPATTERN('19990101','YYMMDD','YYYYMMDD', 1950) returns '990101'
REPATTERN('20000101','YYMMDD','YYYYMMDD', 1950) returns '000101'
REPATTERN('19490101','YYMMDD','YYYYMMDD', 1950) raises ERROR

REPEAT

REPEAT returns a bit, character, graphic or widechar string consisting of x concatenated to itself the number of times specified by y. That is, there are $(y + 1)$ occurrences of x.

```
►►──REPEAT(x,y)──────────────────────────────────────────────►◄
```
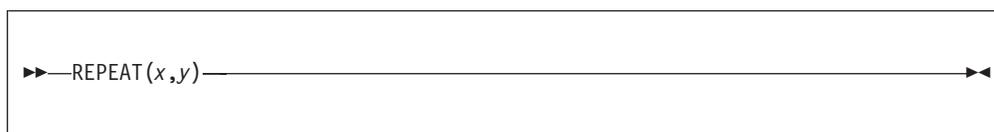
x Bit, character, graphic or widechar expression to be repeated. If x is arithmetic, the following conversions occur:
  - If it is binary, x is converted to bit string
  - If it is decimal, x is converted to character string.

y Expression. If necessary, y is converted to a real fixed-point binary value.

If y is zero or negative, the string x is returned. For an example of the REPEAT built-in function, see "COPY" on page 401.

REVERSE

REVERSE returns a nonvarying string that contains the elements of x in reverse order.

```
►►──REVERSE(x)────────────────────────────────────────────────►◄
```
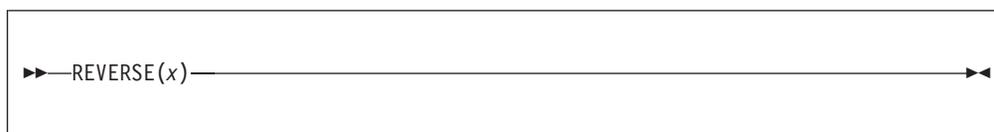
x Expression. x must have a computational type and should have a string type. If x does not have a string type, it is converted to string (that is, from numeric to bit, character, graphic or widechar), according to the rules for concatenation.

Example

```
dcl Source char value('HARPO');
dcl Target char(length(Source));

Target = reverse (Source);     /* 'OPRAH' */
```

RIGHT

RIGHT returns a string that is the result of inserting string x at the right end of a string with length n and padded on the left with the character z as needed. If z is omitted, a blank is used as the padding character.

```
►►──RIGHT(x,n─────────)──────────────────────────────────►◄
             └─,z─┘
```

x   Expression. *x* must have a computational type and can have a character type. If not, it is converted to character.

n   Expression. *n* must have a computational type and is converted to FIXED BINARY(31,0).

z   Expression. If specified, *z* must have the type CHARACTER(1) NONVARYING type.

Example

```
dcl Source char value('One Hundred SCIDS Marks');
dcl Target char(30);

Target = right (Source, length(Target), '*');
          /* '*******One Hundred SCIDS Marks'              */
```

ROUND

ROUND returns the value of *x* rounded at a digit specified by *n*. The result has the mode, base, and scale of *x*.

```
►►──ROUND(x,n)─────────────────────────────────────────────►◄
```
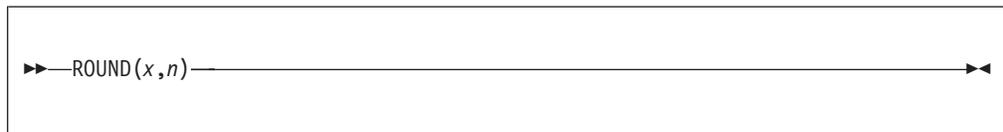
x   Real expression. If *x* is negative, the absolute value is rounded and the sign is restored.

n   Optionally-signed integer. It specifies the digit at which rounding is to occur. *n* must conform to the limits of scaling-factors for FIXED data. If *n* is greater than 0, rounding occurs at the (*n*)th digit to the right of the point. If *n* is zero or negative, rounding occurs at the (1-*n*)th digit to the left of the point.

The precision of a fixed-point result is given in the following example where (p,q) is the precision of *x*, and N is the maximum number of digits allowed:

```
(max(1,min(p-q+1+n,N)),n)
```

Thus, *n* specifies the scaling factor of the result.

In the following example, the value 6.67 is output:

```
dcl X fixed dec(5,4) init(6.6666);
put (round(X,2));
```

Results under compiler option USAGE(ROUND(ANS))

If *x* is FIXED, consider the following example where b = 2 if *x* is BINARY and b = 10 if *x* is DECIMAL:

```
round(x,n) = sign(x)*(b⁻ⁿ)* floor(abs(x)* (bⁿ) + 1/2)
```

If *x* is FLOAT and not equal to 0, consider this example where b = radix(x) and e = exponent(x):

```
round(x,n) = sign(x)*(b⁽ᵉ⁻ⁿ⁾)* floor(abs(x)* (b⁽ⁿ⁻ᵉ⁾) + 1/2)
```

Finally, if *x* is FLOAT and equal to 0, consider this example:

```
round (x,n) = 0
```

Results under compiler option USAGE(ROUND(IBM))

If *x* is FIXED, consider the following example where b = 2 if *x* is BINARY and b = 10 if *x* is DECIMAL.

```
round(x,n) = sign(x)*(b⁻ⁿ)* floor(abs(x)* (bⁿ) + 1/2)
```

If *x* is FLOAT, no action is taken.

SAMEKEY

SAMEKEY returns a bit string of length 1 indicating whether a record that has been accessed is followed by another with the same key.

```
►►──SAMEKEY(x)──────────────────────────────────────────────►◄
```
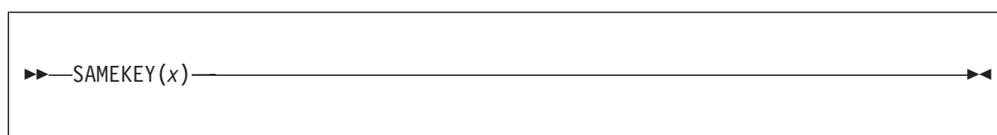
x   File reference. The file must have the RECORD attribute.

Upon successful completion of an input/output operation on file *x*, or immediately before the RECORD condition is raised, the value accessed by SAMEKEY is set to '1'B if the record processed is followed by another record with the same key, and set to '0'B if it is not.

The value accessed by SAMEKEY is also set to '0'B if:
- An input/output operation that raises a condition other than RECORD also causes file positioning to be changed or lost
- The file is not open
- No current cursor position exists in the file.

SCALE

SCALE returns a floating-point value based on the following formula:

$$x*(radix(x)^n)$$

The result has the base, mode, and precision of *x*.

```
►►──SCALE(x,n)──────────────────────────────────────────────►◄
```

x   REAL FLOAT expression.

n   Expression that must have a computational type and is converted to FIXED BINARY(31,0).

SEARCH

SEARCH returns an unscaled REAL FIXED BINARY value specifying the first position in one string at which *any* character, bit, graphic or widechar of another string appears. It also allows you to specify the location at which to start searching.

```
►►──SEARCH(x,y──────)──────────────────────────────────────►◄
              └─,n─┘
```
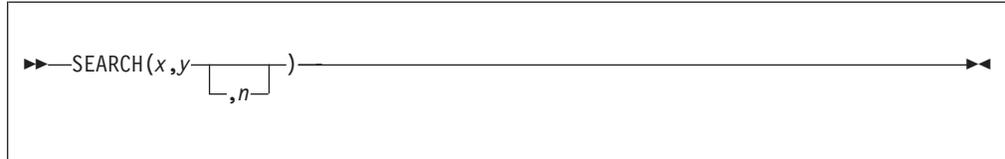
x and y
Expressions. *x* specifies the string in which to search for any character, bit, graphic or widechar that appears in string *y*.

If either *x* or *y* are the null string, the result is zero.

If *y* does not occur in *x*, the result is zero.

n Expression. *n* specifies the location within *x* at which to begin searching. It must have a computational type and is converted to FIXED BINARY(31,0).

Unless 1 ≤ *n* ≤ LENGTH(*x*)+1, STRINGRANGE condition, if enabled, is raised. Its implicit action and normal return give a result of zero.

The BIFPREC compiler option determines the precision of the result returned.

SEARCH can be used to find delimiters in a string of numbers.

Example

```
 dcl Source char value(' Our PL/I wields the Power ');
 dcl Pos fixed bin(31);

/* Find occurrences of any of the characters 'P','o',or 'w' in source *  /

 Pos = search (Source, 'Pow');            /*  returns 6 for the 'P'    */
 Pos = search (Source, 'Pow', Pos+1);   /*  returns 11 for the 'w'   */
 Pos = search (Source, 'Pow', Pos+1);   /*  returns 22 for the 'P'   */
 Pos = search (Source, 'Pow', Pos+1);   /*  returns 23 for the 'o'   */
 Pos = search (Source, 'Pow', Pos+1);   /*  returns 24 for the 'w'   */

 Pos = index (source, 'Pow',1);           /*  returns 22 for the 'Pow' */
```

In the above example, SEARCH returns the position at which any of the three characters ('P', 'o', or 'w') appear. INDEX returns the position at which the whole string 'Pow' appears.

Example

```
 dcl Source char value (' 368,475;121.,856,478')
 dcl Delims char(3) init (',;.');              /* string of delimiters */
 dcl Number(5) char(3);
 dcl Start fixed bin(31);
 dcl End fixed bin(31);

 /* Extract the three-digit numbers from the source string */
 /* by searching for the delimiters                        */
 Start = verify (Source, ' ');
                   /* find start of first number */
 End   = search (Source, ',;.', Start );
                   /* find end of first number */
```

```
         if End = 0 then
            End = length (Source) + 1;
         Number(1) = substr (Source, Start, 3);        /* 368 */
         Start = verify (Source, Delims, End);
                    /* find start of second number */
         End   = search (Source, Delims, Start );
         Number(2) = substr (Source, Start, 3);        /* 475 */
```

SEARCHR
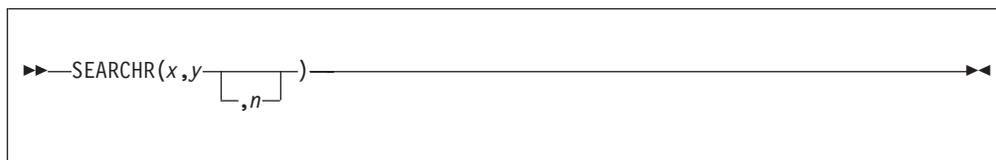
The SEARCHR function performs the same operation as the SEARCH built-in
function except for the following differences:
- The search is done from right to left.
- The default value for *n* is LENGTH(*x*).
- Unless $0 \leq n \leq$ LENGTH(*x*), the STRINGRANGE condition, if enabled, is raised.
  Its implicit action and normal return give a result of zero.

The BIFPREC compiler option determines the precision of the result returned.

The syntax for SEARCHR is described in "SEARCH" on page 468.

Example

```
dcl Source char value (' 555 Bailey Ave, San Jose, CA 95141, USA');
dcl Digits char value ('0123456789');
dcl (Start, End) fixed bin(31);
dcl Num char(20) var;

   /*   Find last number (i.e., zip code)    */

End   = searchr (Source, Digits);        /* returns 35 for the '1' */
Start = verifyr (Source, Digits, End);   /* returns 30 for the ' ' */
Num   = substr (Source, Start + 1, End - Start); /* extract number */
```

SECS

SECS returns a FLOAT BINARY(53) value which is the number of seconds (based
on Lilian format) corresponding to the date *d*.
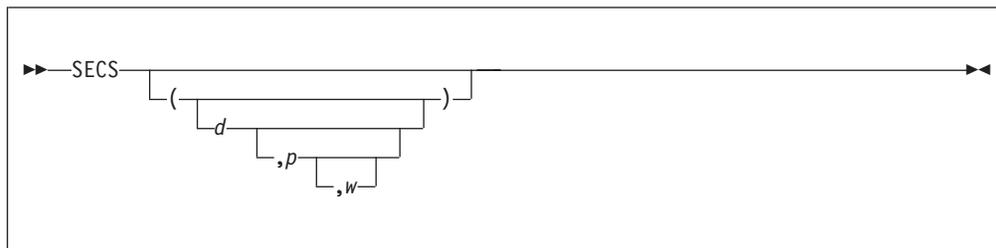
d   A string expression representing a date. If present, *d* specifies the input date as
a character string representing the date/time specified in the pattern *p*. If *d* is
missing, it is assumed to be DATETIME().

d must have a computational type and should have character type. If not, it is
converted to character.

p One of the supported date/time patterns. If *p* is omitted, it is assumed to be the default date/time pattern 'YYYYMMDDHHMISS999'.

p must have a computational type and should have character type. If not, it is converted to character.

w Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The allowed patterns are listed in Table 52 on page 378. For an explanation of Lilian format, see "Date/time built-in functions" on page 376.

Example

```
dcl Dayname (7) char(9) var static nonasgn init( 'Sunday',
                                                 'Monday',
                                                 'Tuesday',
                                                 'Wednesday',
                                                 'Thursday',
                                                 'Friday',
                                                 'Saturday');
dcl Jul4_1776_Secs float bin(53);
dcl Age_Tot_Secs pic 'Z,ZZZ,ZZZ,ZZZ,ZZ9';

Jul4_1776_Secs = secs('17760704','YYYYMMDD');    /* seconds */
Age_Tot_Secs   = secs() - Jul4_1776_Secs;   /* seconds since */
display ('USA became independent on ' ||
         dayname(weekday(secstodays(Jul4_1776_Secs))) ||
         ', July 4, 1776 and at this very moment it has been ' ||
         Age_Tot_Secs, ||    ' seconds.');
```

SECSTODATE

SECSTODATE returns a nonvarying character string containing the date in the date/time pattern specified by *p* that corresponds to *d* seconds (based on Lilian format).
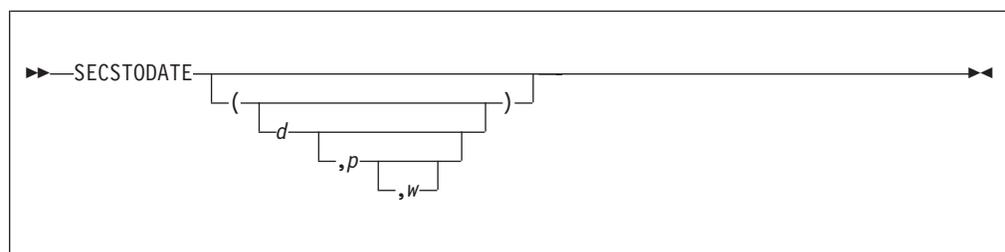
d A string expression representing a date. If omitted, it is assumed to be the value returned by DATETIME(). The value for *d* must have a computational type and is converted to FLOAT BINARY(53), if necessary.

p One of the supported date/time patterns. If omitted, *p* is assumed to be the default date/time pattern 'YYYYMMDDHHMISS999' (the default format returned by DATETIME).

w Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The allowed patterns are listed in Table 52 on page 378. For an explanation of Lilian format, see "Date/time built-in functions" on page 376.

SECSTODAYS

SECSTODAYS returns a FIXED BINARY(31,0) value that represents the number of seconds *x* converted to days, ignoring incomplete days.

```
►►──SECSTODAYS(x)─────────────────────────────────────────────►◄
```
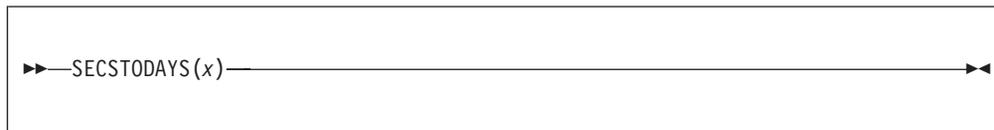
x Expression. The value for *x* must have computational type and should be FLOAT BINARY(53). If not, it is converted to FLOAT BINARY(53).

SECSTODAYS(x) is the same as $x/(24*60*60)$.

For an example, see "SECS" on page 469.

SIGN

SIGN returns an unscaled REAL FIXED BINARY value that indicates whether *x* is positive, zero, or negative.

```
►►──SIGN(x)───────────────────────────────────────────────────►◄
```
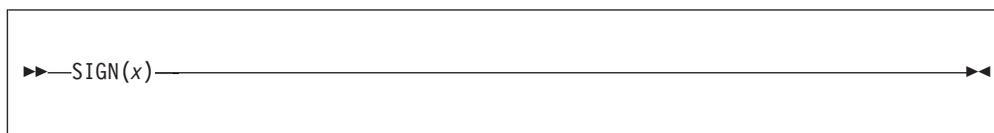
x Real expression.

The returned value is given by:

Value of x	Value Returned
x > 0	+1
x = 0	0
x < 0	−1

The BIFPREC compiler option determines the precision of the result returned.

SIGNED

SIGNED returns a signed FIXED BINARY value of *x*, with a precision specified by *p* and *q*.
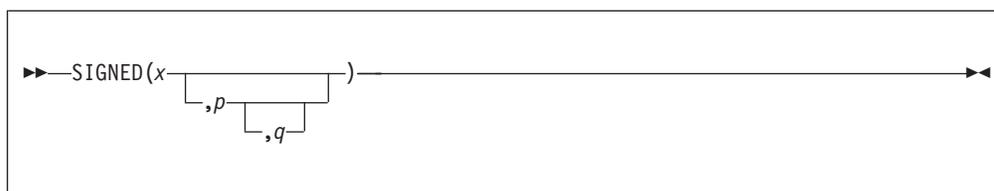
```
►►──SIGNED(x──────────────)──────────────────────────────────►◄
              └─,p─┐
                   └─,q─┘
```

x Expression.

p   Restricted expression that specifies the number of digits to be maintained throughout the operation.

q   Restricted expression that specifies the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is the default.

SIN

SIN returns a floating-point value that is an approximation of the sine of *x*. It has the base, mode, and precision of *x*.

```
►►──SIN(x)────────────────────────────────────►◄
```
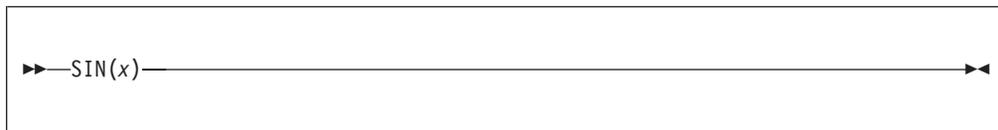
x   Expression whose value is in radians.

SIND

SIND returns a real floating-point value that is an approximation of the sine of *x*. It has the base and precision of *x*.

```
►►──SIND(x)───────────────────────────────────►◄
```
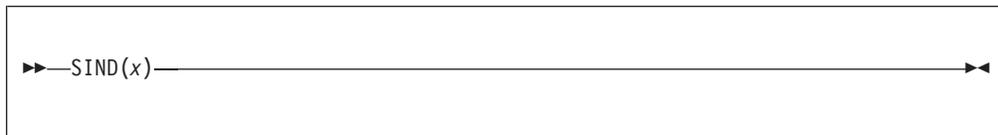
x   Real expression whose value is in degrees.

SINF

SINF is exactly like SIN except that:
- SINF calculates its result inline if hardware architecture permits.
- The argument must be real.
- The maximum supported absolute value of the argument is set by the hardware.
- Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "SIN."

SINH

SINH returns a floating-point value that represents an approximation of the hyperbolic sine of *x*. It has the base, mode, and precision of *x*.

```
►►──SINH(x)───────────────────────────────────►◄
```
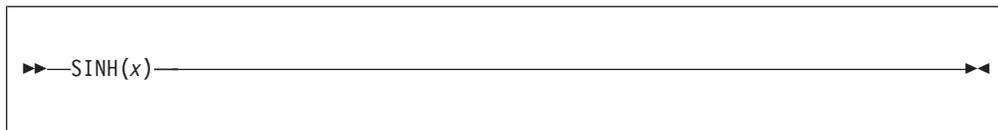
x   Expression whose value is in radians.

SIZE

SIZE returns a FIXED BINARY(31,0) value giving the implementation-defined storage, in bytes, allocated to a variable *x*.

```
►►──SIZE(x)────────────────────────────────────────────►◄
```

x   A variable of any data type, data organization, alignment, and storage class, except as listed below.

x cannot be:
- A BASED, DEFINED, parameter, subscripted, or structure or union base-element variable that is an unaligned fixed-length bit string
- A minor structure or union whose first or last base element is an unaligned fixed-length bit string (except where it is also the first or last element of the containing major structure or union)
- A major structure or union that has the BASED, DEFINED, or parameter attribute, and which has an unaligned fixed-length bit string as its first or last element
- A variable not in connected storage

The value returned by SIZE(x) is the maximum number of bytes that could be transmitted in the following circumstances:

```
declare F file record input
        environment(scalarvarying);
read file(F) into(x);
```

If *x* is:
- A varying-length string, the returned value includes the length-prefix of the string and the number of bytes in the maximum length of the string
- An area, the returned value includes the area control bytes and the maximum size of the area
- An aggregate containing areas or varying-length strings, the returned value includes the area control bytes, the maximum sizes of the areas, the length prefixes of the strings, and the number of bytes in the maximum lengths of the strings.

The SIZE built-in function must not be used on a BASED variable with adjustable extents if that variable has not been allocated.

To get the number of bytes currently required by a variable, as opposed to the number of bytes allocated to it, use the CURRENTSIZE built-in function. See "CURRENTSIZE" on page 404 for more details.

Example

```
dcl Scids   char(17)         init('See you at SCIDS!') static;
dcl Vscids  char(20) varying init('See you at SCIDS!') static;
dcl Stg fixed bin(31);

Stg = storage    (Scids);       /* 17 bytes */
Stg = currentsize (Scids);      /* 17 bytes */
Stg = size (Vscids);            /* 22 bytes */
```

```
Stg = currentsize (Vscids);      /* 19 bytes */
Stg = size (Stg);                /* 4  bytes */
Stg = currentsize (Stg);         /* 4  bytes */
```

SOURCEFILE

SOURCEFILE returns a nonvarying character string containing the name of the file that contains the statement where this function is invoked.

```
►►──SOURCEFILE──┬──────┬──────────────────────────────────────────────►◄
                └─()─┘
```
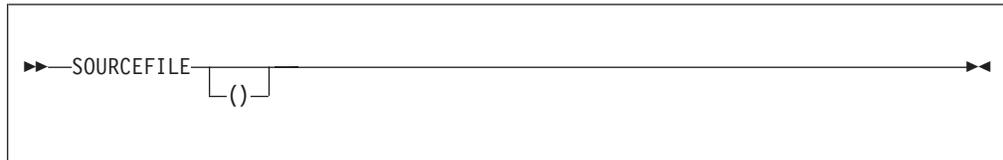
SOURCEFILE can be used in restricted expressions.

The string returned is system dependent and should be used for tracing and debugging purposes only.

SOURCELINE

SOURCELINE() returns a FIXED BINARY(31,0) value that is the line number of the statement where this function is invoked, within the file that contains that statement. If the statement extends over several source lines, the number returned is that of the line on which the statement starts.

```
►►──SOURCELINE──(──)──────────────────────────────────────────────────►◄
```

SOURCELINE() can be used in restricted expressions.

SQRT

SQRT returns a floating-point value that is an approximation of the positive square root of *x*. It has the base, mode, and precision of *x*.

```
►►──SQRT(x)───────────────────────────────────────────────────────────►◄
```

x Expression. If *x* is real, it must not be less than zero.

SQRTF

SQRTF is the same as SQRT except for these differences:
- SQRTF calculates its result inline if hardware architecture permits.
- The argument must be real.
- Invalid arguments will generate hardware exceptions.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "SQRT."

STORAGE

Abbreviation: STG

STORAGE is a synonym for SIZE. See the syntax for "SIZE" on page 473.

STRING

STRING returns an element bit or character string that is the concatenation of all the elements of *x*.

```
►►──STRING(x)──────────────────────────────────────────────►◄
```
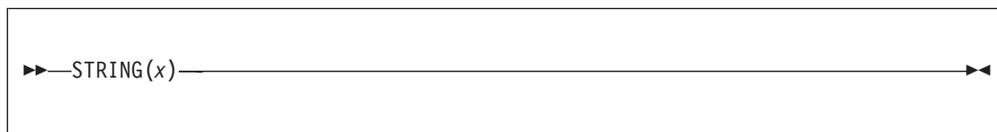
x   Aggregate or element reference.

STRING is restricted as follows:
- It cannot be applied to unions or structures containing unions.
- If applied to a scalar, the scalar must be a bit string, a character string, a pictured character string, a pictured numeric string, a graphic string, or a widechar string.
- If applied to a structure, the structure must contain no padding bytes and the elements of the structure must be either:
  - All unaligned bit strings
  - All character strings, each of which is either a character string, a pictured string, or a pictured numeric string
  - All graphic strings
  - All widechar strings
- If applied to an array, all elements in the array are subject to the restrictions as described previously.

The following are valid STRING targets:
```
dcl
  1 A,
    2 B  bit(8),
    2 C  bit(2),
    2 D  bit(8);

dcl
  1 W,
    2 X  char(2),
    2 Y  pic'aa',
    2 Z  char(6);

dcl
  1 W,
    2 X  char(2),
    2 Y  pic'99',
    2 Z char(6);
```

The following are invalid STRING targets:
```
dcl
  1 A,
    2 B  bit(8) aligned,
    2 C  bit(2),
    2 D  bit(8) aligned;
```

STRING pseudovariable

The STRING pseudovariable assigns a string to *x* as if *x* were a string scalar. Any remaining strings in *x* are filled with blanks or zero bits, or, if varying-length, are given zero length.

```
►►──STRING(x)───────────────────────────────►◄
```

x   Aggregate or element reference. Each base element of *x* must be either all bit-string or all character-string.

The STRING pseudovariable must not be used out of context.

The pseudovariable is also subject to the restrictions of the STRING built-in function. For more information on the restrictions, refer to 475.

SUBSTR

SUBSTR returns a substring, specified by *y* and *z*, of *x*.

```
►►──SUBSTR(x,y────────)─────────────────────►◄
              └─,z─┘
```
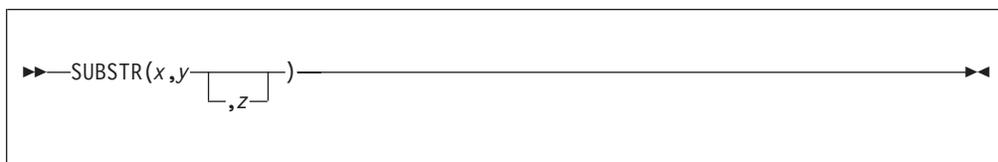
x   String expression. It specifies the string from which the substring is extracted. If *x* is not a string, it is converted to character.

y   Expression that is converted to FIXED BINARY(31,0). *y* specifies the starting position of the substring in *x*.

z   Expression that is converted to FIXED BINARY(31,0). *z* specifies the length of the substring in *x*. If *z* is zero, a null string is returned. If *z* is omitted, the substring returned is position *y* in *x* to the end of *x*.

The STRINGRANGE condition is raised if *z* is negative or if the values of *y* and *z* are such that the substring does not lie entirely within the current length of *x*. It is not raised when $y = $ LENGTH$(x)+1$ and $z = 0$. For an example of the SUBSTR built-in function, see "SEARCH" on page 468.

SUBSTR pseudovariable

The SUBSTR pseudovariable assigns a string value to a substring, specified by *y* and *z*, of *x*. The remainder of *x* is unchanged. Assignments to a varying string do not change the length of the string.

```
►►──SUBSTR(x,y────────)──────────────────────►◄
              └─,z─┘
```
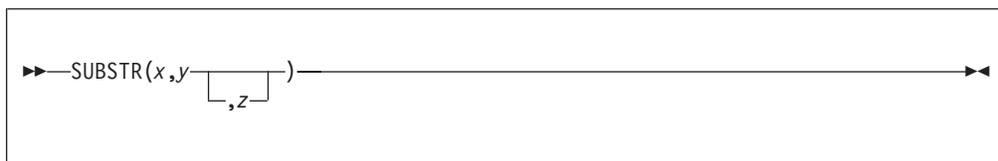
x   String-reference. *x* must not be a numeric character.

y Expression. *y* expression that can be converted to a FIXED BINARY value which specifies the starting position of the substring in *x*.

z Expression. *z* specifies the length of the substring in *x*. It can be converted to a real fixed-point binary value. If *z* is zero, a null string is returned. If *z* is omitted, the substring returned is position *y* in *x* to the end of *x*.

y and *z* can be arrays only if *x* is an array.

SUBTRACT

SUBTRACT is equivalent to ADD(x,-y,p,q).

```
►►──SUBTRACT(x,y,p───────)──────────────────────────────────►◄
                    └─,q─┘
```
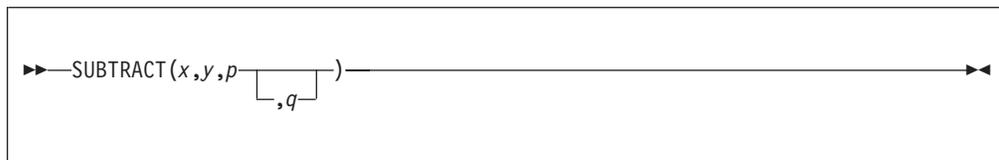
For details about arguments, refer to "ADD" on page 387 for argument descriptions.

SUCC

SUCC returns a floating-point value that is the smallest representable number larger than *x*. It is the base, mode, and precision of *x*. The OVERFLOW condition is raised if there is no such number.

```
►►──SUCC(x)──────────────────────────────────────────────────►◄
```
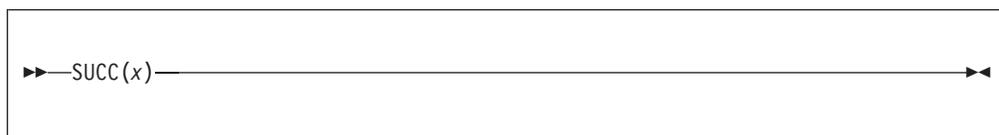
x REAL FLOAT expression.

SUCC satisfies the following relationships:

```
pred(succ(x)) = x
succ(pred(x)) = x
succ(x)       = -pred(-x)
succ(0d0)     = tiny(0d0)
```

SUM

SUM returns the sum of all the elements in *x*. The base, mode, and scale of the result match those of *x*.
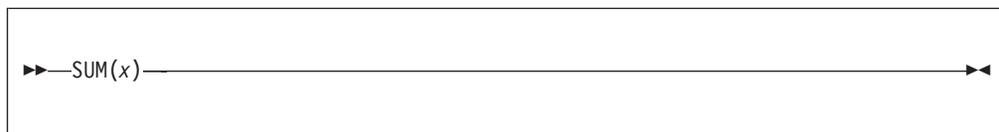
```
►►──SUM(x)───────────────────────────────────────────────────►◄
```

x Array expression. If the elements of *x* are strings, they are converted to fixed-point integer values.

If the elements of *x* are fixed-point, the precision of the result is (N,q), where N is the maximum number of digits allowed, and *q* is the scaling factor of *x*.

If the elements of *x* are floating-point, the precision of the result matches *x*.

SYSNULL

SYSNULL returns the system null pointer value. You can assign SYSNULL to handles and compare it with handles. You can use SYSNULL to initialize static pointer and offset variables.

```
►►──SYSNULL────────────────────────────────────────►◄
              └─()─┘
```

Note: NULL and SYSNULL may compare equal; however, you should not write code that depends on their equality.

See also "NULL" on page 441.

SYSTEM

SYSTEM(x) returns a FIXED BIN(31,0) value that is the return value from the command processor when it is invoked with the command contained in *x*.

```
►►──SYSTEM──────────────────────────────────────────►◄
           └─(x)─┘
```

x   Must have a computational type and should have characater type. If not, *x* is converted to character.

TALLY

TALLY returns a FIXED BINARY(31,0) result that indicates the number of times string *y* appears in string *x*. If *y* does not appear in *x*, a value of 0 is returned.

```
►►──TALLY(x,y)──────────────────────────────────────►◄
```

x and y
   String expressions.

   Both *x* and *y* must have computational type and should be character, bit, graphic or widechar type.

If either *x* or *y* are the null string, the result is zero.

Example

```
TALLY ('We''ve got the Power!', 'power');     /* returns 0 */
TALLY ('We''ve got the Power!', 'Power');     /* returns 1 */
TALLY ('We''ve got the Power!', ' ');         /* returns 3 */
TALLY ('We''ve got the Power!', 'e');         /* returns 4 */
TALLY ('1001'B, '1'B);                        /* returns 2 */
```

TAN

TAN returns a floating-point value that is an approximation of the tangent of x. It has the base, mode, and precision of x.

```
►►──TAN(x)────────────────────────────────────►◄
```

x   Expression whose value is in radians.

TAND

TAND returns a real floating-point value that is an approximation of the tangent of x. It has the base and precision of x.

```
►►──TAND(x)───────────────────────────────────►◄
```
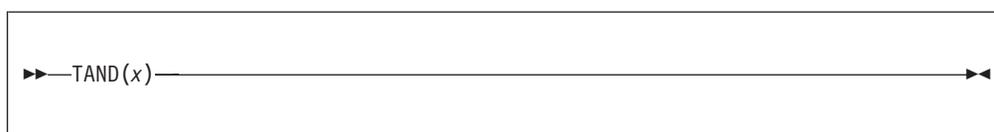
x   Real expression whose value is in degrees.

TANF

TANF is exactly like TAN except that:
- TANF calculates its result inline if hardware architecture permits.
- The argument must be real.
- The maximum supported absolute value of the argument is set by the hardware.
- Invalid arguments may raise the INVALIDOP condition, generate some other hardware exception or cause some other unpredictable result.
- The accuracy of the result is set by the hardware.

For the definition and syntax, see "TAN."

TANH

TANH returns a floating-point value that is an approximation of the hyperbolic tangent of x. It has the base, mode, and precision of x.

```
►►──TANH(x)───────────────────────────────────►◄
```
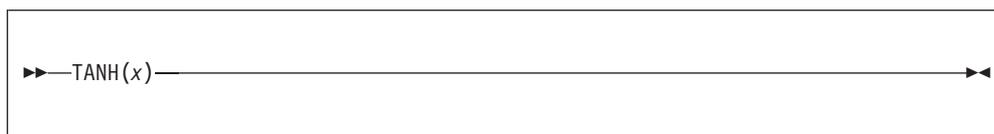
x   Expression whose value is in radians.

THREADID

WRKSTN▶

THREADID (short for thread identifier) returns a FIXED BINARY(31,0) value that is the operating system thread identifier for an attached thread. If you invoke THREADID without an argument, it returns the id for the current thread.

THREADID

```
►►──THREADID──────────────────────────────────────────►◄
              └─(──x──)─┘
```
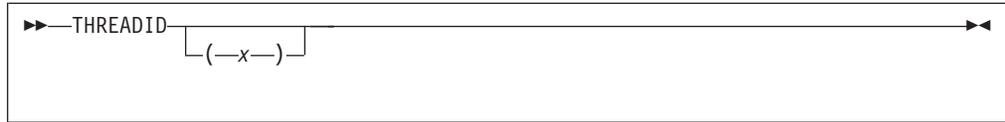
x   Task reference. You can specify the value of *x* in the THREAD option of the ATTACH statement.

The value returned by this built-in function can be used as a parameter to system functions such as DosSetPriority.

◄⊟

TIME

TIME returns a character string timestamp in the format HHMISS999.

```
►►──TIME──────────────────────────────────────────►◄
         └─()─┘
```
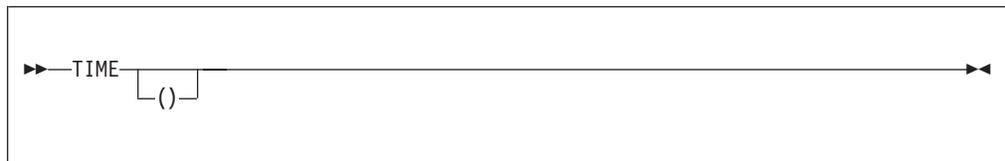
TINY

TINY returns a floating-point value that is the smallest positive value *x* can assume. It has the base, mode, and precision, of *x*.

```
►►──TINY(x)──────────────────────────────────────────►◄
```
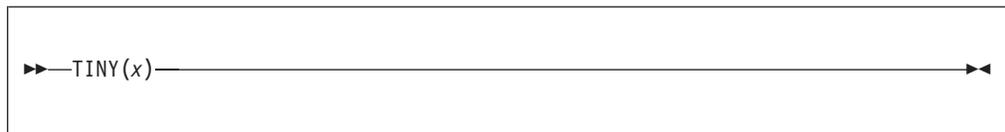
x   REAL FLOAT expression.

TINY(x) is a constant and can be used in restricted expressions.

TRANSLATE

TRANSLATE returns a character string of the same length as *x*.

```
►►──TRANSLATE(x,y──────)──────────────────────────────────►◄
                   └─,z─┘
```
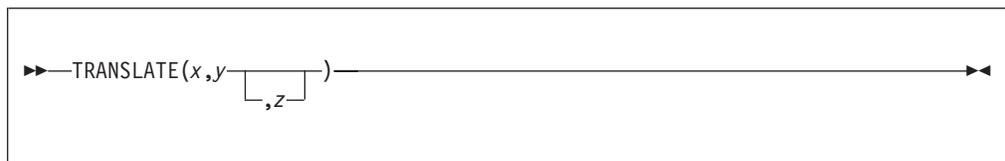
x   Character expression to be searched for possible translation of its characters.

y   Character expression containing the translation values of characters.

z   Character expression containing the characters that are to be translated. If *z* is omitted, it defaults to collate().

TRANSLATE operates on each character of *x* as follows:

If a character in *x* is found in *z*, the character in *y* that corresponds to that in *z* is copied to the result; otherwise, the character in *x* is copied directly to the result. If *z* contains duplicates, the leftmost occurrence is used.

y is padded with blanks, or truncated, on the right to match the length of *z*.

Any arithmetic or bit arguments are converted to character. TRANSLATE does not support GRAPHIC or WIDECHAR data.

Example

```
dcl source char value("Ein Raetsel gibt es nicht.");
dcl target char(length(source));
dcl (to   value ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'),
     from value ('abcdefghijklmnopqrstuvwxyz')) char;

target = translate(source, to, from);
        /* "EIN RAETSEL GIBT ES NICHT." */
```

Note that you could also use the UPPERCASE built-in for the same purpose as the TRANSLATE built-in in the example above. However, while the UPPERCASE built-in function will translate only the standard alphabetic characters, TRANSLATE can be used to translate other characters. For example, if ″Raetsel″ were spelled with an a-umlaut, TRANSLATE could translate the a-umlaut to A-umlaut if those characters were added to the *from* and *to* strings, respectively.

TRIM

TRIM returns a nonvarying character string with characters trimmed from one or both ends.

```
►►──TRIM(x──────────────)─────────────────────────────►◄
          └─,y─┐
              └─,z─┘
```
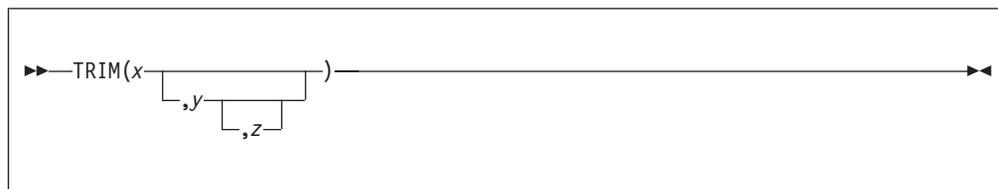
x, y, and z
  Expressions.

  Each must have a computational type and should have the attribute CHARACTER. If not, they are converted.

  x is the string from which the characters defined b*y* y are trimmed from the left, and the characters defined by *z* are trimmed from the right.

  If *z* is omitted, it defaults to a CHARACTER(1) NONVARYING string containing one blank.

If *y* and *z* are both omitted, they both default to a CHAR(1) NONVARYING string containing one blank.

Example

```
        dcl Source char value(" *** PL/I's got the Power!  ***  ");
        dcl Target char(length(Source)) varying;

        Target = trim(Source, ' ', '* ');
                    /* "*** PL/I's got the Power!" */
```

TRUNC

TRUNC returns an integer value that is the truncated value of *x*. If *x* is positive or 0, this is the largest integer value less than or equal to *x*. If *x* is negative, this is the smallest integer value greater than or equal to *x*. This value is assigned to the result.

```
►►──TRUNC(x)────────────────────────────────────────►◄
```

x   Real expression.

The base, mode, scale, and precision of the result match those of *x*. Except when x is fixed-point with precision (*p,q*), the precision of the result is given by:

```
(min(N,max(p-q+1,1)),0)
```

where N is the maximum number of digits allowed.

TYPE

TYPE returns the typed structure or union located by the handle, *x*.

```
►►──TYPE(x)──────────────────────────────────────────►◄
```
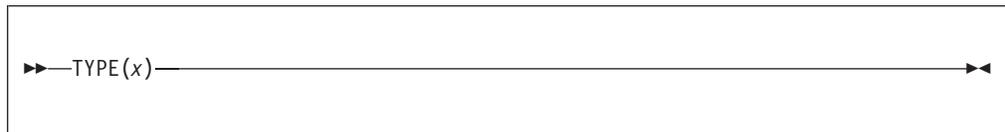
x   Handle

TYPE(x) dereferences the typed structure (or union) *x*. For an example of the TYPE built-in functions, see "TYPE pseudovariable."

TYPE pseudovariable

The TYPE pseudovariable assigns a typed structure or union to the storage located by the handle *x*.

```
►►──TYPE(x)──────────────────────────────────────────►◄
```
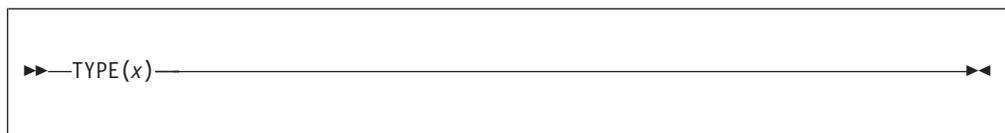
x   Handle

Given a defined structure *T*, the following assignments are valid:

```
dcl P1 handle T;
dcl P2 handle T;
dcl D1 type T;
dcl D2 type T;

D1 = type(P2);              /* Assigns the storage located by P2 to D1 */
type(P1) = type(P2);
type(P1) = D2;              /* Assigns D2 to the storage located by P1 */
```

UNALLOCATED

UNALLOCATED returns a bit(1) value indicating whether or not a specified pointer value is the start of a piece of allocated storage. To use this built-in function, you must also specify the CHECK(STORAGE) compile-time option.

```
►►──UNALLOCATED(P)──────────────────────────────────────────────►◄
```

p Pointer expression.

UNALLOCATED returns the bit(1) value '1'b if the specified pointer value is the start of a piece of storage obtained via the ALLOCATE statement or the ALLOCATE built-in function.

UNSIGNED

UNSIGNED returns an unsigned FIXED BINARY value of *x*, with a precision specified by *p* and *q*.

```
►►──UNSIGNED(x──────────────)──────────────────────────────────►◄
               └─,p─┬──────┘
                    └─,q─┘
```
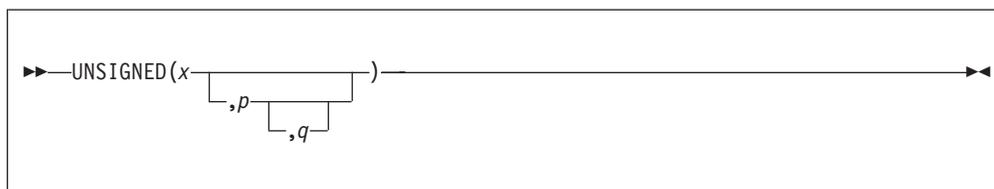
x Expression.

p Integer. It specifies the number of digits to be maintained throughout the operation.

q Optionally-signed integer. It specifies the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is the default.

UNSPEC

UNSPEC returns a bit string that is the internal coded form of *x*.

```
►►──UNSPEC(x)──────────────────────────────────────────────────►◄
```
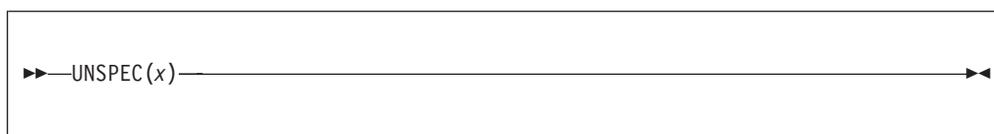
x Scalar, array, structure, or union expression.

UNSPEC

The UNSPEC built-in function is subject to the following rules:

- Under the compiler option USAGE(UNSPEC(IBM)),
  - UNSPEC of structure references and expressions is not allowed.
  - UNSPEC of an array yields an array of BIT.
- Under the compiler option USAGE(UNSPEC(ANS)),
  - For aggregates, UNSPEC is allowed only for those that contain no padding bytes or bits.
  - The result will always be BIT scalar. UNSPEC of an array does not yield an array of BIT.

Note: Use of UNSPEC can affect the portability of your program.

The length of the returned bit string depends on the attributes of *x*, as shown in Table 65.

Table 65. Length of bit string returned by UNSPEC

Bit-String length	Attribute of *x*
8	SIGNED FIXED BINARY(p,q), 1 <= p <= 7 UNSIGNED FIXED BINARY(p,q), 1 <= p <= 8 ORDINAL SIGNED PRECISION(p), 1 <= p <= 7 ORDINAL UNSIGNED PRECISION(p), 1 <= p <= 8
16	SIGNED FIXED BINARY(p,q), 8 <= p <= 15 UNSIGNED FIXED BINARY(p,q), 9 <= p <= 16 ORDINAL SIGNED PRECISION(p), 8 <= p <= 15 ORDINAL UNSIGNED PRECISION(p), 9 <= p <= 16
32	ENTRY LIMITED SIGNED FIXED BINARY(p,q), 16 <= p <= 31 UNSIGNED FIXED BINARY(p,q), 17 <= p <= 32 ORDINAL SIGNED PRECISION(p), 16 <= p <= 31 ORDINAL UNSIGNED PRECISION(p), 17 <= p<= 32 FLOAT DECIMAL(p), 1 <= p <= 6 FLOAT BINARY(p), 1 <= p <= 21 OFFSET FILE constant or variable POINTER HANDLE
64	SIGNED FIXED BINARY(p), 31 < p UNSIGNED FIXED BINARY(p), 32 < p FLOAT BINARY(p), 21 < p < 53 FLOAT DECIMAL(p), 7 <= p <= 16 LABEL constant or variable ENTRY constant or variable
128	FLOAT BINARY(p), 54 <= p FLOAT DECIMAL(p), 17 <= p TASK
n	BIT(n)
8*n	CHARACTER(n)PICTURE (with character-string-value length of *n*)
16*n	GRAPHIC(n)WIDECHAR(n)
16+n	BIT(n) VARYING where *n* is the maximum length of *x*
16+(8*n)	CHARACTER(n) VARYING where *n* is the maximum length of *x*
8+(8*n)	CHARACTER(n) VARYINGZ where *n* is the maximum length of *x*

Table 65. Length of bit string returned by UNSPEC (continued)

Bit-String length	Attribute of *x*
16+(16*n)	GRAPHIC(n) VARYING where *n* is the maximum length of *x*WIDECHAR(n) VARYING where *n* is the maximum length of *x*
16+(16*n)	GRAPHIC(n) VARYINGZ where *n* is the maximum length of *x*WIDECHAR(n) VARYINGZ where *n* is the maximum length of *x*
8*(n+16)	AREA (n)
8*FLOOR(n)	FIXED DECIMAL (p,q) where n = (p+2)/2

Alignment and storage requirements for program-control data can vary across supported systems.

If *x* is a varying-length string, its two-byte prefix is included in the returned bit string. If *x* is an area, the returned value includes the control information.

UNSPEC pseudovariable

The UNSPEC pseudovariable assigns a bit value directly to *x*; that is, without conversion. The bit value is padded, if necessary, on the right with '0'B to match the length of *x*, according to Table 65.

```
►►─UNSPEC(x)──────────────────────────────────────────────►◄
```
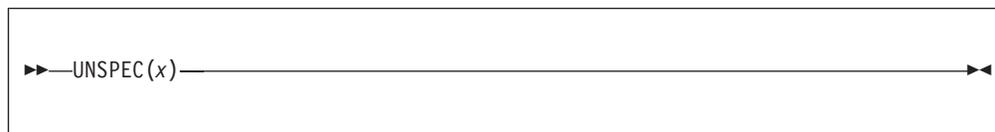
x   Reference.

If *x* is a varying length string, its 2-byte prefix is included in the field to which the bit value is assigned. If *x* is an area, its control information is included in the receiving field.

The pseudovariable is subject to the rules for the UNSPEC built-in function described in "UNSPEC" on page 483.

Note: Use of UNSPEC can affect the portability of your program.

Example

```
 dcl 1 Str1 nonasgn static,
      2 * fixed bin(15) littleendian init(16),  /* '1000'X */
      2 * char init('33'x),
      2 * bit init('1'b),
      2 Ba(4) bit init('1'b, '0'b, '1'b, '0'b),
      2 B3 bit(3) init('111'b),
      2 * char(0);
 dcl Bit_Str1 bit(size(Str1)*8);
 dcl Bit_Ba   bit(dim(Ba)*length(Ba(1)));
 dcl Bit_B3   bit(length(B3));

 Bit_Ba   = unspec(Ba);   /* result is scalar '1010'B not an array */
 Bit_B3   = unspec(B3);   /* '111'B                                */
 Bit_Str1 = unspec(Str1); /* '100033D7'B4 or
                              '100033'B4 ‖ '11010111'B            */
```

UPPERCASE

UPPERCASE returns a character string with all the alphabetic characters from a to z converted to their uppercase equivalent.

```
►►──UPPERCASE(x)──────────────────────────────────────────────►◄
```
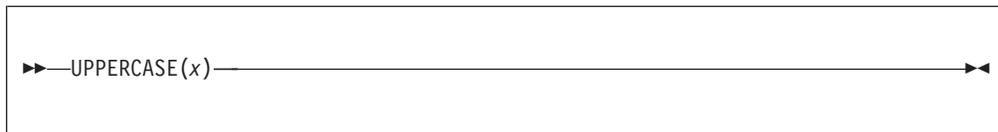
x Expression. If necessary, *x* is converted to character.

UPPERCASE(x) is equivalent to

```
TRANSLATE( x,
           'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
           'abcdefghijklmnopqrstuvwxyz' )
```

VALID

VALID returns a BIT(1) value that is '1'B under the following conditions:
- If *x* is PICTURE and its contents are valid for *x*'s picture specification
- If *x* is FIXED DECIMAL and the data in *x* is valid fixed decimal data

If these conditions are not met, the result is '0'B.

```
►►──VALID(x)──────────────────────────────────────────────────►◄
```
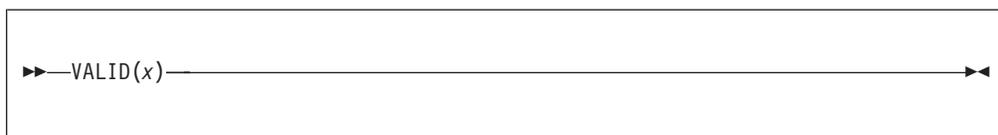
x Reference with either picture or fixed decimal type.

VALIDDATE

VALIDDATE returns a '1'B if the string d holds a date/time value that matches the pattern p.
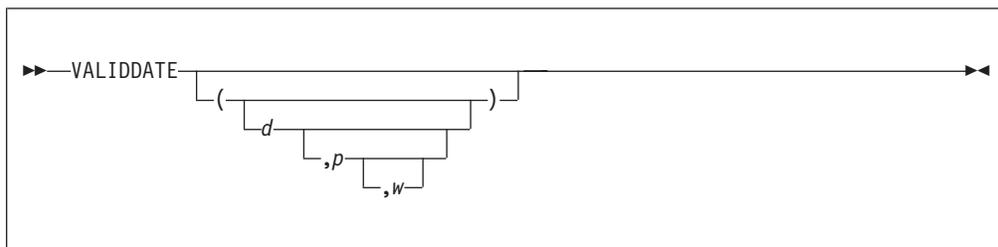
```
►►──VALIDDATE─┬───────────────────┬──────────────────────────►◄
             └─(─┬─────┬─────────┬─)─┘
                 └──d──┘         
                      └─,p─┬──────┘
                           └─,w─┘
```

d A string expression representing a date.

If present, *d* specifies the input date as a character string representing date/time according to the pattern *p*. If *d* is missing, it is assumed to be DATETIME().

d must have computational type and should have character type. If not, *d* is converted to character.

p One of the supported date/time patterns.

If present, it specifies the date/time pattern of *d*. If *p* is missing, it is assumed to be the default date/time pattern of 'YYYYMMDDHHMISS9999'.

p must have computational type and should have character type. If not, it is converted to character.

w Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

Allowable patterns are listed in Table 52 on page 378. For an explanation of Lilian format, see "Date/time built-in functions" on page 376.

Example

```
dcl duedate   char(8);
dcl (b1,b2)   bit(1);

duedate = '19950228';
b1 = validdate( duedate, 'YYYYMMDD' ); /* b1 = '1'b */

duedate = '02301995';
b2 = validdate( duedate, 'DDMMYYYY' ); /* b2 = '0'b */
```

VARGLIST

VARGLIST returns the address of the first optional parameter passed to a procedure with a variable number of arguments

```
►►──VARGLIST()──────────────────────────────────────────────►◄
```
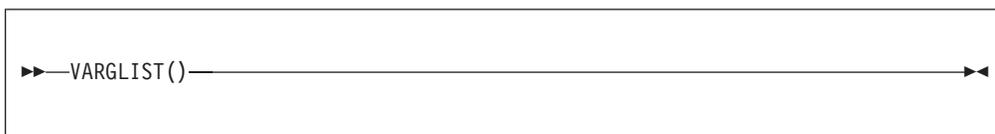
VARGLIST may be used only inside a procedure whose last parameter has the LIST attribute.

VARGSIZE

VARGSIZE returns the number of bytes that a variable would occupy on the stack if it were passed byvalue.
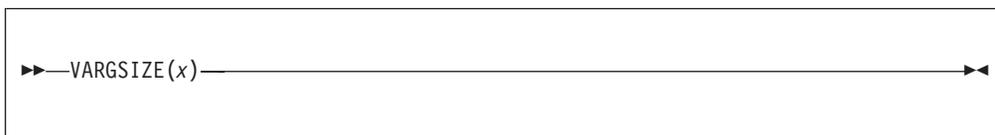
```
►►──VARGSIZE(x)──────────────────────────────────────────────►◄
```

x A variable of any data type, data organization, alignment, and storage class, except as listed below.

x cannot be:

- A BASED, DEFINED, parameter, subscripted, or structure or union base-element variable that is an unaligned fixed-length bit string
- A minor structure or union whose first or last base element is an unaligned fixed-length bit string (except where it is also the first or last element of the containing major structure or union)
- A major structure or union that has the BASED, DEFINED, or parameter attribute, and which has an unaligned fixed-length bit string as its first or last element

> • A variable not in connected storage

> VARGSIZE(x) returns the number of bytes that x would occupy on the stack if it were passed byvalue. This value will be at least as large as SIZE(x); it will be larger if the value returned by SIZE(x) needs to be rounded up to a 4-byte multiple.

> VARGSIZE is meant to be used only inside a procedure whose last parameter has the LIST attribute.

VERIFY

> VERIFY returns an unscaled REAL FIXED BINARY value indicating the position in *x* of the leftmost character, widechar, graphic, or bit that is not in *y*. It also allows you to specify the location within *x* at which to begin processing.

> If all the characters, widechars, graphics, or bits in *x* do appear in *y*, a value of zero is returned. If *x* is the null string, a value of zero is returned. If *x* is not the null string and *y* is the null string, the value of *n* is returned. The default value for *n* is one.

```
►►──VERIFY(x,y─────────)─────────────────────────────────────►◄
              └─,n─┘
```
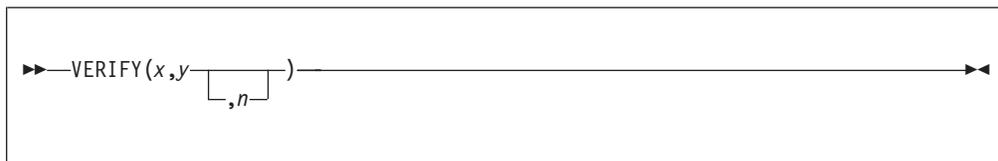
> **x** String-expression.

> **y** String-expression.

> **n** Expression *n* specifies the location within *x* where processing begins. It must have a computational type and is converted to FIXED BINARY(31,0).

> Unless 1 ≤ *n* ≤ LENGTH(*x*) + 1, the STRINGRANGE condition, if enabled, is raised. Its implicit action and normal return give a result of 0. If *n* = LENGTH(*x*) + 1, the result is zero.

> The BIFPREC compiler option determines the precision of the result returned.

Example

```
X = '  a  b';          /* Two blanks in each space */
Y = ' ';               /* One blank                */
N = 1;
I = verify(X,Y,N);     /* I = 3 */

do while (I > 0);
  display ( 'Nonblank at position ' || trim(I) );
  N = I + 1;
  I = verify(X,Y,N);
end;
```

> After the first pass through the do-loop, N=4 and VERIFY(X,Y,N) returns 6. After the second pass, N=7 (LENGTH(x)+1), VERIFY(X,Y,N) now returns 0, and the loop ends.

VERIFYR

The VERIFYR function performs the same operation as the VERIFY built-in function except that:
* The verification is done from right to left.
* The default value for *n* is LENGTH(*x*).

```
►►──VERIFYR(x,y─────────)──────────────────────────────────►◄
                └─,n─┘
```
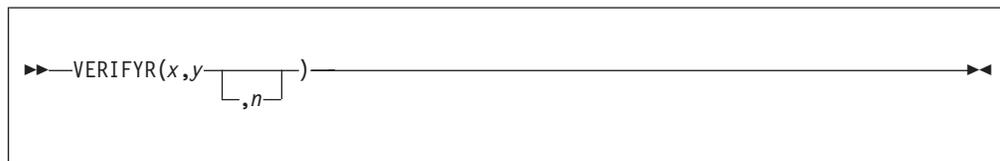
Unless 0 ≤ *n* ≤ LENGTH(*x*), the STRINGRANGE condition, if enabled, is raised. If *n* = 0, the result is zero.

The BIFPREC compiler option determines the precision of the result returned.

For argument descriptions, refer to "VERIFY" on page 488.

Example

```
X = 'a  b ';         /* Two blanks in each space */
Y = ' ';             /* One blank                */
N = length(X);       /* N = 6                    */
I = verifyr(X,Y,N);  /* I = 4                    */

do while (I > 0);
  display ( 'Nonblank at position ' || trim(I) );
  N = I - 1;
  I = verifyr(X,Y,N);
end;
```

After the first pass through the do-loop, N=3 and VERIFYR(X,Y,N) returns 1. After the second pass, N=0, VERIFYR(X,Y,N) returns 0, and the loop ends. For another example, see "SEARCHR" on page 469.

WCHARVAL

WCHARVAL returns the WIDECHAR(1) value corresponding to an integer.

```
►►──WCHARVAL──(─n─)──────────────────────────────────────────►◄
```
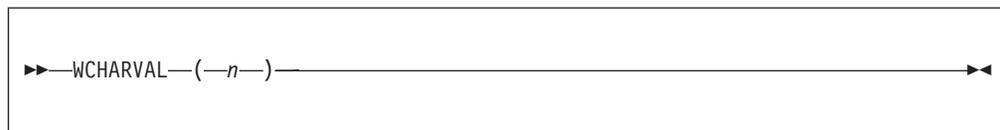
n Expression converted to UNSIGNED FIXED BIN(16) if necessary.

If n is in bigendian format, WCHARVAL(n) has the same bit value as n (that is, UNSPEC(WCHARVAL(n)) is equal to UNSPEC(n)), but it has the attributes WIDECHAR(1).

WCHARVAL is the inverse of RANK (when applied to widechar).

WEEKDAY

WEEKDAY returns a FIXED BINARY(31,0) value that is the number of days *x* converted to the day of the week, where 1=Sunday, 2=Monday, . . . 7=Saturday. If *x* is missing, it is assumed to be DAYS for today.

```
►►──WEEKDAY─────────────────────────────────────►◄
            └─(─┬────┬─)─┘
                └─x─┘
```

x Expression. If present, *x* specifies the input date as days. If missing, *x* is assumed to be DAYS().

If *x* is missing and today's date is not available from the system, a result of zero is returned.

x must have computational type and will be converted to FIXED BINARY(31,0), if necessary.

For an example of WEEKDAY, see "SECS" on page 469.

WHIGH

WHIGH returns a widechar string of length *x*, where each widechar has the highest widechar value (hexadecimal FFFF).
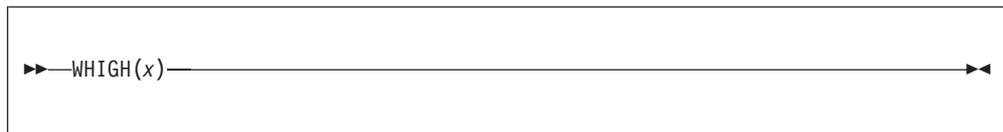
```
►►──WHIGH(x)─────────────────────────────────────►◄
```

x Expression. If necessary, *x* is converted to a positive real fixed-point binary value. If *x* = 0, the result is the null widechar string.

WIDECHAR

WIDECHAR returns the widechar value of *x*, with a length specified by *y*.

```
►►──WIDECHAR(x─┬─────┬─)──────────────────────────►◄
              └─,y─┘
```
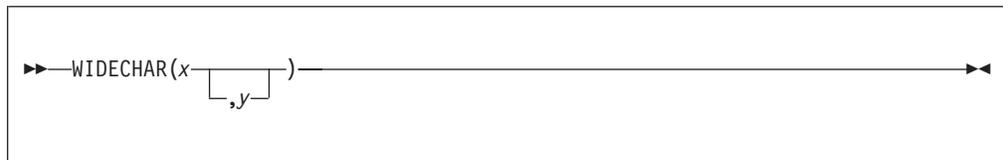
Abbreviation: WCHAR

x Expression.

x must have a computational type.

The values of *x* are not checked.

y Expression. If necessary, y is converted to a real fixed-point binary value.

If *y* is omitted, the length is determined by the rules for type conversion.

y cannot be negative.

If *y* = 0, the result is the null widechar string.

WLOW

WLOW returns a widechar string of length *x*, where each widechar has the lowest widechar value (hexadecimal 0000).

```
►►──WLOW(x)────────────────────────────────────────────►◄
```

x   Expression. If necessary, *x* is converted to a positive real fixed-point binary value. If *x* = 0, the result is the null widechar string.

XMLCHAR

The XMLCHAR built-in function dumps data from a structure as XML into a buffer. It returns the number of bytes written to the buffer.

```
►►──XMLCHAR──(──x──,──p──,──n──)──────────────────────────►◄
```

x   Structure-reference.

p   Address of target buffer.

n   Length of target buffer.

The buffer length must have a computational type and will be converted to FIXED BINARY(31,0).

The buffer length must be nonnegative.

The structure-reference *x* must contain only computational data, i.e. only string and numeric data. The structure-reference *x* may contain arrays, but if itself is an array, then it must be completely subscripted.

The structure-reference *x* may contain substructures, but any contained substructure must not use an * in place of a name. However, an * may be used as the name of a base element, but in that case, the unnamed element will not be written to the target buffer.

When the xml is created,
- each name in the struture will be witten out, first enclosed ″<″ and ″>″ and later enclosed in ″</″ and ″>″
- numeric and bit data will be converted to character
- leading and trailing blanks will be trimmed wherever possible

Example

Given this code fragment:

```
dcl buffer   char(800);
dcl written  fixed bin(31);
dcl next     pointer;
dcl left     fixed bin(31);
dcl
  1 a,
    2 a1,
      3 b1 char(8),
      3 b2 char(8),
    2 a2,
      3 c1 fixed bin,
      3 c2 fixed dec(5,1);

b1 = ' t1';
b2 = 't2';
c1 = 17;
c2 = -29;
next = addr(buffer);
left = stg(buffer);
written = xmlchar( a, next, left );
next += written;
left -= written;
```

The following bytes would be written to the buffer, and `written` would be set equal to 72.

```
<A><A1><B1>t1</B1><B2>t2</B2></A1><A2><C1>17</C1><C2>-29.0</C2></A2></A>
```

Y4DATE

Y4DATE takes a date value with the patter 'YYMMDD' and returns the date value with the two-digit year widened to a four-digit year.
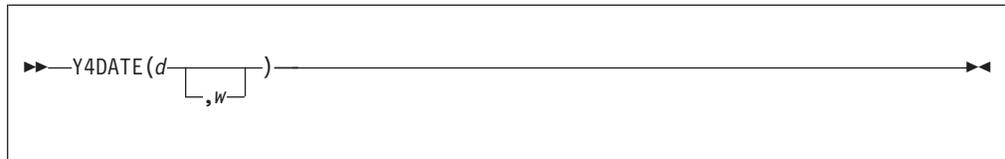
d   A string expression representing a date.

d must have computational type and should have character type. If not, *d* is converted to character.

w   Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The returned value has the attributes CHAR(8) NONVARYING and is calculated as follows:

```
dcl y2 pic'99';
dcl y4 pic'9999';
dcl c  pic'99';

y2 = substr(d,1,2);
cc = w/100;

if y2 < mod(w,100) then
  y4 = 100*cc + 100 + y2;
```

```
  else
    y4 = 100*cc + y2;

  return( y4 || substr(d,3) );
```

Y4DATE('990101',1950) returns '19990101'
Y4DATE('000101',1950) returns '20000101'

Y4JULIAN

Y4JULIAN takes a date value with the patter 'YYDDD' and returns the date value with the two-digit year widened to a four-digit year.
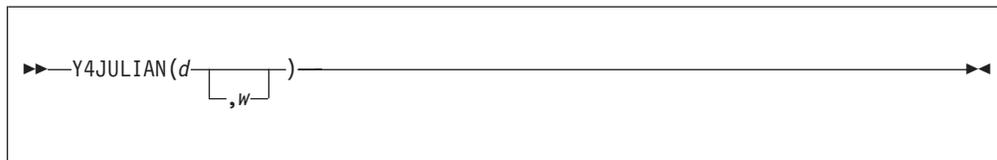
d   A string expression representing a date. The length of *d* must be at least 5. If it is larger than 5, excess characters must be formed by leading blanks.

d must have computational type and should have character type. If not, it is converted to character.

w   Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The returned value has the attributes CHAR(7) NONVARYING and is calculated as follows:

```
  dcl y2 pic'99';
  dcl y4 pic'9999';
  dcl c  pic'99';

  y2 = substr(d,1,2);
  cc = w/100;

  if y2 < mod(w,100) then
    y4 = 100*cc + 100 + y2;
  else
    y4 = 100*cc + y2;

  return( y4 || substr(d,3) );
```

Y4JULIAN('99001',1950) returns '1999001'
Y4JULIAN('00001',1950) returns '2000001'.

Y4YEAR

Y4YEAR takes a date value with the patter 'YY' and returns the date value with the two-digit year widened to a four-digit year.

```
►►──Y4YEAR(d────────)────────────────────────────►◄
              └─,w─┘
```

d   A string expression representing a date. The length of *d* must be at least 2. If it is larger than 2, excess characters must be formed by leading blanks.

   d must have computational type and should have character type. If not, it is converted to character.

w   Specifies an expression (such as 1950) that can be converted to an integer. If negative, it specifies an offset to be subtracted from the value of the year when the code runs. If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

The returned value has the attributes CHAR(4) NONVARYING and is calculated as follows:

```
dcl y2 pic'99';
dcl y4 pic'9999';
dcl c  pic'99';

y2 = d;
cc = w/100;

if y2 < mod(w,100) then
  y4 = 100*cc + 100 + y2;
else
  y4 = 100*cc + y2;

return( y4 );
```

Y4YEAR('99',1950) returns '1999'
Y4YEAR('00',1950) returns '2000'

Chapter 20. Type Functions

Using type functions, you can manipulate defined types. Type functions are distinguished from built-in functions in the following ways:
- At least one of the arguments is a defined type.
- They cannot be declared.
- Arguments are enclosed in the (: and :) composite symbols, rather than in (and) symbols.

In this chapter, the type functions are listed in alphabetical order. In general, each description has the following format:
- A heading showing the syntax of the reference
- A description of the value returned
- A description of any arguments
- Any other qualifications on using the function.

Invoking type functions

Use the following syntax to invoke type functions.
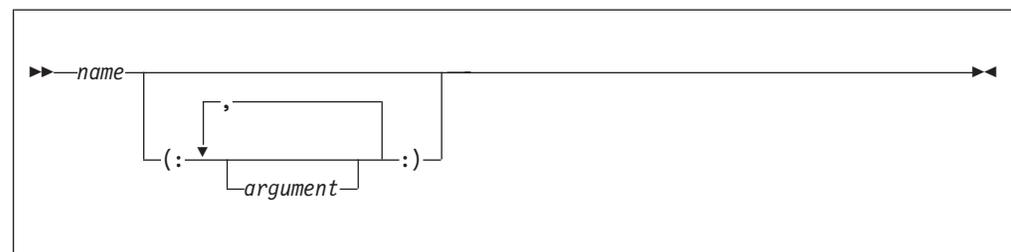
The arguments for a type function are enclosed by the delimiters (: and :).

Specifying arguments for type functions

Arguments for type functions can be type names (aliases, named structures and unions, ordinals) and other data types.

Brief descriptions of type functions

Table 66. Type functions

Function	Description
BIND	Converts a pointer to a handle for a type
CAST	Converts an expression to a specified type using C conversion rules
FIRST	Returns the first value in an ordinal set
LAST	Returns the last value in an ordinal set
NEW	Acquires storage for a structure type and returns a handle to the acquired storage
RESPEC	Changes the attributes of an expression to a specified type without changing the bit pattern of the expression
SIZE	Returns the amount of storage needed to represent a type

BIND

BIND converts the pointer *p* to a handle for the structure type *t*. The BIND function can be used as a locator for a member of a typed structure.

```
►►──BIND──(:──t──,──p──:)────────────────────────────────►◄
```

t   Name of a structure type

p   Pointer expression

CAST

CAST converts the expression *x* to the type *t* using C conversion rules.

```
►►──CAST──(:──t──,──x──:)──────────────────────────────►◄
```

t   Name of a scalar ″C type″

x   A scalar expression also having ″C type″

The supported ″C types″ are
- REAL FIXED BIN(p,0)
- REAL FIXED DEC(p,q) where p >= q and q>= 0.
- NATIVE FLOAT
- ORDINAL
- POINTER or HANDLE
- LIMITED ENTRY

If *x* is FLOAT or FIXED DEC, then *t* must be FLOAT, FIXED or ORDINAL, and if *t* is FLOAT or FIXED DEC, then *x* must be FLOAT, FIXED or ORDINAL.

Any conversions that are needed follow the ANSI C rules. This means, for instance, that SIZE will not be raised by CAST and that if negative values are cast to UNSIGNED, then the result will be a large positive number.

FIRST

FIRST returns the first value in the ordinal set *t*.
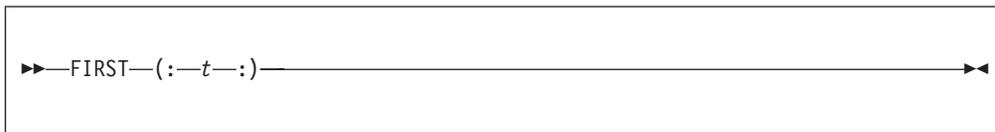
```
▶▶──FIRST──(:──t──:)──────────────────────────────────────────────▶◀
```

t   Name of an ordinal type

Example

```
define ordinal Color ( Red,
                       Orange,
                       Yellow,
                       Green,
                       Blue,
                       Indigo,
                       Violet );

display (ordinalname( first(:Color:) ));  /* RED */
```

LAST

LAST returns the last value in the ordinal set *t*.
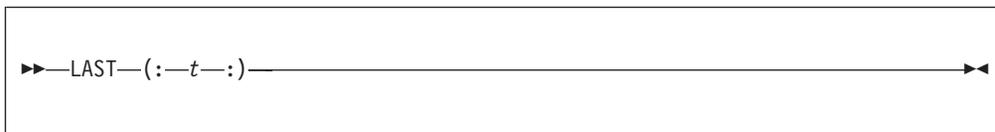
```
▶▶──LAST──(:──t──:)───────────────────────────────────────────────▶◀
```

t   Name of an ordinal type

Example

```
define ordinal Color ( Red,
                       Orange,
                       Yellow,
                       Green,
                       Blue,
                       Indigo,
                       Violet );

display (ordinalname( last(:Color:) )); /* VIOLET */
```

NEW

NEW acquires heap storage for structure type *t* and returns a handle to the acquired storage.

```
►►─NEW─(:─t─:)──────────────────────────────────────►◄
```

t   Name of a structure type

NEW(:t:) is equivalent to BIND(: t, ALLOC(SIZE(:t:)) :).

RESPEC

RESPEC changes the attributes of the expression *x* to the type *t* without changing the bit value of the expression.

```
►►─RESPEC─(:─t─,─x─:)────────────────────────────────►◄
```

t   Name of a scalar type

p   A scalar expression

x must have the same as *t*, and if either *x* or *t* is UNALIGNED BIT, then both must be (in which case the function is somewhat uninteresting since it would do nothing).

As an example, if *t* is a type with the attributes LIMITED ENTRY, then RESPEC(t, sysnull()) would return a ″null″ function pointer.

SIZE

SIZE returns the amount of storage needed for a variable declared with the type *t*.

```
►►─SIZE─(:─t─:)──────────────────────────────────────►◄
```

t   Name of a structure or union type

Chapter 21. Preprocessor Facilities

Preprocessor facilities

The compiler provides a MACRO preprocessor for source program alteration. It is executed prior to compilation, when you specify the MACRO or PP(MACRO) compile-time option. The MACRO preprocessor scans the preprocessor input and generates preprocessor output. The preprocessor output can serve as input to the compiler.

This description of the preprocessor assumes that you know the PL/I language described throughout this publication.

The *Preprocessor input* is a string of characters. consisting of intermixed:

- *Preprocessor statements.*[1] Preprocessor statements are executed as they are encountered by the preprocessor scan (with the exception of preprocessor procedures, which must be invoked in order to be executed). Preprocessor statements, except those in preprocessor procedures, begin with a percent symbol (%). Using a blank to separate the percent symbol from the rest of the statement is optional.

  The preprocessor executes preprocessor statements and alters the input text accordingly. Preprocessor statements can cause alteration of the input text in any of the following ways:
  - Any identifier (and an optional argument list) appearing in the input text can be changed to an arbitrary string of text.
  - You can indicate which portions of the input text to copy into the preprocessor output.
  - A string of characters residing in a library can be included in the preprocessor input.
- *Listing control statements*, which control the layout of the printed listing of the program. These statements affect both the insource listing (the preprocessor input) and the source listing (the preprocessor output) and are described in Chapter 9, "Statements and directives," on page 187.
- *Input text*, which is preprocessor input that is not a preprocessor statement or a listing control statement. The input text can be a PL/I source program or any other text, provided that it is consistent with the processing of the input text by the preprocessor scan, described below.

Preprocessor output [2] is a string of characters consisting of intermixed:

- *Listing control statements.* Listing control statements that are scanned in the preprocessor input are copied to the preprocessor output.
- *Output text.* Input text that is scanned and possibly altered is placed in the preprocessor output.

You can specify compile-time options that cause the preprocessor input to be printed or the preprocessor output or both to be printed or to be written to a data set.

---

1. For clarity in this discussion, preprocessor statements are shown with the % symbol (even though, when used in a preprocessor procedure, such a statement would not have a % symbol).

2. Preprocessor replacement output is shown in a formatted style, while actual execution-generated replacement output is unformatted.

Preprocessor Options

The preprocessor is invoked when you specify the MACRO or PP(MACRO) compile-time option.

You may also specify compiler options that affect the preprocessor only. Some of the options can significantly change the behavior of the preprocessor. Of particular note are the options:

FIXED
> Specifies how FIXED variables are treated. This option has two suboptions:
>
> **BINARY**
>> Specifies that FIXED variables are treated as BINARY
>
> **DECIMAL**
>> Specifies that FIXED variables are treated as DECIMAL

CASE
> Specifies if input text is converted to uppercase. This option has two suboptions:
>
> **ASIS**
>> Specifies that input text is left ″as is″.
>
> **UPPER**
>> Specifies that input text is converted to upper case.

The defaults for these options are FIXED(DECIMAL) and CASE(UPPER).

For more information on how to specify these options, see the Programming Guide.

Preprocessor Scan

The preprocessor starts its scan at the beginning of the preprocessor input and scans each character sequentially.

By default the CASE(UPPER) option is in effect, and the preprocessor converts lowercase characters in the input (except for those in comments and string constants) to uppercase. But if the CASE(ASIS) suboption is in effect, the text will be left as is.

Preprocessor Statements

Preprocessor statements are executed when encountered. You can:

- Define preprocessor names using the %DECLARE statement and appearance as a label prefix.

  If a preprocessor variable is not explicitly declared, a diagnostic message is issued and the variable is given the default attribute of CHARACTER. However, the variable is not activated for replacement unless it appears in a subsequently executed %ACTIVATE statement. The variable can be referenced in preprocessor statements.
- Activate an identifier using the %DECLARE or %ACTIVATE statement, thus initiating replacement activity, as described below under "Input Text" on page 502.
- Deactivate an identifier using the %DEACTIVATE statement, thus terminating replacement activity.
- Generate a message in the compiler listing using the %NOTE statement.
- Include string of characters into the preprocessor input.

- Cause the preprocessor to continue the scan at a different point in the preprocessor input using the %GOTO, %IF, %null, %DO, or %END statement.
- Change values of preprocessor variables using the %assignment or %DO statement.
- Define preprocessor procedures using the %PROCEDURE, %RETURN, and %END statements. A preprocessor procedure can be invoked by a function reference in a preprocessor expression, or, if the function procedure name is active, by encountering a function reference in the preprocessor scan of input text.

Listing Control Statements

Listing control statements that are not contained in a preprocessor procedure are copied into the preprocessor output, each on a line of its own.

Input Text

The input text, after replacement of any active identifiers by new values, is copied into the preprocessor output. Invalid characters (part of a character constant or comment) are replaced with blanks in the preprocessor output. To determine replacements, the input text is scanned for:

- Characters that are not part of this PL/I character set are treated as delimiters and are otherwise copied to this output unchanged.
- PL/I character constants or PL/I comments. These are passed through unchanged from input text to preprocessor output by the preprocessor unless they appear in an argument list to an active preprocessor procedure. However, this can cause mismatches between input and output lines for strings or comments extending over several lines, when the input and output margins are different. This is especially true when V format input is used, since the output is always F format, with margins in columns 2 and 72. The output line numbering in these cases also shows this inevitable mismatch.
- Active Identifiers. For an identifier to be replaced by a new value, the identifier must be first *activated* for replacement. Initially, an identifier can be activated by its appearance in a %DECLARE statement. It can be deactivated by executing a %DEACTIVATE statement, and it can be reactivated by executing a %ACTIVATE or %DECLARE statement.

  An identifier that matches the name of an active preprocessor variable is replaced in the preprocessor output by the value of the variable.

  When an identifier matches the name of an active preprocessor function (either programmer-written or built-in) the procedure is invoked and the invocation is replaced by the returned value.

  Identifiers can be activated with either the RESCAN or the NORESCAN options. If the NORESCAN option applies, the value is immediately inserted into the preprocessor output. If the RESCAN option applies, a rescan is made during which the value is tested to determine whether it, or any part of it, should be replaced by another value. If it cannot be replaced, it is inserted into the preprocessor output; if it can be replaced, replacement activity continues until no further replacements can be made. Thus, insertion of a value into the preprocessor output takes place only after all possible replacements have been made.

Replacement values must not contain % symbols, unmatched quotation marks, or unmatched comment delimiters.

The scan terminates when an attempt is made to scan beyond the last character in the preprocessor input. The preprocessor output is then complete and compilation can begin.

Preprocessor Variables and Data Elements

A preprocessor variable is specified in a %DECLARE statement with either the FIXED or the CHARACTER attribute. No other attributes can be declared for a preprocessor variable. (Other attributes are supplied by the preprocessor, however.) All variables have storage equivalent to the STATIC storage class.

Preprocessor data types are coded arithmetic and string data, and are either:

FIXED

A preprocessor variable declared with the FIXED attribute is, by default, given the attributes DECIMAL(5,0).

If the FIXED(BINARY) is in effect, then it is given the attributes BINARY(31,0).

Fractional values are not supported.

CHARACTER

A preprocessor variable declared with the CHARACTER attribute is given the VARYING attribute.

The preprocessor also supports X character string constants.

String repetition factors are not allowed for character constants. However, the COPY built-in function may be used to replicate a constant.

BIT

There are no preprocessor bit variables. However, bit constants are allowed, and bit values result from comparison operators, the concatenation operator (when used with bit operands), the *not* operator, and the PARMSET built-in function. The preprocessor-expression in the %IF statement converts to a bit value.

Preprocessor References and Expressions

Preprocessor references and expressions are written and evaluated in the same way as described in Chapter 4, "Expressions and references," with the following additional comments:

- The operands of a preprocessor expression can consist only of preprocessor variables, references to preprocessor procedures, fixed decimal constants, bit constants, character constants, and references to preprocessor built-in functions.
- While an array may be declared outside of a preprocessor procedure (so that it can be shared across multiple procedures), it may not be referenced outside a procedure (except as the first argument to one of the array-enquiry built-in functions).
- The exponentiation symbol (**) cannot be used.
- Under the FIXED(DECIMAL) option:
  - For arithmetic operations, only decimal arithmetic of precision (5,0) is performed; that is, each operand is converted to a decimal fixed-point integer value of precision (5,0) before the operation is performed, and the decimal fixed-point result is converted to precision (5,0). For example, the expression 3/5 evaluates to 0, rather than to 0.6.

    Any character value being converted to an arithmetic value must be in the form of an optionally signed integer. A null string converts to 0.

    – The conversion of a fixed-point value to a bit value always results in a string of length CEIL(3.32*5), that is, 17.

    – The conversion of a fixed-point value to a character value always results in a string of length 8 and has the same value that would result from converting a FIXED DEC(5,0) value to CHARACTER in a PL/I program.

- Under the FIXED(BINARY) option:

    – For arithmetic operations, only binary arithmetic of precision (31,0) is performed; that is, each operand is converted to a binary fixed-point integer value of precision (31,0) before the operation is performed, and the binary fixed-point result is converted to precision (31,0). For example, the expression 3/5 evaluates to 0, rather than to 0.6.

      Any character value being converted to an arithmetic value must be in the form of an optionally signed integer. A null string converts to 0.

    – The conversion of a fixed-point value to a bit value always results in a string of 31.

    – The conversion of a fixed-point value to a character value results in a string of varying length because leading blanks are trimmed.

Scope of Preprocessor Names

The scope of a preprocessor name is determined by where it is declared. The scope of a name declared within a preprocessor procedure is that procedure. The scope of a name declared within an included string is that string and all input text scanned after that string is included (except any preprocessor procedure in which the name is also declared). The scope of any other name is the entire preprocessor input (except any preprocessor procedure in which the name is also declared).

Preprocessor Procedures

A preprocessor procedure is delimited by %PROCEDURE and %END statements. If the procedure is not defined with a RETURNS attribute, then it may not contain ANSWER statements, but it must not contain any RETURN statements. Conversely, if the procedure is a function, then it must contain at least one RETURN statement, and it must not contain any ANSWER statements.

The statements and groups that can be used within a preprocessor procedure are:

- The preprocessor ANSWER statement.
- The preprocessor assignment statement.
- The preprocessor DECLARE statement.
- The preprocessor DO-group.
- The preprocessor GO TO statement. (A GO TO statement appearing in a preprocessor procedure cannot transfer control to a point outside of that procedure.)
- The preprocessor IF statement.
- The preprocessor ITERATE statement.
- The preprocessor LEAVE statement.
- The preprocessor null statement.
- The preprocessor NOTE statement.
- The preprocessor REPLACE statement.
- The preprocessor RETURN statement.
- The preprocessor SELECT-group.

- The %PAGE, %SKIP, %PRINT, and %NOPRINT listing control statements.

Preprocessor statements in a preprocessor procedure do not begin with a percent symbol.

Preprocessor procedures cannot be nested. A preprocessor ENTRY declaration is not permitted in a preprocessor procedure.

A preprocessor procedure entry name, together with the arguments to the procedure, is called a *function reference.* A preprocessor procedure can be invoked by a function reference in a preprocessor expression, or, if the function procedure name is active, by encountering a function reference in the preprocessor scan of input text. Preprocessor procedure entry names need not be specified in %DECLARE statements.

Provided its entry name is active, a preprocessor procedure need not be scanned before it is invoked. It must, however, be present either in:
- The preprocessor input
- A string included prior to the point of invocation

The result of a preprocessor procedure reference encountered before that procedure is incorporated into the preprocessor input is undefined.

The value returned by a preprocessor function (that is, the value of the preprocessor expression in the RETURN statement) replaces the function reference and its associated argument list in the preprocessor output.

Arguments and Parameters for Preprocessor Procedures

The number of arguments in the procedure reference and the number of parameters in the %PROCEDURE statement need not be the same. The arguments are evaluated before any match is made with the parameter list. If there are more positional arguments than parameters, the excess arguments on the right are ignored. (For an argument that is a function reference, the function is invoked and executed, even if the argument is ignored later.) Parameters that are not set by the function reference are given values of zero, for FIXED parameters, or the null string, for CHARACTER parameters.

Parameters should not be set more than once by a function reference. However, if the value of a parameter is specified more than once, for example both by its position and by keyword, the error is diagnosed and the leftmost setting is used for the invocation.

If the function reference appears in a preprocessor statement, the arguments are associated with the parameters in the normal fashion. Dummy arguments can be created and the arguments converted to the attributes of the corresponding parameters, in the same manner as described under "Passing arguments to procedures" on page 108.

If the function reference appears in input text, dummy arguments are always created. The arguments are interpreted as character strings and are delimited by a comma or right parenthesis. A comma or right parenthesis does not act as a delimiter, however, if it appears between matching parentheses, single quotes, or comment delimiters. For example, the positional argument list (A(B,C),D) has two arguments, namely, the string A(B,C) and the string D. Blanks in arguments

Arguments and parameters

(including leading and trailing blanks) are significant but, if such blanks extend to the end of a line and are not enclosed in quotes or comment delimiters, they are replaced by one blank.

When a function reference is encountered in input text, each argument is scanned for possible replacement activity. This replacement activity has no effect on the number of arguments passed to the function. Any commas or parentheses introduced into arguments by replacement activity are not treated as delimiters, but simply as characters in the argument. If keyword invocation is used, the keywords themselves are not eligible for replacement activity. After all replacements are made, each resulting argument is converted to the type indicated by the corresponding parameter attribute in the preprocessor procedure statement for the function entry name.

%PROCEDURE Statement

The %PROCEDURE statement is used in conjunction with a %END statement to delimit a preprocessor procedure. The syntax for the %PROCEDURE statement is:
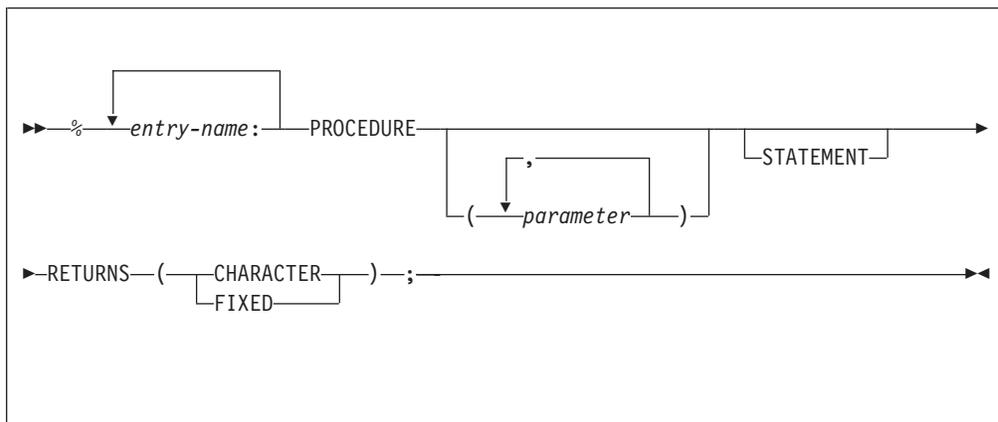
Abbreviation: %PROC

parameter
specifies a parameter of the function procedure.

STATEMENT
If the reference occurs in input text and the STATEMENT option is present:

- The arguments can be specified either in the positional argument list or by keyword reference.
- The end of the reference must be indicated by a semicolon. The semicolon is not retained when the replacement takes place.

For example, a preprocessor procedure headed by:

```
%FIND:PROC(A,B,C) STATEMENT...;
```

must be invoked from a preprocessor expression by a reference of the form:

```
FIND(arg1,arg2,arg3)
```

If the reference is in input text, the procedure can be invoked by any of the following references (or similar ones), all of which have the same result:

```
FIND(X,Y,Z);

FIND B(Y) C(Z) A(X);

FIND(X) C(Z) B(Y);

FIND(,Y,Z) A(X);
```

RETURNS
  The attribute CHARACTER or FIXED must be specified in the RETURNS attribute list to specify the attribute of the value returned by the function procedure.

Preprocessor RETURN Statement

The preprocessor RETURN statement can be used only in a preprocessor procedure and only when the procedure has the RETURNS attribute, and it therefore, can have no leading %. It returns a value as well as control back to the point from which the preprocessor procedure was invoked. At least one RETURN statement must appear in each preprocessor procedure that has the RETURNS attribute.

The value returned by a preprocessor function procedure to the point of invocation is specified by the preprocessor-expression in a RETURN statement in the procedure. The syntax of the preprocessor RETURN statement is:

```
►►──┬──────────┬──RETURN──(──preprocessor-expression──)──;──────────►◄
    │ ◄──────┐ │
    └─▼─label:─┘
```
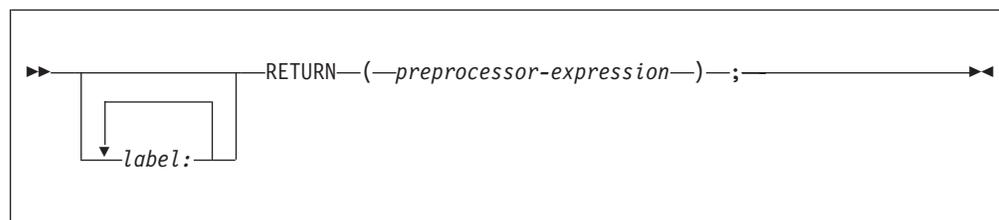
preprocessor-expression
  The value is converted to the RETURNS attribute specified in the %PROCEDURE statement before it is passed back to the point of invocation.

Preprocessor ANSWER Statement

The preprocessor ANSWER statement can be used only in a preprocessor procedure that does not have the RETURNS attribute.

The ANSWER statement produces text and/or invokes other preprocessor procedures. The answered text replaces the invocation of the preprocessor procedure in the source text. You can use any number of ANSWER statements in a preprocessor procedure.

ANSWER
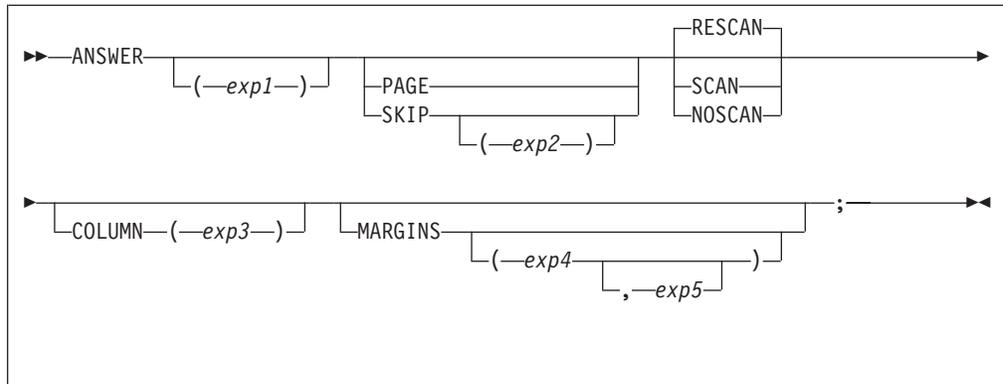
Abbreviations: ANS for ANSWER, COL for COLUMN, MAR for MARGINS

exp1

Represents a character expression that represents the ANSWER text. The ANSWER text can be either a single character string constant or a preprocessor expression of any complexity.

If it is an expression, the expression evaluation occurs in the usual manner and the result is converted to a single character string.

If SCAN or RESCAN is in effect, the character string is scanned for replacements and preprocessor procedure invocations. This replacement is done within the scope of the preprocessor procedure and **not** in the scope into which the answered text is returned. The answered text is then inserted into the source at the point of the preprocessor invocation. After the text is returned into the source, it is not scanned for any replacement activity.

Replacement activity in the string follows the same rules as those for source text scanning and replacement. See "Example" on page 509.

PAGE

Forces the answer text to be placed on a new page of the output source by generating a %PAGE directive.

SKIP

Forces the answer text to be placed on a new line of the output source. The value of *exp2* represents the arithmetic expression specifying the number of lines to be skipped. If *exp2* is not specified, the default value is 1.

RESCAN

Allows eligible preprocessor identifiers to be replaced once or multiple times depending on their scanning status (SCAN or RESCAN).

NOSCAN

Inhibits replacement of eligible preprocessor identifiers.

SCAN

Allows eligible preprocessor identifiers to be replaced only once regardless of their scanning status, that is, SCAN and RESCAN status of an identifier is treated as SCAN.

COLUMN

Specifies the starting column in the source program line in which the answer text is placed. The value of *exp3* represents the arithmetic expression for the column number of the source program line where the answer text starts.

MARGINS

Specifies where the output text is placed within the output record. The value of

exp4 represents the arithmetic expression for the left margin for the output text. The value of *exp5* represents the arithmetic expression for the right margin for the output text.

The values specified for *exp5* must be within the range returned by the MACLMAR (left margin) and MACRMAR (right margin) built-in functions.

If you do not specify the MARGINS option for an ANSWER statement, the default value is MARGINS(MACLMAR,MACRMAR); if you specify the MARGINS option with no operands, the default value is MARGINS(MACCOL,MACRMAR).

You must not use both the RETURN statement with an expression and the ANSWER statement within the same preprocessor procedure.

Example

```
%dcl (Expression, Single_string) entry;
%dcl (Deactivated_macro, Statement_function) entry;
%dcl Deactivated_variable character;
%deact Deactivated_variable, Deactivated_macro;
%Deactivated_variable = '** value of deactivated variable **';

%Deactivated_macro: procedure returns( character );
   return( '** value of deactivated macro **' );
%end;

%Statement_function: procedure( key1 ) stmt returns( fixed );
   dcl key1 fixed;
   return( key1 + key1 );
%end;

%Expression: procedure;
   ANS( Counter ) skip;
   ANS( Deactivated_macro ) skip;
   ANS( Deactivated_variable ) skip;
   /* The following is invalid:                           */
   /* ANS( Statement_function Key1(7););                   */
%end;

%Single_string: procedure;
   ANS( 'Counter' ) skip;
   ANS( 'Deactivated_macro' ) skip;
   ANS( 'Deactivated_variable' ) skip;
   ANS( 'Statement_function Key1( 7 );' ) skip;
%end;


Expression              /* Generates:                      */
                        /*   00001                         */
                        /*   ** value of deactivated macro **   */
                        /*   ** value of deactivated variable ** */

Single_string           /* Generates:                      */
                        /*   Counter                       */
                        /*   Deactivated_macro             */
                        /*   Deactivated_variable          */
                        /*        14                       */
```

Preprocessor Built-In Functions

A function reference can invoke one of a set of predefined functions called *preprocessor built-in functions.* These built-in functions are invoked in the same way that programmer-defined functions are invoked, except that they must be invoked with the correct number of arguments.

Preprocessor built-in functions

The preprocessor built-in functions are:

COLLATE	HBOUND	MAX	SYSPARM
COMMENT	INDEX	MIN	SYSTEM
COMPILEDATE	LBOUND	PARMSET	SYSVERSION
COMPILETIME	LENGTH	QUOTE	TRANSLATE
COPY	MACCOL	REPEAT	VERIFY
COUNTER	MACLMAR	SUBSTR	
DIMENSION	MACRMAR		

The preprocessor executes a reference to a preprocessor built-in function in input text only if the built-in function name is active. The built-in functions can be activated by a %DECLARE or %ACTIVATE statement.

In preprocessor statements, the preprocessor built-in function names are always active as built-in functions unless they are declared with some other meaning.

If a preprocessor built-in function name is used as the name of a user-defined preprocessor procedure, references to the name are references to the procedure, not to the built-in function. In such cases, the identifiers must be declared with the BUILTIN attribute when the built-in function is to be used within a preprocessor procedure.

The following preprocessor built-in functions do not require arguments and must not be given a null argument:

COLLATE	COUNTER	MACRMAR	SYSTEM
COMPILEDATE	MACCOL	SYSPARM	SYSVERSION
COMPILETIME	MACLMAR		

COLLATE

COLLATE returns a CHARACTER string of length 256 comprising the 256 possible character values one time each in the collating order.

```
►►──COLLATE──────────────────────────────────►◄
```

COMMENT

COMMENT converts a CHARACTER expression into a comment.

```
►►──COMMENT(x)────────────────────────────────►◄
```

x Expression that is to be converted to a comment.

x should have CHARACTER type, and if not, it is converted thereto.

x is enclosed with a /* and an */.

If *x* contains /* or */ composite symbols, they are replaced by /> and </, respectively.

COMPILEDATE

COMPILEDATE returns a CHARACTER string of length 17 containing the date and the time of the compilation.

```
►►──COMPILEDATE──────────────────────────────────────────────────►◄
```

The format of the string returned by COMPILEDATE is:

yyyy   current year
mm   current month
dd   current day
hh   current hour
mm   current minute
ss   current second
ttt   current millisecond

The time zone and accuracy are system dependent.

The following example shows how to print the string returned by COMPILEDATE when your program is run:

```
%DECLARE COMP_DATE CHAR;
%COMP_DATE=QUOTE(COMPILEDATE);
PUT EDIT (COMP_DATE) (A);
```

COMPILETIME

COMPILETIME returns a CHARACTER string of length 18 containing the date and the time of compilation.

```
►►──COMPILETIME──────────────────────────────────────────────────►◄
```
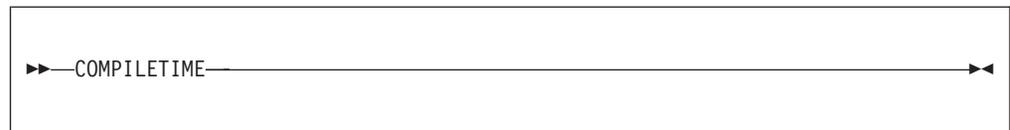
The format of the string returned by COMPILETIME is:

DD   Day of the month
.   Period
MMM   Month in the form JAN, FEB, MAR, etc.
.   Period
YY   Year
b   Blank
HH   Hour
.   Period
MM   Minute
.   Period
SS   Second

A leading zero in the day of the month field is replaced by a blank; no other leading zeros are suppressed.

COMPILETIME

If no timing facility is available, the last 8 characters of the returned string are set to 00.00.00.

The following example shows how to print the string returned by COMPILETIME when your program is executed:

```
%DECLARE COMP_TIME CHAR;
%COMP_TIME=QUOTE(COMPILETIME);
PUT EDIT (COMP_TIME) (A);
```

COPY

COPY returns a CHARACTER string consisting of *y* concatenated copies of the string *x*.
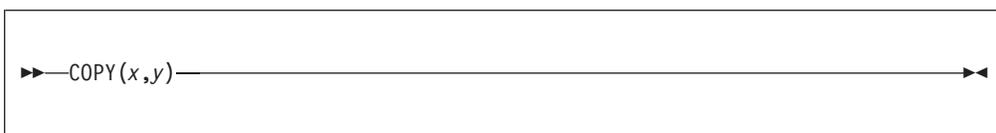
```
►►──COPY(x,y)──────────────────────────────────►◄
```

x Expression.

x should have CHARACTER type, and if not, it is converted thereto.

y Expression that specifies the number of repetitions. *y* should have FIXED type, and if not, it is converted thereto.

y must be nonnegative.

If *y* is zero, the result is a null string.

COUNTER

COUNTER returns a CHARACTER string of length 5 containing a decimal number. The returned number is 00001 for the first invocation, and is incremented by one on each successive invocation.

```
►►──COUNTER──────────────────────────────────────►◄
```
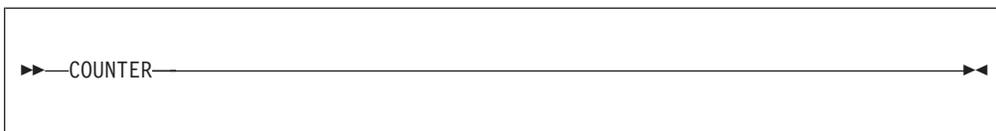
If COUNTER is invoked 99999 times, the next time it is invoked, a diagnostic message is issued and 00000 is returned. The next invocation after that is treated as the first.

The COUNTER built-in function can be used to generate unique names, or for counting purposes.

DIMENSION

DIMENSION returns a FIXED value specifying current extent of dimension *y* of *x*.
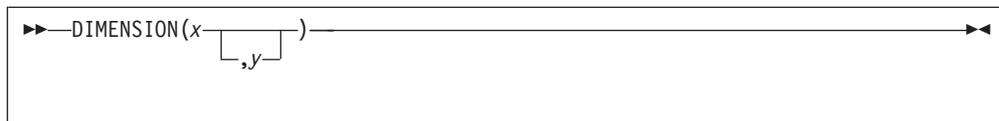
```
►►──DIMENSION(x────────)──────────────────────────────────────────►◄
                  └─,y─┘
```

Abbreviation: DIM

x   Array reference. *x* must not have less than *y* dimensions.

y   Expression specifying a particular dimension of *x*.

   y should have FIXED type, and if not, it will be converted thereto.

   y must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

   y can be omitted only if the array is one-dimensional.

HBOUND

HBOUND returns a FIXED value specifying current upper bound of dimension *y* of *x*.
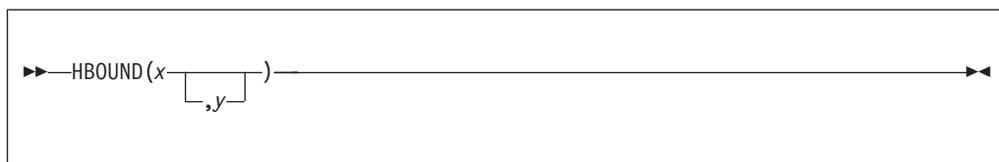
```
►►──HBOUND(x────────)──────────────────────────────────────────►◄
              └─,y─┘
```

x   Array reference. *x* must not have less than *y* dimensions.

y   Expression specifying a particular dimension of *x*.

   y should have FIXED type, and if not, it will be converted thereto.

   y must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

   y can be omitted only if the array is one-dimensional.

INDEX

INDEX returns a FIXED value indicating the starting position within *x* of a substring identical to *y*. You can also specify the location within *x* where processing begins.

```
►►──INDEX(x,y────────)──────────────────────────────────────────►◄
               └─,n─┘
```
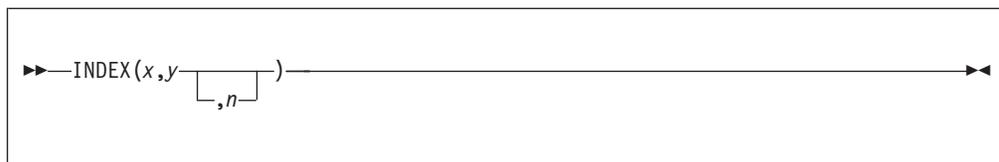
x   Expression to be searched.

   x should have CHARACTER type, and if not, it will be converted thereto.

y   Target expression of the search.

   y should have CHARACTER type, and if not, it will be converted thereto.

n   *n* specifies the location within *x* at which to begin processing.

   n should have FIXED type, and if not, it will be converted thereto.

If *y* does not occur in *x*, or if either *x* or *y* have zero length, the value zero is returned.

n must be greater than 0 and no greater than 1 + LENGTH(*x*).

If *n* = LENGTH(*x*) + 1, the result is zero.

LBOUND

LBOUND returns a FIXED value specifying current lower bound of dimension *y* of *x*.
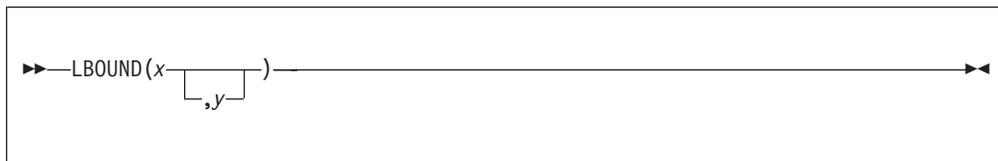
```
►►──LBOUND(x──────────)──────────────────────────────────►◄
              └─,y─┘
```

x   Array reference. *x* must not have less than *y* dimensions.

y   Expression specifying a particular dimension of *x*.

  y should have FIXED type, and if not, it will be converted thereto.

  y must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

  y can be omitted only if the array is one-dimensional.

LENGTH

LENGTH returns a FIXED value specifying the current length of a given character expression x.

```
►►──LENGTH──(──x──)───────────────────────────────────────►◄
```
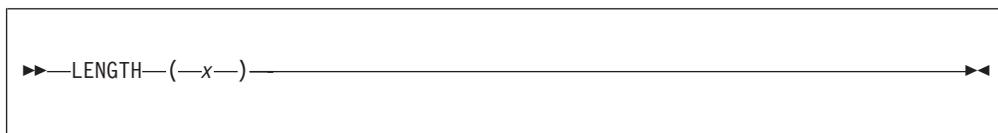
x   Expression.

  x should have CHARACTER type, and if not, it is converted thereto.

MACCOL

MACCOL returns a FIXED value that represents the column where the outermost macro invocation starts in the source text that contains the macro invocation.

```
►►──MACCOL───────────────────────────────────────────────►◄
```
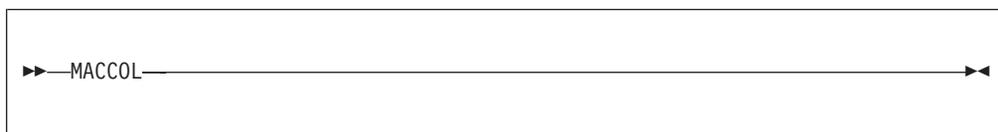
The value returned is not affected by nested macro invocations.

MACLMAR

MACLMAR returns a FIXED value that represents the column number of the left source margin in MARGINS compiler option.

```
►►──MACLMAR──────────────────────────────────────────────────────►◄
```

See the MARGINS option in the Programming Guide.

MACRMAR

MACRMAR returns a FIXED value that represents the column number of the right source margin in MARGINS compiler option.

```
►►──MACRMAR──────────────────────────────────────────────────────►◄
```

See the MARGINS option in the Programming Guide.

MAX

MAX returns the largest value from a set of two or more expressions.

```
              ┌─,─┐
►►──MAX(─x,─▼─y─┴─)──────────────────────────────────────────►◄
```

x and y
    Expressions.

All the arguments should be FIXED, and any that are not FIXED are converted thereto.

MIN

MIN returns the smallest value from a set of two or more expressions.

```
              ┌─,─┐
►►──MIN(─x,─▼─y─┴─)──────────────────────────────────────────►◄
```

x and y
    Expressions.

All the arguments should be FIXED, and any that are not FIXED are converted thereto.

PARMSET

PARMSET returns a BIT value indicating if a specified parameter was set on invocation of the procedure.

PARMSET

```
►►──PARMSET──(──x──)──────────────────────────────────────►◄
```

x   Must be a parameter of the preprocessor procedure.

The PARMSET built-in function can be used only within a preprocessor procedure.

PARMSET returns a bit value of '1'B if the parameter x was explicitly set by the function reference which invoked the procedure, and a bit value of '0'B if it was not—that is, if the corresponding argument was omitted from the function reference in a preprocessor expression, or was the null string in a function reference from input text.

PARMSET can return '0'B, even if a matching argument does appear in the reference, but the reference is in another preprocessor procedure, as follows:

- If the argument is not itself a parameter of the invoking procedure, PARMSET returns the value '1'B.
- If the argument is a parameter of the invoking procedure, PARMSET returns the value for the specified parameter when the invoking procedure was itself invoked.

QUOTE

QUOTE returns a CHARACTER string that represents *x* as a valid quoted string.

```
►►──QUOTE(x)──────────────────────────────────────────────►◄
```

x   Expression that is converted to a quoted string.

      x should have CHARACTER type, and if not, it is converted thereto.

If *x* contains single quotation marks, each is replaced by two consecutive single quotation marks.

REPEAT

REPEAT returns a CHARACTER string consisting of (*y* + 1) concatenated copies of the string *x*.

```
►►──REPEAT(x,y)───────────────────────────────────────────►◄
```

x   Expression.

      x should have CHARACTER type, and if not, it is converted thereto.

y   Expression that specifies the number of repetitions. *y* should have FIXED type, and if not, it is converted thereto.

y must be nonnegative.

If *y* is zero, the result is *x* (converted to character as necessary).

SUBSTR

SUBSTR returns a substring, specified by *y* and *z*, of *x*.

```
►►──SUBSTR(x,y─────────)──────────────────────────────────►◄
              └─,z─┘
```
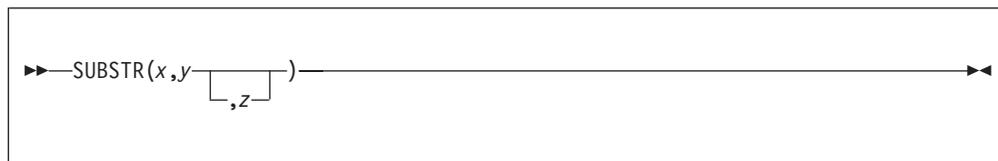
x   Expression specifies the string from which the substring is extracted.

   x should have CHARACTER type, and if not, it is converted thereto.

y   Expression that specifies the starting position of the substring in *x*.

   y should have FIXED type, and if not, it is converted thereto.

z   Expression that specifies the length of the substring in *x*.

   z should have FIXED type, and if not, it is converted thereto.

   If *z* is zero, a null string is returned. If *z* is omitted, the substring returned is position *y* in *x* to the end of *x*.

z must be nonnegative, and the values of *y* and *z* must be such that the substring lies entirely within the current length of *x*.

If $y = \text{LENGTH}(x)+1$ and $z = 0$, then the null string is returned.

SYSPARM

SYSPARM returns the CHARACTER string value of the SYSPARM compiler option.

```
►►──SYSPARM──────────────────────────────────────────────►◄
```
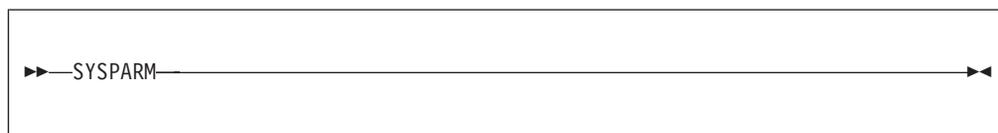
The value returned is not translated to uppercase; the exact value as specified in the compiler option is returned. For more information, see the description of the SYSPARM compiler option in the Programming Guide.

SYSPARM allows information external to the program to be accessed without modifying the source program.

SYSTEM

SYSTEM returns a CHARACTER string that contains the value of the SYSTEM compiler option that is in effect.

```
►►──SYSTEM───────────────────────────────────────────────►◄
```
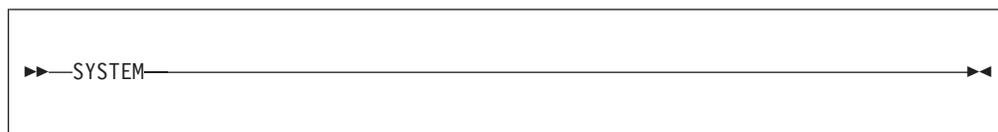
For more information, see the description of the SYSTEM compiler option in the Programming Guide.

SYSVERSION

SYSVERSION returns a CHARACTER string containing the product name as well as the version, release, and modification level.

```
►►──SYSVERSION─────────────────────────────────────────────►◄
```
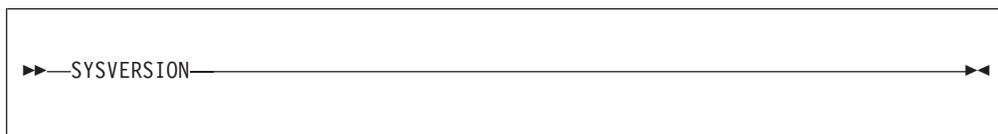
TRANSLATE

TRANSLATE returns a CHARACTER string of the same length as *x*, but with selected characters translated.

```
►►──TRANSLATE(x,y─────────)─────────────────────────────────►◄
                  └─,z─┘
```
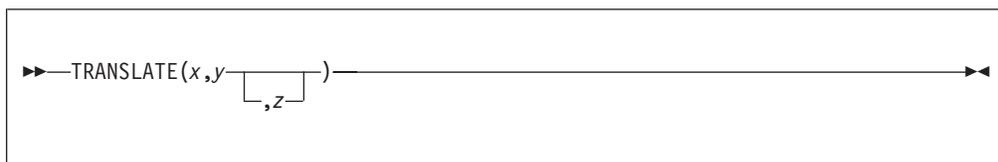
x   Expression to be searched for possible translation of its characters.

     x should have CHARACTER type, and if not, it is converted thereto.

y   Expression containing the translation values of characters.

     y should have CHARACTER type, and if not, it is converted thereto.

z   Expression containing the characters that are to be translated. If *z* is omitted, it defaults to COLLATE.

     z should have CHARACTER type, and if not, it is converted thereto.

TRANSLATE operates on each character of *x* as follows:

If a character in *x* is found in *z*, the character in *y* that corresponds to that in *z* is copied to the result; otherwise, the character in *x* is copied directly to the result. If *z* contains duplicates, the leftmost occurrence is used.

y is padded with blanks, or truncated, on the right to match the length of *z*.

VERIFY

VERIFY returns a FIXED value indicating the position in *x* of the leftmost character that is not in *y*. It also allows you to specify the location within *x* at which to begin processing.

```
►►──VERIFY(x,y─────────)─────────────────────────────────────►◄
              └─,n─┘
```
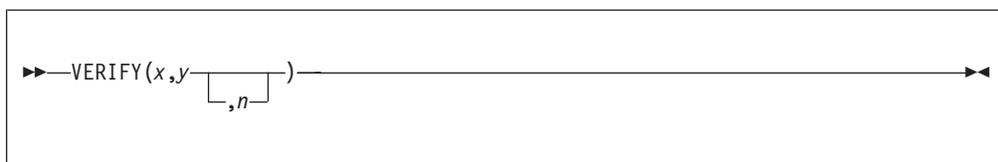
x   Expression.

     x should have CHARACTER type, and if not, it is converted thereto.

y   Expression.

     y should have CHARACTER type, and if not, it is converted thereto.

n   Expression *n* specifies the location within *x* where processing begins.

n should have FIXED type, and if not, it is converted thereto.

If all the characters in *x* do appear in *y*, a value of zero is returned. If *x* is the null string, a value of zero is returned. If *x* is not the null string and *y* is the null string, the value of *n* is returned. The default value for *n* is one.

n must be greater than 0 and no greater than 1 + LENGTH(*x*).

If *n* = LENGTH(*x*) + 1, the result is zero.

Preprocessor Statements

This section lists alphabetically the preprocessor statements and discusses each.

Comments can appear within preprocessor statements wherever blanks can appear. Such comments are not inserted into preprocessor output text.

All preprocessor statements can be labeled.

%ACTIVATE Statement

A %ACTIVATE statement makes an identifier active and eligible for replacement. Any subsequent encounter of that identifier in the input text while the identifier is active initiates replacement activity.
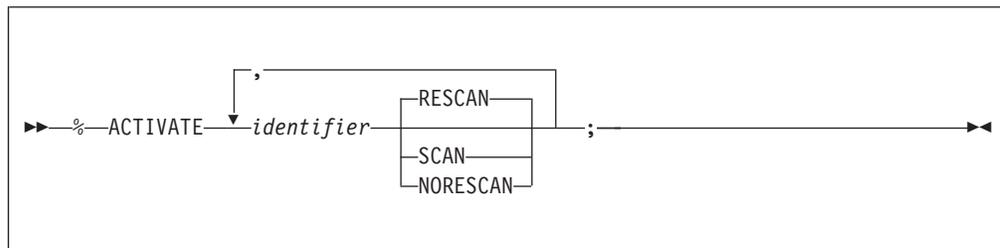
Abbreviation: %ACT

identifier
Specifies the name of a preprocessor variable, a preprocessor procedure, or a preprocessor built-in function.

The identifier should not refer to an array variable.

RESCAN
Specifies that the identifier is replaced as many times as necessary to replace all active identifiers before being placed into the output.

SCAN
Specifies that the identifier is replaced only once before being placed into the output.

NORESCAN
Synonym for SCAN.

Using the %ACTIVATE statement for an identifier that is already active has no effect, except possibly to change the scanning status.

%assignment Statement

A %assignment statement evaluates a preprocessor expression and assigns the result to a preprocessor variable.
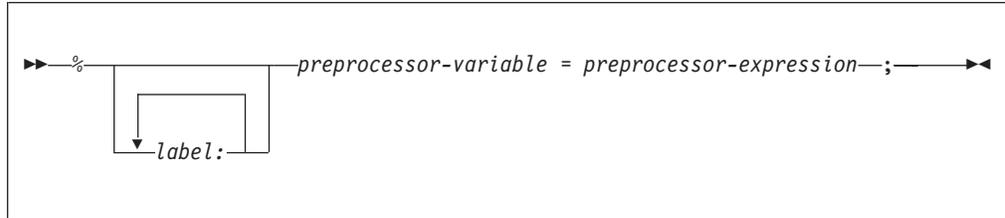
Compound and multiple assignments are not allowed.

The target in an assignment may not be an array, but it may be an array element.

%DEACTIVATE Statement

A %DEACTIVATE statement makes an identifier inactive.
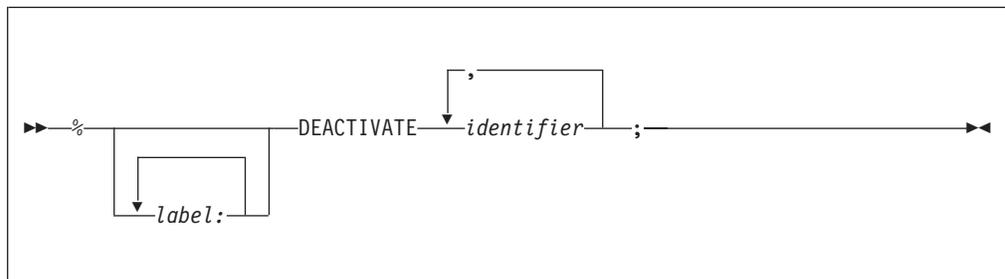
Abbreviation: %DEACT

identifier
Specifies the name of either a preprocessor variable, a preprocessor procedure, or a preprocessor built-in function.

The deactivation of an identifier causes loss of its replacement capability but not its value. Hence, the reactivation of such an identifier need not be accompanied by the assignment of a replacement value.

The deactivation of an identifier does not prevent it from receiving new values in subsequent preprocessor statements.

Deactivation of a deactivated identifier has no effect.

%DECLARE Statement

The %DECLARE statement establishes an identifier as a macro variable, macro procedure, or built-in function. In addition, scanning status can be specified for macro variables.

```
►►─%─DECLARE─┬─────identifier──────────┬─┬─BUILTIN─┬─;──►◄
             │       ┌─,─┐             │ └─ENTRY───┘
             │       ▼   │             │
             │    ┌─,─┐  │             │
             │    ▼   │  │             │
             └─(───identifier───)──────┘
```

Or

```
►►─%─DECLARE─┬──────────────────────┬─;──────────────────►
             │  ┌─,─┐               │
             │  ▼   │               │
             └─ identifier description ─┘

►─┤ identifier description: ├─┬─identifier──────────────────┬───►
                              │            ┌─(─┤ dimension ├─)─┐ │
                              │  ┌─,─┐                          │
                              │  ▼   │                          │
                              └─(───identifier───────────────────)─┘
                                         └─(─┤ dimension ├─)─┘

►─┬─────────────────┬──►◄
  └─┤ attributes ├──┘
```

dimension:

```
├─┬────────────┬─hbound─────────────────────────────────────┤
  │ ┌─,─┐      │
  │ ▼   │      │
  └─lbound─:───┘
  ┌─,─┐
  ▼   │
  └─*─┘
```

attributes:

```
├─┬─CHARACTER─┬─┬──────────┬─┬─NOSCAN──┬────────────────────┤
  └─FIXED─────┘ ├─EXTERNAL─┤ ├─SCAN────┤
               └─INTERNAL─┘ └─RESCAN──┘
```
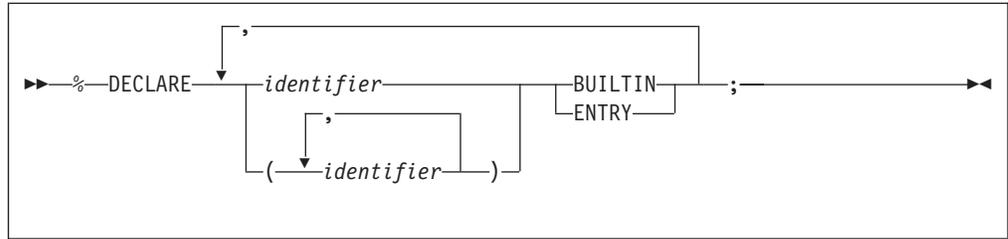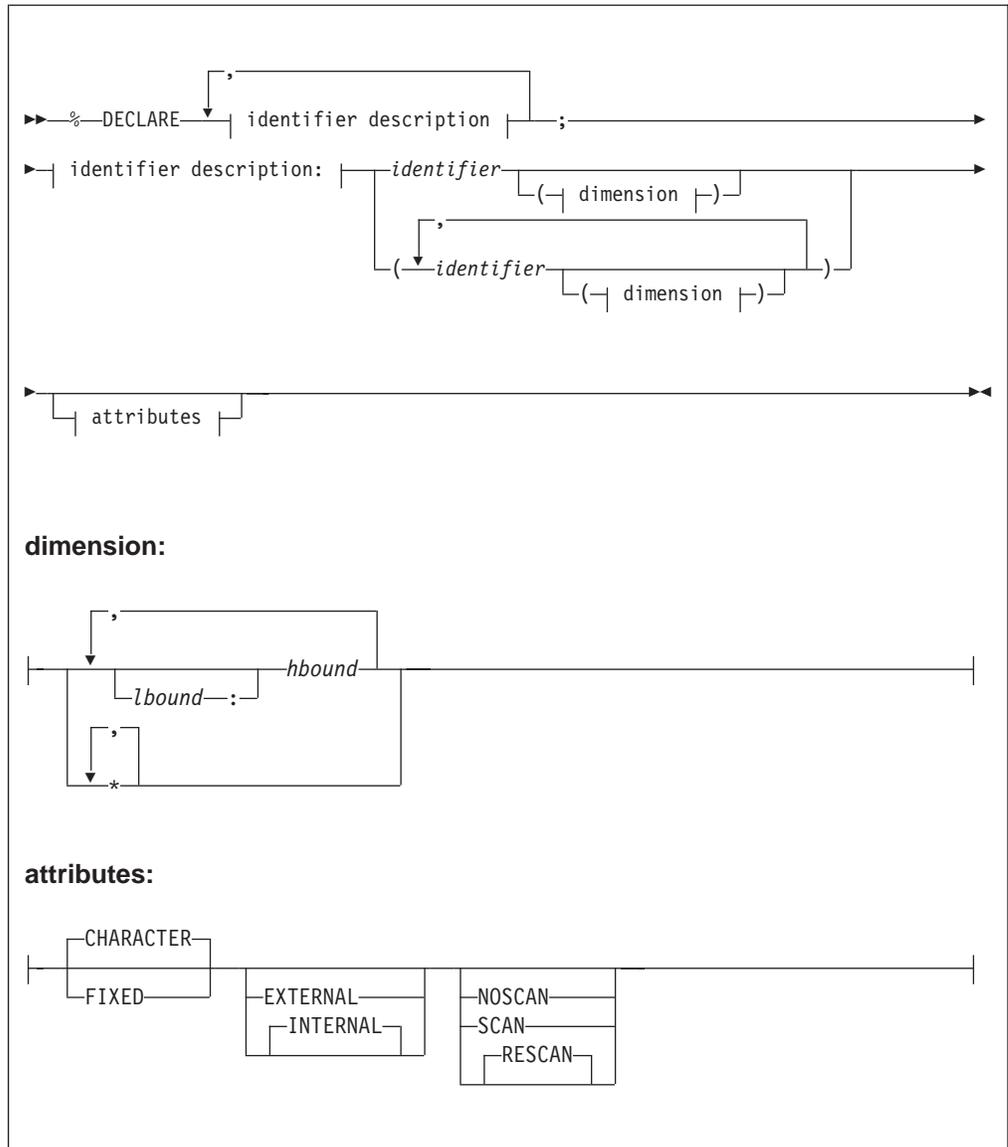
Abbreviations: %DCL for %DECLARE, CHAR for CHARACTER, INT for INTERNAL, EXT for EXTERNAL

identifier description
>   Specifies the names and attributes of macro facility identifiers.

BUILTIN
Specifies that the identifier is the preprocessor built-in function of the same name.

CHARACTER
Specifies that the identifier represents a varying-length character string that has no maximum length.

ENTRY
Specifies that the identifier is a preprocessor procedure.

The declaration activates the entry name.

The declaration of a preprocessor procedure entry name can be performed explicitly by its appearance as the label of a %PROCEDURE statement. This explicit declaration, however, does not activate the preprocessor procedure name.

FIXED
Specifies that the identifier represents an integer.

Under the (default) FIXED(DECIMAL) option, it is also given the attributes DECIMAL(5,0).

Under the FIXED(BINARY) option, it is also given the attributes BINARY(31,0).

RESCAN
Specifies that the identifier is active and is replaced as many times as necessary.

SCAN
Specifies that the identifier is active and is replaced only once in output.

NOSCAN
Specifies that the identifier is inactive and is not to be replaced in output.

dimension
Dimension specification for array variables. No more than 15 dimensions may be specified.

Note: While an array may be declared outside of a preprocessor procedure (so that it can be shared across multiple procedures), it may not be referenced outside a procedure (except as the first argument to one of the array-enquiry built-in functions).

lbound
The desired lower bound for that dimension. The default is 1.

hbound
The desired upper bound for that dimension.

INTERNAL
This attribute is valid only inside a procedure. If specified outside a procedure, a diagnostic message is issued and the variable is given the EXTERNAL attribute.

All variables declared outside a procedure are EXTERNAL, and all variables declared inside a procedure are INTERNAL.

EXTERNAL
This attribute is valid only outside a procedure. If specified inside a procedure, a diagnostic message is issued and the variable is given the INTERNAL attribute.

%DO Statement

The %DO statement, and its corresponding %END statement, delimit a preprocessor DO-group, and can also specify repetitive execution of the DO-group.

The syntax for the %DO statement is described under "DO statement" on page 195.

Note: All the formats of the DO statement are supported except
- UPTHRU and DOWNTHRU are not accepted.
- The "specification" in Type 3 DO statements cannot be specified multiple times.

Preprocessor DO-groups can be nested.

Control cannot transfer to a Type 3 preprocessor DO-group, except by return from a preprocessor procedure invoked from within the DO-group.

Preprocessor statements, input text, and listing control statements can appear within a preprocessor DO-group. The preprocessor statements are executed, and any input text is scanned for possible replacement activity.

%END Statement

The %END statement is used in conjunction with %DO, %SELECT or %PROCEDURE statements to delimit preprocessor DO-groups, SELECT-groups or preprocessor procedures.
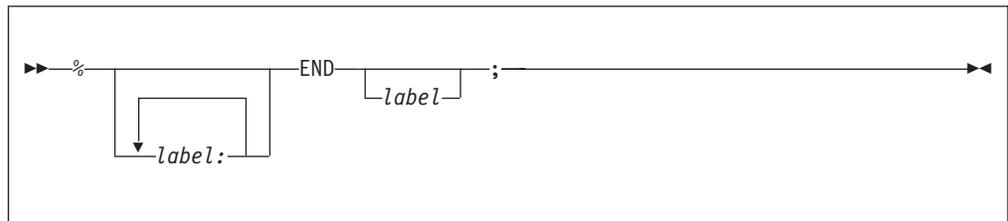
The label following END must be a label of a %PROCEDURE, %DO or %SELECT statement. Multiple closure is allowed.

%GO TO Statement

The %GO TO statement causes the preprocessor to continue its scan at the specified label.
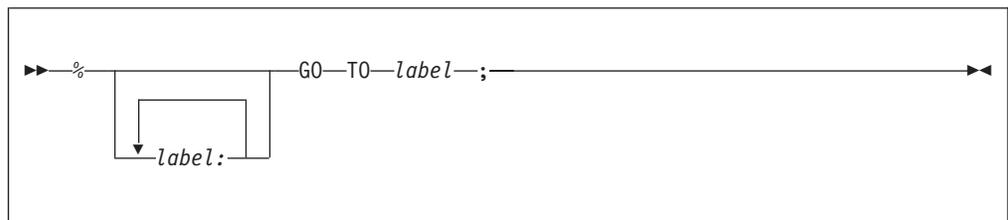
Abbreviation: %GOTO

%GO

The label following the GO TO specifies the point to which the scan is transferred. It must be a label of a preprocessor statement, although it cannot be the label of a preprocessor procedure.

A preprocessor GO TO statement appearing within a preprocessor procedure cannot transfer control to a point outside of that procedure. In other words, the label following GO TO must be contained within the procedure.

See "%INCLUDE Statement" below, for a restriction regarding the use of %GO TO with included strings.

%IF Statement

The %IF statement controls the flow of the scan according to the bit value of a preprocessor expression.
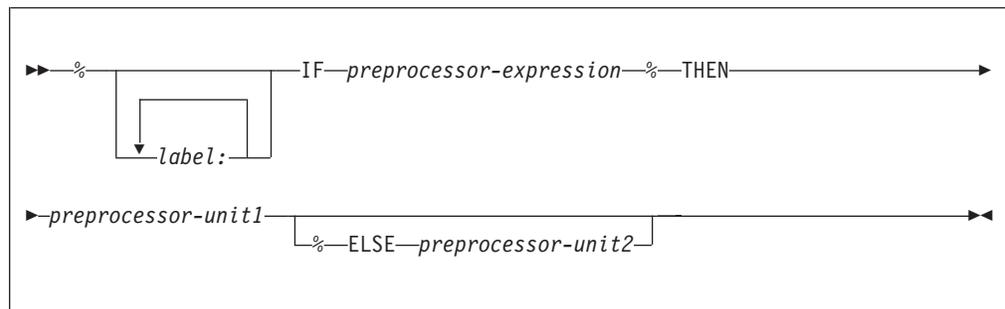
preprocessor-expression
> Is evaluated and converted to a bit string (if the conversion cannot be made, it is an error).

preprocessor-unit
> Is any single preprocessor statement (other than %DECLARE, %PROCEDURE, %END, or %DO) a preprocessor DO-group, or a preprocessor SELECT-group. Otherwise, the description is the same as that given under "IF statement" on page 208.

If any bit in the string has the value '1'B, unit1 is executed and unit2, if present, is ignored; if all bits are '0'B, unit1 is ignored and unit2, if present, is executed.

Scanning resumes immediately following the %IF statement, unless, of course, a %GO TO or preprocessor RETURN statement in one of the units causes the scan to resume elsewhere.

%IF statements can be nested in the same manner used for nesting IF statements, as described under "IF statement" on page 208.

%INCLUDE Statement

The external text specified by a %INCLUDE statement is included into the preprocessor input at the point at which the %INCLUDE statement is executed. Such text, once included, is called *included* text and can consist of preprocessor statements, listing control statements, and PL/I source.

The syntax for the %INCLUDE statement is described under "%INCLUDE directive" on page 210.

Each *dataset* and *member name* pair identifies the external text to be incorporated into the source program.

The scan continues with the first character in the included text. The included text is scanned in the same manner as the preprocessor input. Hence, included text can contribute to the preprocessor output being formed.

%INCLUDE statements can be nested. In other words, included text can contain %INCLUDE statements.

A %GO TO statement in included text can transfer control only to a point within the same include file. The target label in the %GOTO statement must not precede the %GOTO.

Preprocessor statements, DO-groups, SELECT-groups and procedures in included text must be complete. For example, it is not allowable to have half of a %IF statement in an included text and half in another portion of the preprocessor input.

If the preprocessor input and the included text contain no preprocessor statements other than %INCLUDE, execution of the preprocessor can be omitted. (This necessitates the use of the INCLUDE compile-time option.)

For example, assume that PAYRL is a member of the data set SYSLIB and contains the following text (a structure declaration):

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST   CHARACTER (30) VARYING,
            3 FIRST  CHARACTER (15) VARYING,
            3 MIDDLE CHARACTER (3) VARYING,
          2 CURR,
            3 (REGLAR, OVERTIME) FIXED DECIMAL (8,2),
          2 YTD LIKE CURR;
```

Then the following preprocessor statements:

```
%DECLARE PAYROLL CHARACTER;
%PAYROLL='CUM_PAY';
%INCLUDE PAYRL;
%DEACTIVATE PAYROLL;
%INCLUDE PAYRL;
```

generate two structure declarations in the preprocessor output text. The only difference between them is their names, CUM_PAY and PAYROLL.

Execution of the first %INCLUDE statement incorporates the text in PAYRL into the preprocessor input. When the preprocessor scan encounters the identifier PAYROLL in this included text, it replaces it with the current value of the active preprocessor variable PAYROLL, namely, CUM_PAY. Further scanning of the included text results in no additional replacements. The preprocessor scan then encounters the %DEACTIVATE statement and deactivates the preprocessor variable PAYROLL. When the second %INCLUDE statement is executed, the text in PAYRL once again is incorporated into the preprocessor input. This time, however, scanning of the included text results in no replacements whatsoever.

%ITERATE Statement

The %ITERATE statement transfers control to the %END statement that delimits its containing iterative DO-group. The current iteration completes and the next iteration, if needed, is started.

%ITERATE

The ITERATE statement can be specified inside a non-iterative DO-group as long as that DO-group is contained in an iterative DO-group.
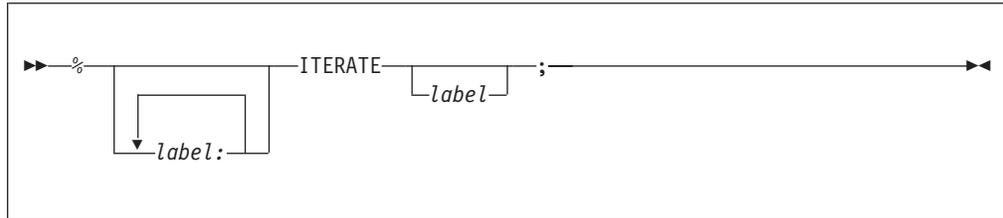
```
►►─%─┬──────────────┬─ITERATE──┬───────┬─;────────────────────►◄
      │  ┌◄────────┐ │          └─label─┘
      └──┴─label:──┴─┘
```

label-constant
   Must be the label of a containing DO-group. If omitted, control transfers to the END statement of the most recent iterative do-group containing the ITERATE statement.

%LEAVE Statement

When contained in or specifying a simple DO-group, the %LEAVE statement terminates the group. When contained in or specifying an iterative DO-group, the %LEAVE statement terminates all iterations of the group, including the current iteration. The flow of control goes to the same point it would normally go to if the do-group had terminated by reaching its END statement.

```
►►─%─┬──────────────┬─LEAVE──┬───────┬─;──────────────────────►◄
      │  ┌◄────────┐ │        └─label─┘
      └──┴─label:──┴─┘
```

label-constant
   Must be a label of a containing DO-group. The DO-group that is left is the DO-group that has the specified label. If *label-constant* is omitted, the DO-group that is left is the group that contains the LEAVE statement.

%NOTE Statement

The %NOTE statement generates a preprocessor diagnostic message of specified text and severity.
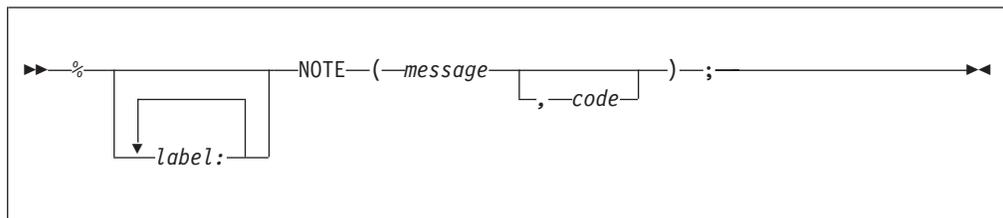
```
►►─%─┬──────────────┬─NOTE──(──message──┬─────────┬──)──;──────►◄
      │  ┌◄────────┐ │                   └─,─code─┘
      └──┴─label:──┴─┘
```

message
   A character expression whose value is the required diagnostic message.

code
   A fixed expression whose value indicates the severity of the message, as follows:

Code	Severity
0	I
4	W
8	E
12	S
16	U

If *code* is omitted, the default is 0.

If *code* has a value other than those listed above, a diagnostic message is produced and a default value is taken. If the value is less than 0 or greater than 16, severity U is the default. Otherwise, the next lower severity is the default.

Generated messages are filed together with other preprocessor messages. Whether or not a particular message is subsequently printed depends upon its severity level and the setting of the compiler FLAG option.

Generated messages of severity U cause immediate termination of preprocessing and compilation. Generated messages of severity S, E, or W might cause termination of compilation, depending upon the setting of the NOSYNTAX and NOCOMPILE compile-time options.

%null Statement

The %null statement does nothing and does not modify sequential statement execution.

```
►►─%─;─────────────────────────────────────────────────────►◄
```
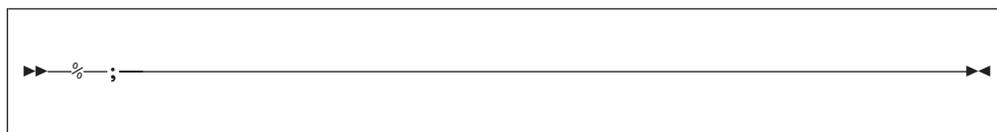
Note: The %PROCEDURE and RETURN statements are described earlier in this chapter.

%REPLACE Statement

The %REPLACE statement allows for the immediate replacement of a name with a string constant, or a numeric constant. The name does not need to be a declared variable to have a value assigned to it.

```
►►─%REPLACE─identifier─┬─BY───┬─┬─string-constant─────┬─►◄
                       └─WITH─┘ └─arithmetic-constant─┘
```
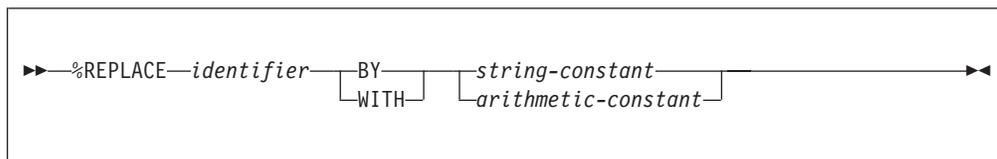
identifier
  Name to be replaced.

string-constant
  The name, if undeclared, will be given the CHARACTER attribute

arithmetic-constant
  The name, if undeclared, will be given the FIXED attribute

> Under the FIXED(DEC) option, the value will be converted to FIXED DEC(5,0).
>
> Under the FIXED(BIN) option, the value will be converted to FIXED BIN(31,0).

%SELECT Statement

The %SELECT statement, and its corresponding %END statement, delimit a preprocessor SELECT-group.
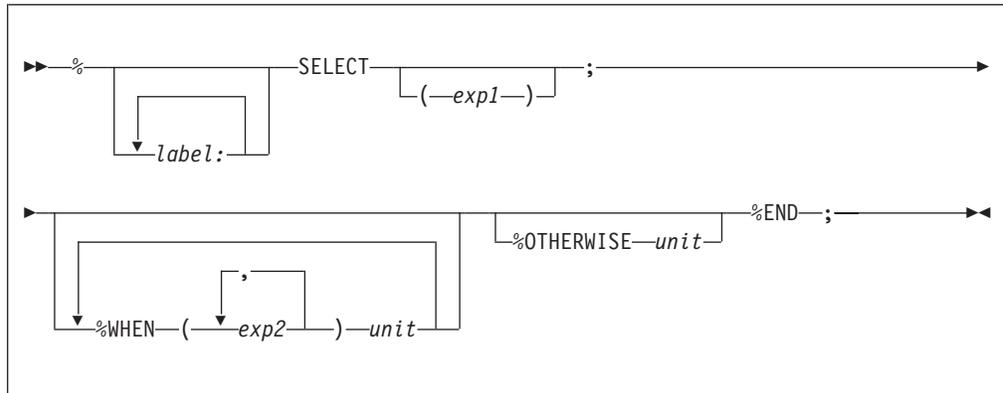
Preprocessor Examples

Example 1

If the preprocessor input contains:

```
%DECLARE A CHARACTER, B FIXED;
%A = 'B+C';
%B = 2;
X = A;
```

the following is inserted into the preprocessor output:

```
X = 2+C;
```

The preprocessor statements activate A and B with the default RESCAN, assign the character string 'B+C' to A, and assign the constant 2 to B.

The fourth line is input text. The current value of A, which is 'B+C', replaces A in the preprocessor output. But this string contains the preprocessor variable B. Upon rescanning B, the preprocessor finds that it has been activated. Hence, the value 2 replaces B in the preprocessor output. The preprocessor variable B has a default precision of (5,0) and, therefore, actually contains 2 preceded by four zeros. When this value replaces B in the string 'B+C' it is converted to a character string and becomes 2 preceded by seven blanks.

Further rescanning shows that 2 cannot be replaced; scanning resumes with +C which, again, cannot be replaced.

If, in the above example, the preprocessor variable A was activated by this statement:

```
%ACTIVATE A NORESCAN;
```

the preprocessor output would be:

```
X = B+C;
```

Example 2
If the preprocessor input contains:

```
%DECLARE I FIXED, T CHARACTER;
%DEACTIVATE I;
%I = 15;
%T = 'A(I)';
S  = I*T*3;
%I = I+5;
%ACTIVATE I;
%DEACTIVATE T;
R  = I*T*2
```

the preprocessor output would be as follows (replacement blanks are not shown):

```
S = I*A(I)*3;
R = 20*T*2;
```

Example 3
This example illustrates how preprocessor facilities can be used to speed up the execution of a DO-group, such as:

```
DO I=1 TO 10;
Z(I)=X(I)+Y(I);
END;
```

The following would accomplish the same thing, but without the requirements of incrementing and testing during execution of the compiled program:

```
%DECLARE I FIXED;
%DO I = 1 TO 10;
Z(I)=X(I)+Y(I);
%END;
%DEACTIVATE I;
```

The third line is input text and is scanned for replacement activity. The first time that this line is scanned, I has the value 1 and has been activated. Therefore, the following is inserted into the preprocessor output:

```
Z(      1)=X(      1)+Y(      1);
```

Each 1 is preceded by seven blanks.

For each increment of I, up to and including 10, the input text is scanned and each occurrence of I is replaced by its current value. As a result, the following is inserted into the preprocessor output:

```
Z(      1)=X(      1)+Y(      1);
Z(      2)=X(      2)+Y(      2);
          .
          .
          .
Z(     10)=X(     10)+Y(     10);
```

When the value of I reaches 11, control falls through to the %DEACTIVATE statement.

Example 4
In the preprocessor input below, VALUE is a preprocessor function procedure that returns a character string of the form 'arg1(arg2)', where *arg1* and *arg2* represent the arguments that are passed to the function:

```
DECLARE (Z(10), Q) FIXED;
%A='Z';
%ACTIVATE A, VALUE;
Q = 6 + VALUE(A,3);
```

```
%DECLARE A CHARACTER;
%VALUE: PROC(ARG1,ARG2) RETURNS(CHAR);
       DCL ARG1 CHAR, ARG2 FIXED;
       RETURN(ARG1||'('||ARG2||')');
       %END VALUE;
```

When the scan encounters the fourth line, A is active and is thus eligible for replacement. Since VALUE is also active, the reference to it in the fourth line invokes the preprocessor function procedure of that name.

However, before the arguments A and 3 are passed to VALUE, A is replaced by its value Z (assigned to A in a previous assignment statement), and 3 is converted to fixed-point to conform to the attribute of its corresponding parameter. VALUE then performs a concatenation of these arguments and the parentheses and returns the concatenated value, that is, the string Z (3), to the point of invocation. The returned value replaces the function reference and the result is inserted into the preprocessor output. Thus, the preprocessor output generated is:

```
DECLARE (Z(10),Q) FIXED;
Q = 6+Z(      3);
```

Example 5

The preprocessor function procedure GEN defined below can generate a GENERIC declaration for up to 99 entry names with up to 99 parameter descriptors in the parameter descriptor lists. Only four are generated in this example.

```
%DCL GEN ENTRY;
DCL A GEN (A,2,5,FIXED);
  %GEN: PROC(NAME,LOW,HIGH,ATTR) RETURNS (CHAR);
DCL (NAME, SUFFIX, ATTR, STRING) CHAR, (LOW, HIGH, I, J) FIXED;
STRING='GENERIC(';
DO I=LOW TO HIGH;                       /* ENTRY NAME LOOP*/
   IF I>9 THEN
      SUFFIX=SUBSTR(I, 7, 2);
                                        /* 2 DIGIT SUFFIX*/
   ELSE SUFFIX=SUBSTR(I, 8, 1);
                                        /* 1 DIGIT SUFFIX*/
   STRING=STRING||NAME||SUFFIX||' WHEN (';
   DO J=1 TO I;                         /* DESCRIPTOR LIST*/
      STRING=STRING||ATTR;
      IF J<I                            /* ATTRIBUTE SEPARATOR*/
        THEN STRING=STRING||',';
        ELSE STRING=STRING||')';
                                        /* LIST SEPARATOR */
   END;
   IF I<HIGH THEN                       /* ENTRY NAME SEPARATOR*/
      STRING=STRING||',';
   ELSE STRING=STRING||')';
                                        /* END OF LIST /*
END;
RETURN (STRING)
% END;
```

The preprocessor output produced is:

```
DCL A GENERIC(A2 WHEN (FIXED,FIXED),
             A3 WHEN (FIXED, FIXED, FIXED),
             A4 WHEN (FIXED, FIXED, FIXED, FIXED),
             A5 WHEN (FIXED, FIXED, FIXED, FIXED, FIXED));
```

Example 6

This example shows a preprocessor procedure that implements a statement of the form:

```
SEARCH TABLE(array) FOR(value)
USING(variable) AND(variable);
```

This statement searches a specified two-dimensional array for a specified value, using specified or default variables for the array subscripts. After execution of the statement, the array subscript variables identify an element that contains the specified value. If no element contains the specified value, both subscript variables are set to -22222.

The preprocessor procedure that implements this statement is:

```
%SEARCH:
 PROC(TABLE,FOR,USING,AND) STATEMENT RETURNS(CHARACTER);

   DECLARE(TABLE,FOR,USING,AND,LABL, DO1,DO2) CHARACTER,
       (PARMSET,COUNTER) BUILTIN;

   IF PARMSET(TABLE) & PARMSET(FOR) THEN;
   ELSE SERR:DO;
   NOTE ('MISSING OR INVALID ARGUMENT(S)'||'FOR ''SEARCH''',4);
   RETURN ('/*INVALID SEARCH STATEMENT*/');
   END;

   IF ¬PARMSET(USING) THEN
     USING='I';
   IF ¬PARMSET(AND)   THEN
     AND='J';
   IF USING = AND  THEN
     GO TO SERR;

   LABL='SL'||COUNTER;
   DO1=LABL||': DO '||USING||'=LBOUND('||TABLE||',1)
     TO HBOUND('||TABLE||',1);';
   DO2='DO '||AND||'=LBOUND('||TABLE||',2)
     TO HBOUND ('||TABLE||',2);';

   RETURN(DO1||DO2||'SELECT('||TABLE
     ||'('||USING||','||AND||'));
   WHEN('||FOR||') LEAVE '||LABL||';
   OTHER;
   END '||LABL||';
   IF '||AND||' > H  BOUND('||TABLE||',2) THEN
     '||USING||','||AND||'.' = -22222;');
 %END SEARCH;
```

The PARMSET built-in function is used to determine which parameters are set when the procedure is invoked. If USING is not set, the default array subscript variable I is used. If AND is not set, J is used. If TABLE or FOR is not set, or if the invocation results in the same variable being used for both subscripts, a preprocessor diagnostic message is issued and a comment is returned in the preprocessor output.

The COUNTER built-in function is used to generate unique labels for the preprocessor output returned by the procedure.

The procedure can be invoked with keyword arguments or positional arguments, or a combination of the two. The following invocations of the procedure produce identical results:

```
SEARCH TABLE(LIST.NAME) FOR('J.DOE') USING(I) AND(J);

SEARCH TABLE(LIST.NAME) FOR('J.DOE');

SEARCH(LIST.NAME) FOR('J.DOE');
```

Preprocessor examples

```
SEARCH(LIST.NAME,'J.DOE');

SEARCH(,'J.DOE') TABLE(LIST.NAME);
```

The preprocessor output returned by any of these invocations is:

```
SL00001:
DO I=LBOUND(LIST.NAME,1) TO HBOUND(LIST.NAME,1);
   DO J=LBOUND(LIST.NAME,2) TO HBOUND(LIST.NAME,2);
      SELECT(LIST.NAME(I,J));
   WHEN('J.DOE') LEAVE SL00001;
   OTHER;
END SL00001;
IF J > HBOUND(LIST.NAME,2) THEN
   I,J = -22222;
```

The label SL00001 is returned only for the first invocation. A new unique label is returned for each subsequent invocation.

Appendix. Limits

Table 67 summarizes the implementation limits for the PL/I language elements. Table 68 on page 536 summarizes the implementation limits for the macro facility language elements.

Table 67. Language element limits

Language Element	Description	Limit
Arrays	Maximum number of dimensions for an array	15
	Minimum lower bound (Note 1)	−2147483648
	Maximum upper bound (Note 1)	+2147483647

Note 1: Under the compile-time option CMPAT(V1), the minimum lower bound is −32768 and the maximum upper bound is 32767. Also, these bounds should be used with caution. For instance, if A has the maximum upper bound and JX has the attributes SIGNED FIXED BIN(31), then the loop DO JX = LBOUND(A) TO HBOUND(A) will ″wrap″ after it hits the last element in the array. It would not ″wrap″ if UPTHRU were used instead of TO.

Language Element	Description	Limit
Structures	Maximum number of levels in a structure	15
	Maximum level-number in a structure	255
Arithmetic Precisions	Maximum precision for FIXED DECIMAL	31 (Note 2)
	Maximum precision for FIXED BINARY	63 (Note 3)
	Maximum precision for FLOAT DECIMAL	33 (Note 4)
	Maximum precision for FLOAT BINARY	109 (Note 5)
	Maximum scale factor for FIXED data	127
	Minimum scale factor for FIXED data	-128

Note 2: This is true only if you specify the compile-time option LIMITS(FIXEDDEC(31)); the default is 15.

Note 3: This is true only if you specify the compile-time option LIMITS(FIXEDBIN(63)); the default is 31.

Note 4: On Intel, the maximum FLOAT DECIMAL precision is 18.

Note 5: On Intel, the maximum FLOAT BINARY precision is 64.

Language Element	Description	Limit
String and AREA Variables or Constants	Maximum length of CHARACTER	32767
	Maximum length of BIT	32767
	Maximum length of GRAPHIC	16383
	Maximum length of WIDECHAR	16383
	Maximum size of AREA	2147483647

Table 67. Language element limits (continued)

Language Element	Description	Limit
Built-In Functions	Maximum number of arguments to the IAND, IOR, MAX, and MIN functions	64
	Maximum values for the precision (p) in the ADD, BINARY, DECIMAL, DIVIDE, FIXED, FLOAT, MULTIPLY, PRECISION, and SUBTRACT functions	same as corresponding limit for arithmetic precision
	Maximum values for the scale (q) in the ADD, BINARY, DECIMAL, DIVIDE, FIXED, MULTIPLY, PRECISION, and SUBTRACT functions	same as corresponding limit for arithmetic precisions
	Maximum number of digits (N) in the CEIL, FLOOR, MAX, MIN, MOD, ROUND and TRUNC functions	same as corresponding limit for arithmetic precisions

Table 67. Language element limits (continued)

Language Element	Description	Limit
Program Size	Maximum length of an identifier	100
	Maximum number of procedures in a program	255
	Maximum number of lexical units (keywords, identifiers, delimiters, etc) before a statement type can be resolved	511
	Maximum number of DEFAULT-statements in a block	31
	Maximum number of %PUSH statements	63
	Maximum number of %INCLUDE statements	2047
	Maximum nesting of %INCLUDE statements	2046
	Maximum number of lines in any source file	1048575
	Maximum number of statements	16777215
	Maximum number of LIKE-attributes in a block	63
	Maximum number of output expressions in a data-list	60
	Maximum number of repetitive DO-specifications in a data-list	50
	Maximum size of a data aggregate containing no unaligned bits	2147483647
	Maximum size of a data aggregate containing some unaligned bits	268435455
	Maximum number of arguments in a CALL or function reference	255
	Maximum number of parameters for a procedure	4095
	Maximum nesting of factored attributes	15
	Maximum nesting of BEGIN and PROCEDURE statements	30
	Maximum nesting of DO-groups	49
	Maximum nesting of IF statements	49
	Maximum nesting of SELECT-statements	49
	Maximum length of %NOTE message	32767

Table 67. Language element limits (continued)

Language Element	Description	Limit
Miscellaneous	Maximum number of picture characters in a character picture	511
	Maximum number of bytes in a numeric picture	253
	Maximum number of numeric picture characters in a numeric picture	31
	Maximum number of bytes in the external representation of CHARACTER, X, BIT, BX, GRAPHIC, GX, WX and M string constants The external representation includes all quotes and string suffixes. For example, the string '01010110'B has 11 bytes in its external specification, but only 1 byte in its internal representation. Similarly, the string 'Ain''t Misbehavin''' has 21 bytes in its external specification, but only 17 in its internal representation.	3072
	Maximum length for a KEYTO character string	120
	Maximum length for a KEYTO graphic or widechar string	60
	Maximum KEY length	32763
	Maximum line size for LINESIZE	32,759 for F-format or U-format, and 32,751 for V-format
	Minimum line size for LINESIZE	1
	Maximum page size for PAGESIZE	32,767
	Minimum page size for PAGESIZE compiler option	1
	Maximum size of DISPLAY character string	126
	Maximum DISPLAY reply message.	72 bytes
	Range of IEEE normalized floating-point numbers	+3.30E-4932 to +1.21E+4932, 0, -3.30E-4932 to -1.21E+4932
	Range of hex floating-point numbers	+10E-78 to +10E75, 0, -10E-78 to +10E+75

Table 68. Macro facility limits

Language Element	Description	Limit
Arrays	Maximum number of dimensions	15
	Minimum lower bound	–32768
	Maximum upper bound	+32767

Table 68. Macro facility limits (continued)

Language Element	Description	Limit
Arithmetic Range	Min and max for a FIXED variable under FIXED(DECIMAL) option	same as FIXED DECIMAL(5) identifier
	Min and max for a FIXED variable under FIXED(BINARY) option	same as FIXED BINARY(31) identifier
Macro Procedures	Maximum nesting level	1
Keys	Maximum number of keyword parameters	4096

Limits

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user⌐s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

> IBM Corporation
> J74/G4
> 555 Bailey Avenue
> San Jose, CA 95141-1099
> U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

> IBM World Trade Asia Corporation
> Licensing
> 2-31 Roppongi 3-chome, Minato-ku
> Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

> INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors.

Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	Language Environment
CICS	MVS
CICS/ESA	OpenEdition
DB2	z/OS
DFSMS	RACF
DFSORT	System/390
IBM	VisualAge
IMS	z/OS
IMS/ESA	

Intel is a registered trademark of Intel Corporation in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and other countries.

Pentium is a registered trademark of Intel Corporation in the United States and other countries.

Unicode is a trademark of the Unicode Consortium.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be the trademarks or service marks of others.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Bibliography

Enterprise PL/I publications

Programming Guide, SC27-1457
Language Reference, SC27-1460
Messages and Codes, SC27-1461
Diagnosis Guide, GC27-1459
Compiler and Run-Time Migration Guide, GC27-1458

PL/I for MVS & VM

Installation and Customization under MVS, SC26-3119
Language Reference, SC26-3114
Compile-Time Messages and Codes, SC26-3229
Diagnosis Guide, SC26-3149
Migration Guide, SC26-3118
Programming Guide, SC26-3113
Reference Summary, SX26-3821

z/OS Language Environment

Concepts Guide, SA22-7567

Debugging Guide, GA22-7560

Run-Time Messages, SA22-7566

Customization, SA22-7564

Programming Guide, SA22-7561

Programming Reference, SA22-7562

Run-Time Migration Guide, GA22-7565

Writing Interlanguage Communication Applications, SA22-7563

CICS Transaction Server

Application Programming Guide, SC33-1687

Application Programming Reference, SC33-1688

Customization Guide, SC33-1683

External Interfaces Guide, SC33-1944

DB2 UDB for OS/390 and z/OS

Administration Guide, SC26-9931

An Introduction to DB2 for OS/390, SC26-9937

Application Programming and SQL Guide, SC26-9933

Command Reference, SC26-9934

Messages and Codes, GC26-9940

SQL Reference, SC26-9944

DFSORT™

Application Programming Guide, SC33-4035

Installation and Customization, SC33-4034

IMS/ESA®

Application Programming: Database Manager, SC26-8015

Application Programming: Database Manager Summary, SC26-8037

Application Programming: Design Guide, SC26-8016

Application Programming: Transaction Manager, SC26-8017

Application Programming: Transaction Manager Summary, SC26-8038

Application Programming: EXEC DL/I Commands for CICS and IMS™, SC26-8018

Application Programming: EXEC DL/I Commands for CICS and IMS Summary, SC26-8036

z/OS MVS

JCL Reference, SA22-7597

JCL User's Guide, SA22-7598

System Commands, SA22-7627

z/OS UNIX System Services

UNIX System Services Command Reference, SA22-7802

UNIX System Services Programming: Assembler Callable Services Reference, SA22-7803

UNIX System Services User's Guide, SA22-7801

z/OS TSO/E

Command Reference, SA22-7782

User's Guide, SA22-7794

z/Architecture

Principles of Operation, SA22-7832

Unicode® and character representation

OS/390 Support for Unicode: Using Conversion Services, SC33-7050

Glossary

This glossary defines terms for all platforms and releases of PL/I. It might contain terms that this manual does not use. If you do not find the terms for which you are looking, see the index in this manual or *IBM Dictionary of Computing*, SC20-1699.

A

access.   To reference or retrieve data.

action specification.   In an ON statement, the ON-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever the appropriate condition is raised.

activate (a block).   To initiate the execution of a block. A procedure block is activated when it is invoked. A begin-block is activated when it is encountered in the normal flow of control, including a branch. A package cannot be activated.

activate (a preprocessor variable or preprocessor entry point).   To make a macro facility identifier eligible for replacement in subsequent source code. The %ACTIVATE statement activates preprocessor variables or preprocessor entry points.

active.   The state of a block after activation and before termination. The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text. The state in which an event variable is said to be during the time it is associated with an asynchronous operation. The state in which a task variable is said to be when its associated task is attached. The state in which a task is said to be before it has been terminated.

actual origin (AO).   The location of the first item in the array or structure.

additive attribute.   A file description attribute for which there are no defaults, and which, if required, must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute*.

adjustable extent.   The bound (of an array), the length (of a string), or the size (of an area) that might be different for different generations of the associated variable. Adjustable extents are specified as expressions or asterisks (or by REFER options for

based variables), which are evaluated separately for each generation. They cannot be used for static variables.

aggregate.   See *data aggregate*.

aggregate expression.   An array, structure, or union expression.

aggregate type.   For any item of data, the specification whether it is structure, union, or array.

allocated variable.   A variable with which main storage is associated and not freed.

allocation.   The reservation of main storage for a variable. A generation of an allocated variable. The association of a PL/I file with a system data set, device, or file.

alignment.   The storing of data items in relation to certain machine-dependent boundaries (for example, a fullword or halfword boundary).

alphabetic character.   Any of the characters A through Z of the English alphabet and the alphabetic extenders #, $, and @ (which can have a different graphic representation in different countries).

alphameric character.   An alphabetic character or a digit.

alternative attribute.   A file description attribute that is chosen from a group of attributes. If none is specified, a default is assumed. Contrast with *additive attribute*.

ambiguous reference.   A reference that is not sufficiently qualified to identify one and only one name known at the point of reference.

area.   A portion of storage within which based variables can be allocated.

argument.   An expression in an argument list as part of an invocation of a subroutine or function.

argument list.   A parenthesized list of zero or more arguments, separated by commas, following an entry name constant, an entry name variable, a generic name, or a built-in function name. The list becomes the parameter list of the entry point.

arithmetic comparison.   A comparison of numeric values. See also *bit comparison, character comparison*.

arithmetic constant.   A fixed-point constant or a floating-point constant. Although most arithmetic constants can be signed, the sign is not part of the constant.

arithmetic conversion. The transformation of a value from one arithmetic representation to another.

arithmetic data. Data that has the characteristics of base, scale, mode, and precision. Coded arithmetic data and pictured numeric character data are included.

arithmetic operators. Either of the prefix operators + and −, or any of the following infix operators: + − * / **

array. A named, ordered collection of one or more data elements with identical attributes, grouped into one or more dimensions.

array expression. An expression whose evaluation yields an array of values.

array of structures. An ordered collection of identical structures specified by giving the dimension attribute to a structure name.

array variable. A variable that represents an aggregate of data items that must have identical attributes. Contrast with *structure variable*.

ASCII. American National Standard Code for Information Interchange.

assignment. The process of giving a value to a variable.

asynchronous operation. The overlap of an input/output operation with the execution of statements. The concurrent execution of procedures using multiple flows of control for different tasks.

attachment of a task. The invocation of a procedure and the establishment of a separate flow of control to execute the invoked procedure (and procedures it invokes) asynchronously, with execution of the invoking procedure.

attention. An occurrence, external to a task, that could cause a task to be interrupted.

attribute. A descriptive property associated with a name to describe a characteristic represented. A descriptive property used to describe a characteristic of the result of evaluation of an expression.

automatic storage allocation. The allocation of storage for automatic variables.

automatic variable. A variable whose storage is allocated automatically at the activation of a block and released automatically at the termination of that block.

B

base. The number system in which an arithmetic value is represented.

base element. A member of a structure or a union that is itself not another structure or union.

base item. The automatic, controlled, or static variable or the parameter upon which a defined variable is defined.

based reference. A reference that has the based storage class.

based storage allocation. The allocation of storage for based variables.

based variable. A variable whose storage address is provided by a locator. Multiple generations of the same variable are accessible. It does not identify a fixed location in storage.

begin-block. A collection of statements delimited by BEGIN and END statements, forming a name scope. A begin-block is activated either by the raising of a condition (if the begin-block is the action specification for an ON-unit) or through the normal flow of control, including any branch resulting from a GOTO statement.

binary. A number system whose only numerals are 0 and 1.

binary digit. See *bit*.

binary fixed-point value. An integer consisting of binary digits and having an optional binary point and optional sign. Contrast with *decimal fixed-point value*.

binary floating-point value. An approximation of a real number in the form of a significand, which can be considered as a binary fraction, and an exponent, which can be considered as an integer exponent to the base of 2. Contrast with *decimal floating-point value*.

bit. A 0 or a 1. The smallest amount of space of computer storage.

bit comparison. A left-to-right, bit-by-bit comparison of binary digits. See also *arithmetic comparison, character comparison*.

bit string constant. A series of binary digits enclosed in and followed immediately by the suffix B. Contrast with *character constant*. A series of hexadecimal digits enclosed in single quotes and followed by the suffix B4.

bit string. A string composed of zero or more bits.

bit string operators. The logical operators not and exclusive-or (¬), and (&), and or (|).

bit value. A value that represents a bit type.

block. A sequence of statements, processed as a unit, that specifies the scope of names and the allocation of storage for names declared within it. A block can be a package, procedure, or a begin-block.

bounds. The upper and lower limits of an array dimension.

break character. The underscore symbol (_). It can be used to improve the readability of identifiers. For instance, a variable could be called OLD_INVENTORY_TOTAL instead of OLDINVENTORYTOTAL.

built-in function. A predefined function supplied by the language, such as SQRT (square root).

built-in function reference. A built-in function name, which has an optional argument list.

built-in name. The entry name of a built-in subroutine.

built-in subroutine. Subroutine that has an entry name that is defined at compile-time and is invoked by a CALL statement.

buffer. Intermediate storage, used in input/output operations, into which a record is read during input and from which a record is written during output.

C

call. To invoke a subroutine by using the CALL statement or CALL option.

character comparison. A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison, bit comparison*.

character string constant. A sequence of characters enclosed in single quotes; for example, 'Shakespeare''s 'Hamlet:''.

character set. A defined collection of characters. See *language character set* and *data character set*. See also *ASCII* and *EBCDIC*.

character string picture data. Picture data that has only a character value. This type of picture data must have at least one A or X picture specification character. Contrast with *numeric picture data*.

closing (of a file). The dissociation of a file from a data set or device.

coded arithmetic data. Data items that represent numeric values and are characterized by their base (decimal or binary), scale (fixed-point or floating-point), and precision (the number of digits each can have). This data is stored in a form that is acceptable, without conversion, for arithmetic calculations.

combined nesting depth. The deepest level of nesting, determined by counting the levels of PROCEDURE/BEGIN/ON, DO, SELECT, and IF...THEN...ELSE nestings in the program.

comment. A string of zero or more characters used for documentation that are delimited by /* and */.

commercial character.
- CR (credit) picture specification character
- DB (debit) picture specification character

comparison operator. An operator that can be used in an arithmetic, string locator, or logical relation to indicate the comparison to be done between the terms in the relation. The comparison operators are:

    = (equal to)

    > (greater than)

    < (less than)

    >= (greater than or equal to)

    <= (less than or equal to)

    ¬= (not equal to)

    ¬> (not greater than)

    ¬< (not less than)

compile time. In general, the time during which a source program is translated into an object module. In PL/I, it is the time during which a source program can be altered, if desired, and then translated into an object program.

compiler options. Keywords that are specified to control certain aspects of a compilation, such as: the nature of the object module generated, the types of printed output produced, and so forth.

complex data. Arithmetic data, each item of which consists of a real part and an imaginary part.

composite operator. An operator that consists of more than one special character, such as <=, **, and /*.

compound statement. A statement that contains other statements. In PL/I, IF, ON, OTHERWISE, and WHEN are the only compound statements. See *statement body*.

concatenation. The operation that joins two strings in the order specified, forming one string whose length is equal to the sum of the lengths of the two original strings. It is specified by the operator ||.

condition. An exceptional situation, either an error (such as an overflow), or an expected situation (such as the end of an input file). When a condition is raised (detected), the action established for it is processed. See also *established action* and *implicit action*.

condition name. Name of a PL/I-defined or programmer-defined condition.

condition prefix. A parenthesized list of one or more condition names prefixed to a statement. It specifies whether the named conditions are to be enabled or disabled.

connected aggregate. An array or structure whose elements occupy contiguous storage without any intervening data items. Contrast with *nonconnected aggregate*.

connected reference. A reference to connected storage. It must be apparent, prior to execution of the program, that the storage is connected.

connected storage. Main storage of an uninterrupted linear sequence of items that can be referred to by a single name.

constant. An arithmetic or string data item that does not have a name and whose value cannot change. An identifier declared with the VALUE attribute. An identifier declared with the FILE or the ENTRY attribute but without the VARIABLE attribute.

constant reference. A value reference which has a constant as its object

contained block, declaration, or source text. All blocks, procedures, statements, declarations, or source text inside a begin, procedure, or a package block. The entire package, procedure, and the BEGIN statement and its corresponding END statements are not contained in the block.

containing block. The package, procedure, or begin-block that contains the declaration, statement, procedure, or other source text in question.

contextual declaration. The appearance of an identifier that has not been explicitly declared in a DECLARE statement, but whose context of use allows the association of specific attributes with the identifier.

control character. A character in a character set whose occurrence in a particular context specifies a control function. One example is the end-of-file (EOF) marker.

control format item. A specification used in edit-directed transmission to specify positioning of a data item within the stream or printed page.

control variable. A variable that is used to control the iterative execution of a DO statement.

controlled parameter. A parameter for which the CONTROLLED attribute is specified in a DECLARE statement. It can be associated only with arguments that have the CONTROLLED attribute.

controlled storage allocation. The allocation of storage for controlled variables.

controlled variable. A variable whose allocation and release are controlled by the ALLOCATE and FREE statements, with access to the current generation only.

control sections. Grouped machine instructions in an object module.

conversion. The transformation of a value from one representation to another to conform to a given set of attributes. For example, converting a character string to an arithmetic value such as FIXED BINARY (15,0).

cross section of an array. The elements represented by the extent of at least one dimension of an array. An asterisk in the place of a subscript in an array reference indicates the entire extent of that dimension.

current generation. The generation of an automatic or controlled variable that is currently available by referring to the name of the variable.

D

data. Representation of information or of value in a form suitable for processing.

data aggregate. A data item that is a collection of other data items.

data attribute. A keyword that specifies the type of data that the data item represents, such as FIXED BINARY.

data-directed transmission. The type of stream-oriented transmission in which data is transmitted. It resembles an assignment statement and is of the form `name = constant`.

data item. A single named unit of data.

data list. In stream-oriented transmission, a parenthesized list of the data items used in GET and PUT statements. Contrast with *format list*.

data set. A collection of data external to the program that can be accessed by reference to a single file name. A device that can be referenced.

data specification. The portion of a stream-oriented transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list(s) and, for edit-directed mode, the format list(s).

data stream. Data being transferred from or to a data set by stream-oriented transmission, as a continuous stream of data elements in character form.

data transmission. The transfer of data from a data set to the program or vice versa.

data type. A set of data attributes.

DBCS. In the character set, each character is represented by two consecutive bytes.

deactivated. The state in which an identifier is said to be when its value cannot replace a preprocessor identifier in source program text. Contrast with *active*.

debugging. Process of removing bugs from a program.

decimal. The number system whose numerals are 0 through 9.

decimal digit picture character. The picture specification character 9.

decimal fixed-point constant. A constant consisting of one or more decimal digits with an optional decimal point.

decimal fixed-point value. A rational number consisting of a sequence of decimal digits with an assumed position of the decimal point. Contrast with *binary fixed-point value*.

decimal floating-point constant. A value made up of a significand that consists of a decimal fixed-point constant, and an exponent that consists of the letter E followed by an optionally signed integer constant not exceeding three digits.

decimal floating-point value. An approximation of a real number, in the form of a significand, which can be considered as a decimal fraction, and an exponent, which can be considered as an integer exponent to the base 10. Contrast with *binary floating-point value*.

decimal picture data. See *numeric picture data.*

declaration. The establishment of an identifier as a name and the specification of a set of attributes (partial or complete) for it. A source of attributes of a particular name.

default. Describes a value, attribute, or option that is assumed when none has been specified.

defined variable. A variable that is associated with some or all of the storage of the designated base variable.

delimit. To enclose one or more items or statements with preceding and following characters or keywords.

delimiter. All comments and the following characters: percent, parentheses, comma, period, semicolon, colon, assignment symbol, blank, pointer, asterisk, and single quote. They define the limits of identifiers, constants, picture specifications, iSUBs, and keywords.

descriptor. A control block that holds information about a variable, such as area size, array bounds, or string length.

digit. One of the characters 0 through 9.

dimension attribute. An attribute that specifies the number of dimensions of an array and indicates the bounds of each dimension.

disabled. The state of a condition in which no interrupt occurs and no established action will take place.

do-group. A sequence of statements delimited by a DO statement and ended by its corresponding END statement, used for control purposes. Contrast with *block*.

do-loop. See *iterative do-group*.

dummy argument. Temporary storage that is created automatically to hold the value of an argument that cannot be passed by reference.

dump. Printout of all or part of the storage used by a program as well as other program information, such as a trace of an error's origin.

E

EBCDIC. (Extended Binary-Coded Decimal Interchange Code). A coded character set consisting of 8-bit coded characters.

edit-directed transmission. The type of stream-oriented transmission in which data appears as a continuous stream of characters and for which a format list is required to specify the editing desired for the associated data list.

element. A single item of data as opposed to a collection of data items such as an array; a scalar item.

element expression. An expression whose evaluation yields an element value.

element variable. A variable that represents an element; a scalar variable.

elementary name. See *base element*.

enabled. The state of a condition in which the condition can cause an interrupt and then invocation of the appropriate established ON-unit.

end-of-step message. message that follows the listng of the job control statements and job scheduler messages and contains return code indicating success or failure for each step.

entry constant. The label prefix of a PROCEDURE statement (an entry name). The declaration of a name with the ENTRY attribute but without the VARIABLE attribute.

entry data. A data item that represents an entry point to a procedure.

entry expression. An expression whose evaluation yields an entry name.

entry name. An identifier that is explicitly or contextually declared to have the ENTRY attribute

(unless the VARIABLE attribute is given) or An identifier that has the value of an entry variable with the ENTRY attribute implied.

entry point.   A point in a procedure at which it can be invoked. *primary entry point* and *secondary entry point*.

entry reference.   An entry constant, an entry variable reference, or a function reference that returns an entry value.

entry variable.   A variable to which an entry value can be assigned. It must have both the ENTRY and VARIABLE attributes.

entry value.   The entry point represented by an entry constant or variable; the value includes the environment of the activation that is associated with the entry constant.

environment (of an activation).   Information associated with and used in the invoked block regarding data declared in containing blocks.

environment (of a label constant).   Identity of the particular activation of a block to which a reference to a statement-label constant applies. This information is determined at the time a statement-label constant is passed as an argument or is assigned to a statement-label variable, and it is passed or assigned along with the constant.

established action.   The action taken when a condition is raised. See also *implicit action* and *ON-statement action*.

epilogue.   Those processes that occur automatically at the termination of a block or task.

evaluation.   The reduction of an expression to a single value, an array of values, or a structured set of values.

event.   An activity in a program whose status and completion can be determined from an associated event variable.

event variable.   A variable with the EVENT attribute that can be associated with an event. Its value indicates whether the action has been completed and the status of the completion.

explicit declaration.   The appearance of an identifier (a name) in a DECLARE statement, as a label prefix, or in a parameter list. Contrast with *implicit declaration*.

exponent characters.   The following picture specification characters:

1. K and E, which are used in floating-point picture specifications to indicate the beginning of the exponent field.

2. F, the scaling factor character, specified with an integer constant that indicates the number of decimal positions the decimal point is to be moved

from its assumed position to the right (if the constant is positive) or to the left (if the constant is negative).

expression.   A notation, within a program, that represents a value, an array of values, or a structured set of values. A constant or a reference appearing alone, or a combination of constants and/or references with operators.

extended alphabet.   The uppercase and lowercase alphabetic characters A through Z, $, @ and #, or those specified in the NAMES compiler option.

extent.   The range indicated by the bounds of an array dimension, by the length of a string, or by the size of an area. The size of the target area if this area were to be assigned to a target area.

external name.   A name (with the EXTERNAL attribute) whose scope is not necessarily confined only to one block and its contained blocks.

external procedure.   A procedure that is not contained in any other procedure. A level-2 procedure contained in a package that is also exported.

external symbol.   Name that can be referred to in a control section other than the one in which it is defined.

External Symbol Dictionary (ESD).   Table containing all the external symbols that appear in the object module.

extralingual character.   Characters (such as $, @, and #) that are not classified as alphanumeric or special. This group includes characters that are determined with the NAMES compiler option.

F

factoring.   The application of one or more attributes to a parenthesized list of names in a DECLARE statement, eliminating the repetition of identical attributes for multiple names.

field (in the data stream).   That portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

field (of a picture specification).   Any character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number.

file.   A named representation, within a program, of a data set or data sets. A file is associated with the data set(s) for each opening.

file constant.   A name declared with the FILE attribute but not the VARIABLE attribute.

file description attributes. Keywords that describe the individual characteristics of each file constant. See also *alternative attribute* and *additive attribute*.

file expression. An expression whose evaluation yields a value of the type file.

file name. A name declared for a file.

file variable. A variable to which file constants can be assigned. It has the attributes FILE and VARIABLE and cannot have any of the file description attributes.

fixed-point constant. See *arithmetic constant*.

fix-up. A solution, performed by the compiler after detecting an error during compilation, that allows the compiled program to run.

floating-point constant. See *arithmetic constant*.

flow of control. Sequence of execution.

format. A specification used in edit-directed data transmission to describe the representation of a data item in the stream (data format item) or the specific positioning of a data item within the stream (control format item).

format constant. The label prefix on a FORMAT statement.

format data. A variable with the FORMAT attribute.

format label. The label prefix on a FORMAT statement.

format list. In stream-oriented transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

fully qualified name. A name that includes all the names in the hierarchical sequence above the member to which the name refers, as well as the name of the member itself.

function (procedure). A procedure that has a RETURNS option in the PROCEDURE statement. A name declared with the RETURNS attribute. It is invoked by the appearance of one of its entry names in a function reference and it returns a scalar value to the point of reference. Contrast with *subroutine*.

function reference. An entry constant or an entry variable, either of which must represent a function, followed by a possibly empty argument list. Contrast with *subroutine call*.

G

generation (of a variable). The allocation of a static variable, a particular allocation of a controlled or automatic variable, or the storage indicated by a particular locator qualification of a based variable or by a defined variable or parameter.

generic descriptor. A descriptor used in a GENERIC attribute.

generic key. A character string that identifies a class of keys. All keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF', are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', 'ABDF', respectively.

generic name. The name of a family of entry names. A reference to the generic name is replaced by the entry name whose parameter descriptors match the attributes of the arguments in the argument list at the point of invocation.

group. A collection of statements contained within larger program units. A group is either a do-group or a select-group and it can be used wherever a single statement can appear, except as an on-unit.

H

hex. See *hexadecimal digit*.

hexadecimal. Pertaining to a numbering system with a base of sixteen; valid numbers use the digits 0 through 9 and the characters A through F, where A represents 10 and F represents 15.

hexadecimal digit. One of the digits 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively.

I

identifier. A string of characters, not contained in a comment or constant, and preceded and followed by a delimiter. The first character of the identifier must be one of the 26 alphabetic characters and extralingual characters, if any. The other characters, if any, can additionally include extended alphabetic, digit, or the break character.

IEEE. Institute of Electrical and Electronics Engineers.

implicit. The action taken in the absence of an explicit specification.

implicit action. The action taken when an enabled condition is raised and no ON-unit is currently established for the condition. Contrast with *ON-statement action*.

implicit declaration. A name not explicitly declared in a DECLARE statement or contextually declared.

implicit opening. The opening of a file as the result of an input or output statement other than the OPEN statement.

infix operator. An operator that appears between two operands.

inherited dimensions. For a structure, union, or element, those dimensions that are derived from the containing structures. If the name is an element that is not an array, the dimensions consist entirely of its inherited dimensions. If the name is an element that is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure with one or more inherited dimensions is called a nonconnected aggregate. Contrast with *connected aggregate*.

input/output. The transfer of data between auxiliary medium and main storage.

insertion point character. A picture specification character that is, on assignment of the associated data to a character string, inserted in the indicated position. When used in a P-format item for input, the insertion character is used for checking purposes.

integer. An optionally signed sequence of digits or a sequence of bits without a decimal or binary point. An optionally signed whole number, commonly described as FIXED BINARY (p,0) or FIXED DECIMAL (p,0).

integral boundary. A byte multiple address of any 8-bit unit on which data can be aligned. It usually is a halfword, fullword, or doubleword (2-, 4-, or 8-byte multiple respectively) boundary.

interleaved array. An array that refers to nonconnected storage.

interleaved subscripts. Subscripts that exist in levels other than the lowest level of a subscripted qualified reference.

internal block. A block that is contained in another block.

internal name. A name that is known only within the block in which it is declared, and possibly within any contained blocks.

internal procedure. A procedure that is contained in another block. Contrast with *external procedure.*

interrupt. The redirection of the program's flow of control as the result of raising a condition or attention.

invocation. The activation of a procedure.

invoke. To activate a procedure.

invoked procedure. A procedure that has been activated.

invoking block. A block that activates a procedure.

iteration factor. In an INITIAL attribute specification, an expression that specifies the number of consecutive elements of an array that are to be initialized with the given value. In a format list, an expression that specifies the number of times a given format item or list of format items is to be used in succession.

iterative do-group. A do-group whose DO statement specifies a control variable and/or a WHILE or UNTIL option.

K

key. Data that identifies a record within a direct-access data set. See *source key* and *recorded key*.

keyword. An identifier that has a specific meaning in PL/I when used in a defined context.

keyword statement. A simple statement that begins with a keyword, indicating the function of the statement.

known (applied to a name). Recognized with its declared meaning. A name is known throughout its scope.

L

label. A name prefixed to a statement. A name on a PROCEDURE statement is called an entry constant; a name on a FORMAT statement is called a format constant; a name on other kinds of statements is called a label constant. A data item that has the LABEL attribute.

label constant. A name written as the label prefix of a statement (other than PROCEDURE, ENTRY, FORMAT, or PACKAGE) so that, during execution, program control can be transferred to that statement through a reference to its label prefix.

label data. A label constant or the value of a label variable.

label prefix. A label prefixed to a statement.

label variable. A variable declared with the LABEL attribute. Its value is a label constant in the program.

leading zeroes. Zeros that have no significance in an arithmetic value. All zeros to the left of the first nonzero in a number.

level number. A number that precedes a name in a DECLARE statement and specifies its relative position in the hierarchy of structure names.

level-one variable. A major structure or union name. Any unsubscripted variable not contained within a structure or union.

lexically. Relating to the left-to-right order of units.

library. An MVS partitioned data set or a CMS MACLIB that can be used to store other data sets called members.

list-directed. The type of stream-oriented transmission in which data in the stream appears as constants separated by blanks or commas and for which formatting is provided automatically.

locator. A control block that holds the address of a variable or its descriptor.

locator/descriptor. A locator followed by a descriptor. The locator holds the address of the variable, not the address of the descriptor.

locator qualification. In a reference to a based variable, either a locator variable or function reference connected by an arrow to the left of a based variable to specify the generation of the based variable to which the reference refers. It might be an implicit reference.

locator value. A value that identifies or can be used to identify the storage address.

locator variable. A variable whose value identifies the location in main storage of a variable or a buffer. It has the POINTER or OFFSET attribute.

locked record. A record in an EXCLUSIVE DIRECT UPDATE file that has been made available to one task only and cannot be accessed by other tasks until the task using it relinquishes it.

logical level (of a structure or union member). The depth indicated by a level number when all level numbers are in direct sequence (when the increment between successive level numbers is one).

logical operators. The bit-string operators not and exclusive-or (¬), and (&), and or (|).

loop. A sequence of instructions that is executed iteratively.

lower bound. The lower limit of an array dimension.

M

main procedure. An external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. This procedure is invoked automatically as the first step in the execution of a program.

major structure. A structure whose name is declared with level number 1.

member. A structure, union, or element name in a structure or union. Data sets in a library.

minor structure. A structure that is contained within another structure or union. The name of a minor structure is declared with a level number greater than one and greater than its parent structure or union.

mode (of arithmetic data). An attribute of arithmetic data. It is either *real* or *complex*.

multiple declaration. Two or more declarations of the same identifier internal to the same block without different qualifications. Two or more external declarations of the same identifier.

multiprocessing. The use of a computing system with two or more processing units to execute two or more programs simultaneously.

multiprogramming. The use of a computing system to execute more than one program concurrently, using a single processing unit.

N

name. Any identifier that the user gives to a variable or to a constant. An identifier appearing in a context where it is not a keyword. Sometimes called a user-defined name.

nesting. The occurrence of:

- A block within another block
- A group within another group
- An IF statement in a THEN clause or in an ELSE clause
- A function reference as an argument of a function reference
- A remote format item in the format list of a FORMAT statement
- A parameter descriptor list in another parameter descriptor list
- An attribute specification within a parenthesized name list for which one or more attributes are being factored

nonconnected storage. Storage occupied by nonconnected data items. For example, interleaved arrays and structures with inherited dimensions are in nonconnected storage.

null locator value. A special locator value that cannot identify any location in internal storage. It gives a positive indication that a locator variable does not currently identify any generation of data.

null statement. A statement that contains only the semicolon symbol (;). It indicates that no action is to be taken.

null string. A character, graphic, or bit string with a length of zero.

numeric-character data. See *decimal picture data*.

numeric picture data. Picture data that has an arithmetic value as well as a character value. This type of picture data cannot contain the characters 'A' or 'X.'

O

object. A collection of data referred to by a single name.

offset variable. A locator variable with the OFFSET attribute, whose value identifies a location in storage relative to the beginning of an area.

ON-condition. An occurrence, within a PL/I program, that could cause a program interrupt. It can be the detection of an unexpected error or of an occurrence that is expected, but at an unpredictable time.

ON-statement action. The action explicitly established for a condition that is executed when the condition is raised. When the ON-statement is encountered in the flow of control for the program, it executes, establishing the action for the condition. The action executes when the condition is raised if the ON-unit is still established or a RESIGNAL statement reestablishes it. Contrast with *implicit action*.

ON-unit. The specified action to be executed when the appropriate condition is raised.

opening (of a file). The association of a file with a data set.

operand. The value of an identifier, constant, or an expression to which an operator is applied, possibly in conjunction with another operand.

operational expression. An expression that consists of one or more operators.

operator. A symbol specifying an operation to be performed.

option. A specification in a statement that can be used to influence the execution or interpretation of the statement.

P

package constant. The label prefix on a PACKAGE statement.

packed decimal. The internal representation of a fixed-point decimal data item.

padding. One or more characters, graphics, or bits concatenated to the right of a string to extend the string to a required length. One or more bytes or bits inserted in a structure or union so that the following element

within the structure or union is aligned on the appropriate integral boundary.

parameter. A name in the parameter list following the PROCEDURE statement, specifying an argument that will be passed when the procedure is invoked.

parameter descriptor. The set of attributes specified for a parameter in an ENTRY attribute specification.

parameter descriptor list. The list of all parameter descriptors in an ENTRY attribute specification.

parameter list. A parenthesized list of one or more parameters, separated by commas and following either the keyword PROCEDURE in a procedure statement or the keyword ENTRY in an ENTRY statement. The list corresponds to a list of arguments passed at invocation.

partially qualified name. A qualified name that is incomplete. It includes one or more, but not all, of the names in the hierarchical sequence above the structure or union member to which the name refers, as well as the name of the member itself.

picture data. Numeric data, character data, or a mix of both types, represented in character form.

picture specification. A data item that is described using the picture characters in a declaration with the PICTURE attribute or in a P-format item.

picture specification character. Any of the characters that can be used in a picture specification.

PL/I character set. A set of characters that has been defined to represent program elements in PL/I.

PL/I prompter. Command processor program for the PLI command that checks the operands and allocates the data sets required by the compiler.

point of invocation. The point in the invoking block at which the reference to the invoked procedure appears.

pointer. A type of variable that identifies a location in storage.

pointer value. A value that identifies the pointer type.

pointer variable. A locator variable with the POINTER attribute that contains a pointer value.

precision. The number of digits or bits contained in a fixed-point data item, or the minimum number of significant digits (excluding the exponent) maintained for a floating-point data item.

prefix. A label or a parenthesized list of one or more condition names included at the beginning of a statement.

prefix operator. An operator that precedes an operand and applies only to that operand. The prefix operators are plus (+), minus (−), and not (¬).

preprocessor. A program that examines the source program before the compilation takes place.

preprocessor statement. A special statement appearing in the source program that specifies the actions to be performed by the preprocessor. It is executed as it is encountered by the preprocessor.

primary entry point. The entry point identified by any of the names in the label list of the PROCEDURE statement.

priority. A value associated with a task, that specifies the precedence of the task relative to other tasks.

problem data. Coded arithmetic, bit, character, graphic, and picture data.

problem-state program. A program that operates in the problem state of the operating system. It does not contain input/output instructions or other privileged instructions.

procedure. A collection of statements, delimited by PROCEDURE and END statements. A procedure is a program or a part of a program, delimits the scope of names, and is activated by a reference to the procedure or one of its entry names. See also *external procedure* and *internal procedure*.

procedure reference. An entry constant or variable. It can be followed by an argument list. It can appear in a CALL statement or the CALL option, or as a function reference.

program. A set of one or more external procedures or packages. One of the external procedures must have the OPTIONS(MAIN) specification in its procedure statement.

program control data. Area, locator, label, format, entry, and file data that is used to control the processing of a PL/I program.

prologue. The processes that occur automatically on block activation.

pseudovariable. Any of the built-in function names that can be used to specify a target variable. It is usually on the left-hand side of an assignment statement.

Q

qualified name. A hierarchical sequence of names of structure or union members, connected by periods, used to identify a name within a structure. Any of the names can be subscripted.

R

range (of a default specification). A set of identifiers and/or parameter descriptors to which the attributes in a DEFAULT statement apply.

record. The logical unit of transmission in a record-oriented input or output operation. A collection of one or more related data items. The items usually have different data attributes and usually are described by a structure or union declaration.

recorded key. A character string identifying a record in a direct-access data set where the character string itself is also recorded as part of the data.

record-oriented data transmission. The transmission of data in the form of separate records. Contrast with *stream data transmission*.

recursive procedure. A procedure that can be called from within itself or from within another active procedure.

reentrant procedure. A procedure that can be activated by multiple tasks, threads, or processes simultaneously without causing any interference between these tasks, threads, and processes.

REFER expression. The expression preceding the keyword REFER, which is used as the bound, length, or size when the based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

REFER object. The variable in a REFER option that holds or will hold the current bound, length, or size for the member. The REFER object must be a member of the same structure or union. It must not be locator-qualified or subscripted, and it must precede the member with the REFER option.

reference. The appearance of a name, except in a context that causes explicit declaration.

relative virtual origin (RVO). The actual origin of an array minus the virtual origin of an array.

remote format item. The letter R followed by the label (enclosed in parentheses) of a FORMAT statement. The format statement is used by edit-directed data transmission statements to control the format of data being transmitted.

repetition factor. A parenthesized unsigned integer constant that specifies:

1. The number of times the string constant that follows is to be repeated.
2. The number of times the picture character that follows is to be repeated.

repetitive specification. An element of a data list that specifies controlled iteration to transmit one or more data items, generally used in conjunction with arrays.

restricted expression. An expression that can be evaluated by the compiler during compilation, resulting in a constant. Operands of such an expression are constants, named constants, and restricted expressions.

returned value. The value returned by a function procedure.

RETURNS descriptor. A descriptor used in a RETURNS attribute, and in the RETURNS option of the PROCEDURE and ENTRY statements.

S

scalar variable. A variable that is not a structure, union, or array.

scale. A system of mathematical notation whose representation of an arithmetic value is either fixed-point or floating-point.

scale factor. A specification of the number of fractional digits in a fixed-point number.

scaling factor. See *scale factor*.

scope (of a condition prefix). The portion of a program throughout which a particular condition prefix applies.

scope (of a declaration or name). The portion of a program throughout which a particular name is known.

secondary entry point. An entry point identified by any of the names in the label list of an entry statement.

select-group. A sequence of statements delimited by SELECT and END statements.

selection clause. A WHEN or OTHERWISE clause of a select-group.

self-defining data. An aggregate that contains data items whose bounds, lengths, and sizes are determined at program execution time and are stored in a member of the aggregate.

separator. See *delimiter*.

shift. Change of data in storage to the left or to the right of original position.

shift-in. Symbol used to signal the compiler at the end of a double-byte string.

shift-out. Symbol used to signal the compiler at the beginning of a double-byte string.

sign and currency symbol characters. The picture specification characters. S, +, –, and $ (or other national currency symbols enclosed in < and >).

simple parameter. A parameter for which no storage class attribute is specified. It can represent an argument of any storage class, but only the current generation of a controlled argument.

simple statement. A statement other than IF, ON, WHEN, and OTHERWISE.

source. Data item to be converted for problem data.

source key. A key referred to in a record-oriented transmission statement that identifies a particular record within a direct-access data set.

source program. A program that serves as input to the source program processors and the compiler.

source variable. A variable whose value participates in some other operation, but is not modified by the operation. Contrast with *target variable*.

spill file. Data set named SYSUT1 that is used as a temporary workfile.

standard default. The alternative attribute or option assumed when none has been specified and there is no applicable DEFAULT statement.

standard file. A file assumed by PL/I in the absence of a FILE or STRING option in a GET or PUT statement. SYSIN is the standard input file and SYSPRINT is the standard output file.

standard system action. Action specified by the language to be taken for an enabled condition in the absence of an ON-unit for that condition.

statement. A PL/I statement, composed of keywords, delimiters, identifiers, operators, and constants, and terminated by a semicolon (;). Optionally, it can have a condition prefix list and a list of labels. See also *keyword statement, assignment statement*, and *null statement*.

statement body. A statement body can be either a simple or a compound statement.

statement label. See *label constant*.

static storage allocation. The allocation of storage for static variables.

static variable. A variable that is allocated before execution of the program begins and that remains allocated for the duration of execution.

stream-oriented data transmission. The transmission of data in which the data is treated as though it were a continuous stream of individual data values in character form. Contrast with *record-oriented data transmission*.

string. A contiguous sequence of characters, graphics, or bits that is treated as a single data item.

string variable. A variable declared with the BIT, CHARACTER, or GRAPHIC attribute, whose values can be either bit, character, or graphic strings.

structure. A collection of data items that need not have identical attributes. Contrast with *array*.

structure expression. An expression whose evaluation yields a structure set of values.

structure of arrays. A structure that has the dimension attribute.

structure member. See *member*.

structuring. The hierarchy of a structure, in terms of the number of members, the order in which they appear, their attributes, and their logical level.

subroutine. A procedure that has no RETURNS option in the PROCEDURE statement. Contrast with *function*.

subroutine call. An entry reference that must represent a subroutine, followed by an optional argument list that appears in a CALL statement. Contrast with *function reference*.

subscript. An element expression that specifies a position within a dimension of an array. If the subscript is an asterisk, it specifies all of the elements of the dimension.

subscript list. A parenthesized list of one or more subscripts, one for each dimension of the array, which together uniquely identify either a single element or cross section of the array.

subtask. A task that is attached by the given task or any of the tasks in a direct line from the given task to the last attached task.

synchronous. A single flow of control for serial execution of a program.

T

target. Attributes to which a data item (source) is converted.

target reference. A reference that designates a receiving variable (or a portion of a receiving variable).

target variable. A variable to which a value is assigned.

task. The execution of one or more procedures by a single flow of control.

task name. An identifier used to refer to a task variable.

task variable. A variable with the TASK attribute whose value gives the relative priority of a task.

termination (of a block). Cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or to some other active block by means of a GO TO statement.

termination (of a task). Cessation of the flow of control for a task.

truncation. The removal of one or more digits, characters, graphics, or bits from one end of an item of data when a string length or precision of a target variable has been exceeded.

type. The set of data attributes and storage attributes that apply to a generation, a value, or an item of data.

U

undefined. Indicates something that a user must not do. Use of a undefined feature is likely to produce different results on different implementations of a PL/I product. In that case, the application program is in error.

union. A collection of data elements that overlay each other, occupying the same storage. The members can be structures, unions, elementary variables, or arrays. They need not have identical attributes.

union of arrays. A union that has the DIMENSION attribute.

upper bound. The upper limit of an array dimension.

V

value reference. A reference used to obtain the value of an item of data.

variable. A named entity used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times.

variable reference. A reference that designates all or part of a variable.

virtual origin (VO). The location where the element of the array whose subscripts are all zero are held. If such an element does not appear in the array, the virtual origin is where it would be held.

Z

zero-suppression characters. The picture
specification characters Z and *, which are used to
suppress zeros in the corresponding digit positions and
replace them with blanks or asterisks respectively.

Index

Special characters

B

B (insertion character)   321
B-format item   304
B3 (bit hex) bit string constant   39
B4 (bit hex) bit string constant   39
BASED attribute   229
based storage
    built-in functions   229
    description   222
    syntax for BASED attribute   229
based variables
    ALLOCATE statement   234
    built-in functions   229
    description   229, 233
    FREE statement   235
    input/output of lists   241
BEGIN statement
    description   111
    valid OPTIONS options for   126
begin-blocks
    activation   111
    description   110
    example   110
    termination   111
BIGENDIAN attribute   243
BINARY (BIN) attribute   29
BINARY (BIN) built-in function   392
binary digit   11
binary fixed-point constant   31, 32
binary fixed-point data
    conversion   79
    description   31
binary floating-point constant   33
binary floating-point data
    conversion   80
    description   33
BINARYVALUE (BINVALUE) built-in function   393
BINARYVALUE built-in function
    for ordinals   143
    using with pointer expressions   56
BIND type function   496
bit
    constant   38
    conversion
        description   76
        rules   84
    data   38
    operators
        description   14
        using in bit operations   63
BIT attribute   34
BIT built-in function   393
bit format item   304
bit operations
    examples   64
    using   63
bit strings, transmission of unaligned   272
BITLOCATION (BITLOC) built-in function   393
BKWD environment characteristic   264
blanks
    description   15

blanks *(continued)*
    using as a delimiter   14
blocks
    activation   89
    begin   110
    description   89
    packages   90
    procedures   92
    termination   90
    types   89
BOOL built-in function   394
Boolean operators   63
bounds   167
    controlled parameter   96
    simple parameter   96
BUF (BUFFERED) attribute   264
buffer-management built-in functions
    COMPARE   399
    HEXIMAGE   421
    MEMINDEX   434
    MEMSEARCH   435
    MEMSEARCHR   435
    MEMVERIFY   436
    MEMVERIFYR   437
    XMLCHAR   491
BUFFERED (BUF) attribute   264
built-in functions
    ABS   386
    accuracy of mathematical functions in   373
    ACOS   386
    ACOSF   386
    ADD   387
    ADDR   387
    ADDRDATA   388
    aggregate arguments   373
    ALL   388
    ALLOCATE (ALLOC)   388
    ALLOCATION (ALLOCN)   389
    ALLOCSIZE   389
    ANY   389
    area variables   240
    arithmetic, summary   374
    array-handling, summary   375
    ASIN   390
    ASINF   390
    ATAN   390
    ATAND   391
    ATANF   391
    ATANH   391
    AUTOMATIC (AUTO)   392
    AVAILABLEAREA   392
    based variables   233
    BINARY
        converting data   74
    BINARY (BIN)   392
    BINARYVALUE
        using with ordinals   143
        using with pointer expressions   56
    BINARYVALUE (BINVALUE)   393
    BIT   393
        converting data   74

I

I (overpunch) picture character   326
IAND built-in function   422
identifier
   asterisk   13
   DBCS   19
   DBCS with double-byte characters   19
   definition   13
   programmer-defined names   13
   SBCS in DBCS form   19
   scalar   46
   using keywords   13
IEEE attribute   244
IEOR built-in function   422
IF statement   524
   description   208
   syntax   208
IF statez   524
IGNORE option of data transmission statements   275
IMAG built-in function   423
IMAG pseudovariable   423
imaginary constants   30
implementation limits   533
implicit
   declaration   150
   freeing
      of based variable   235
      of controlled variable   228
   opening of files   267
implicit action   331
Implicit date
   assignments   44
   comparisons   44
implicitly locator-qualified reference   232
IN option
   ALLOCATE statement   234
   FREE statement   235
IN option with FREE statement, for based
 variables   235
INCLUDE directive   210
INCLUDE statez   524
INDEX built-in function   423
INDEX macro facility built-in function   513
indexed data sets   259
industry standards   4
infix operation   55
infix operators and arrays   70
INITIAL (INIT) attribute   251
INITIAL CALL   253
INITIAL TO   253
initial values
   for unions   251
   on STATIC variables   255
initializing
   array variables   254
   automatic variables   255
   based and controlled variables   256
   static variables   255
   unions   255
INLINE option   131
INOT built-in function   424

input
   conditions
      ENDFILE   344
      ENDPAGE   345
      KEY   349
      NAME   349
      RECORD   351
      TRANSMIT   355
      UNDEFINEDFILE   356
   definition   257
   discussion   257
   of area   241
INPUT attribute   263
input/output built-in functions
   COUNT   402
   ENDFILE   411
   FILEDDINT   413
   FILEDDTEST   414
   FILEDDWORD   414
   FILEID   415
   FILEOPEN   415
   FILEREAD   415
   FILESEEK   415
   FILETELL   416
   FILEWRITE   416
   LINENO   428
   ONSUBCODE   448
   PAGENO   450
   SAMEKEY   467
   summary   379
insertion characters   321
INT (INTERNAL) attribute   153
integer
   value   29
integer manipulation built-in functions
   IAND   422
   IEOR   422
   INOT   424
   IOR   424
   ISIGNED   425
   ISLL   425
   ISRL   426
   IUNSIGNED   426
   LOWER2   431
   RAISE2   462
   summary   379
integral boundary   159
interlanguage communication
   LINKAGE option   132
   linkages
      OPTLINK   132
      SYSTEM   132
interleaved subscripts   177
intermediate results of expressions
   discussion   54
   example   61
INTERNAL (INT) attribute   153
internal procedure   93
internal to, definition   152
INTO option of data transmission statements   275
INVALIDOP condition   348

recursion
   attribute   94
   definition   100
   effect on automatic variables   101
RECURSIVE attribute   100
RECURSIVE option   100
recursive procedures
   description   100
   effect on automatic variables   101
   example   101
   specifying attributes   100
REDUCIBLE (RED) option   133
REENTRANT option   133
REFER option
   description   236
   on AREA attribute   238
reference
   locator   231
references
   description   51
   preprocessor   503
   syntax   51
regional data set   259
REGIONAL(1) environment characteristic   264
relative data sets   259
relative line   313
RELEASE statement
   description   103
   dynamically loading external procedures   101
   example   104
   restrictions   102
REM built-in function   464
remote format item   312
REORDER option   132
REPATTERN built-in function   464
REPEAT built-in function   465
REPEAT macro facility built-in function   516
REPEAT option   198
repetition factor
   for picture characters   316
   for strings   38
repetitive execution (DO statement)   196, 202
REPLACE statez   527
REPLY option   195
RESCAN keyword on ANSWER statement   508
RESCAN option   519
RESERVED attribute   158
RESERVES option   91
RESIGNAL statement   337
RESPEC type function   498
restoring language-specified defaults   167
restricted expressions
   applying built-in functions   71
   description   71
   example   71
restrictions on FETCH and RELEASE
   description   102
   errors during data conversion   86
results of arithmetic operations
   discussion   57
   FLOAT operands   58

results of arithmetic operations *(continued)*
   special cases   63
results of arithmetic operations, under
   RULES(ANS)   59
RETCODE option   133
RETURN statement
   description   125
   returning from a function   125
   using   99
   using in a preprocessor procedure   507
   using with subroutines   125
RETURNS attribute   134
RETURNS option
   description   134
REVERSE built-in function   465
REVERT statement   337
REWRITE statement
   description   273
RIGHT built-in function   465
ROUND built-in function   466

S

S picture character   324
SAMEKEY built-in function   467
scalar identifiers   46
SCALARVARYING environment characteristic   264
SCALARVARYING option   271
SCALE built-in function   467
scale in arithmetic operations   57
scaling factor
   character   328
   description   29
SCAN keyword on ANSWER statement   508
SCAN option   519
scan, preprocessor   501
scope
   of condition prefix   333
   of established action   335
   of label declarations   151
scope of
   preprocessor names   504
SEARCH built-in function   468
SEARCHR built-in function   469
SECS built-in function   469
SECSTODATE built-in function   470
SECSTODAYS built-in function   471
SELECT statement
   description   217
   example of   218
SELECT statez   528
select-groups   217
self-defining data (REFER option)   236
semicolon   14
SEQL (SEQUENTIAL) attribute   263
SEQUENTIAL (SEQL) attribute   263
SET option
   description   234
   specifying a pointer reference   103
   using the ALLOCATE statement   234
   using the LOCATE statement   274

Y4YEAR built-in function 494

Z

Readers' Comments — We'd Like to Hear from You

Enterprise PL/I for z/OS
Language Reference
Version 3 Release 3

Publication No. SC27-1460-03

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?   ☐ Yes   ☐ No

Name

Address

Company or Organization

Phone No.

IBM®

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL   PERMIT NO. 40   ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department HHX/H3
555 Bailey Ave.
San Jose, CA
 95141-1099

Fold and Tape **Please do not staple** Fold and Tape