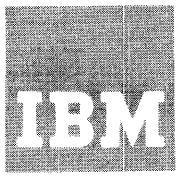
Form No. C28-8202-0
File No. S360-29



Systems Reference Library

IBM System/360

PL/I Subset Reference Manual

This publication provides the rules for writing PL/I Subset programs that are to be compiled using the PL/I D-level compiler under the IBM System/360 Disk and Tape Operating Systems. It is not a reference to the entire PL/I Subset language, but only to those features implemented by the Second Version of the D-level compiler.

PREFACE

This publication is planned for use as a reference book by the PL/I Subset programmer. It is not intended to be a tutorial publication, but is designed for the reader who already has a knowledge of the language and requires a source of reference material.

It is divided into two parts. Part I contains discussions of the concepts of the language. Part II contains detailed rules and syntactic descriptions.

Although implementation information is included, the book is not a complete description of any implementation environment. In general, it contains information needed in writing a program; it does not contain all of the information required to execute a program.

The following features are described as they are implemented in the Second Version of the D-Compiler; they are implemented differently in the First Version:

1. Arithmetic-to-Bit-String Conversion: The First Version uses the internal representation of the arithmetic value; the Second Version takes the absolute value. This will have a different effect only for negative values.

2. FIXEDOVERFLOW Condition: For the First Version, the result of this condition is truncation on the left and the standard system action is to comment and continue. For the Second Version, the result is undefined and the standard system action is to comment and raise the ERROR condition.

3. SKIP Option of PUT: A specification of SKIP(0) under the First Version causes the previously-transmitted line of characters to be replaced by the new line; the new line being the one actually printed. Under the Second Version, such a specification causes overprinting of the previously-transmitted line by the new line; thus, for example, underscoring is possible.

REQUISITE PUBLICATION

For information necessary to compile, linkage edit, and execute a program, the reader should be familiar with the following publication:

IBM System/360 Disk and Tape Operating Systems: PL/I Programmer's Guide, Form C24-9005.

RECOMMENDED PUBLICATIONS

The following publications contain other information that might be valuable to the PL/I programmer or to a programmer who is learning PL/I:

A PL/I Primer, Form C28-6808

A Guide to PL/I for Commercial Programmers, Form C20-1651

A Guide to PL/I for FORTRAN Users, Form C20-1637

The following publication contains a complete description of the PL/I Subset language:

PL/I Subset Language Specifications, Form C28-6809

First Edition

Specifications contained herein are subject to change from time to time. Any such change will be reported in subsequent revisions or Technical Newsletters.

Address comments concerning the contents of the publication to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020.

CONTENTS

ILLUSTRATIONS

FIGURES

TABLES

The PL/I Subset Language was designed for use in a data processing system of limited capacity. The subset is self-contained; i.e., the programmer can learn and use it without referring to the parent PL/I language. While many of the more sophisticated features of PL/I, such as asynchronous operations and compile-time facilities are not included in the PL/I subset, much of the programming power of PL/I has been retained.

Two of the basic characteristics of PL/I that have been carried over into the PL/I subset (hereinafter simply called PL/I) are its block structure and its machine independence. They reduce the need to rewrite complete programs if either the machine environment or the application environment changes.

A PL/I program is composed of blocks of statements called procedure blocks (or procedures) and begin blocks, each of which defines a region of the program. A single program may consist of one procedure or of several procedures and begin blocks. Either a procedure block or a begin block can contain other blocks; a begin block must be contained in a procedure block. Each external procedure, that is, a procedure that is not contained in another procedure, is compiled separately. The same external procedure might be used in a number of different programs. Consequently, a necessary change made in that one block effectively makes the change in all programs that use it.

PL/I is much less machine dependent than most commonly used programming languages. In the interest of efficiency, however, certain features are provided that allow machine dependence for those cases in which complete independence would be too costly.

USE OF THIS PUBLICATION

This publication is designed as a reference book for the PL/I programmer. Its two-part format allows a presentation of the material in such a way that references can be found quickly, in as much or as little detail as the user needs.

Part I, "Concepts of PL/I," is composed of discussions and examples that explain the different features of the language and their interrelationships. To reduce the need for cross references and to allow each chapter to stand alone as a complete reference to its subject, some information is repeated from one chapter to another. Part I can, nevertheless, be read sequentially in its entirety.

Part II, "Rules and Syntactic Descriptions," provides a quick reference to specific information. It includes less information about interrelationships, but it is organized so that a particular question can be answered quickly. Part II is organized purely from a reference point of view; it is not intended for sequential reading.

For example, a programmer would read Chapter 5 in Part I, "Statement Classification," for information about the interactions of different statements in a program; but he would look in Section J of Part II, "Statements," to find all the rules for the use of a specific statement, its effect, options allowed, and the format in which it is written.

In the same manner, he would read Chapter 4 in Part I, "Expressions," for a discussion of the concepts of data conversion, but he would use Section F of Part II, "Data Conversion," to determine the exact results of a particular type of conversion.

An explanation of the syntax language used in this publication to describe elements of PL/I is contained in Part II, Section A, "Syntax Notation."

IMPLEMENTATION CONSIDERATIONS

This publication reflects current features of the D-Compiler. Consequently, some features that are in the PL/I subset language are not described in this publication. One example is the list-directed input/output facility of the language; another is the INITIAL attribute.

Some language features that have been implemented with limitations are described in this book in the light of the limitations. Wherever a description here differs from the description of the same feature in PL/I Subset Language Specifications, Form C28-6809, it is not to be construed as a respecification of the language, but merely a description of the implementation.

Note, however, that this book does reflect current language specifications. For example, the keyword BASED has been added to the language as the attribute specification for based variables, replacing the attribute specification CONTROLLED (pointer-variable).

No attempt is made, however, to provide complete implementation information; this publication is designed for use in conjunction with IBM System/360 Disk and Tape Operating Systems PL/I Programmer's Guide, Form C24-9005. Discussion of implementation is limited to those features that are required for a full explanation of the language. For example, a complete discussion of the ENVIRONMENT attribute is essential to an explanation of record-oriented input and output file organization.

Implementation features identified by the phrase "for System/360 implementations..." apply to all implementations of PL/I (subset or full set) for IBM System/360 computers. Features identified by the phrase "for the D-Compiler..." apply specifically to the IBM D-level compiler (for PL/I subset) under the IBM System/360 Disk and Tape Operating Systems.

A separate publication, IBM System/360 PL/I Reference Manual, Form C28-8201, provides the same type of implementation information as it applies to the F-level compiler (for the PL/I full set) used under the IBM System/360 Operating System.

PART I: CONCEPTS OF PL/I

The modularity of PL/I, the ease with which different combinations of language features can be used to meet different needs, is one of the most important characteristics of PL/I.

This chapter contains brief discussions of most of the basic features to provide an overall description of the language. Each is treated in more detail in subsequent chapters.

MACHINE INDEPENDENCE

No language can be completely machine independent, but PL/I is much less machine dependent than most commonly used programming languages. The methods used to achieve this show in the form of restrictions in the language. The most obvious example is that data with different characteristics cannot in general share the same storage; to equate a floating-point number with a certain number of alphabetic characters would involve assumptions about the representation of these data items which would not be true for all machines.

It is recognized that the price entailed by machine independence may sometimes be too high. In the interest of efficiency, certain features such as UNSPEC, RECORD input/output, and the use of pointers do permit a degree of machine dependence.

PROGRAM STRUCTURE

A PL/I program consists of one or more blocks of statements called procedures. A procedure may be thought of as the main program or as a subroutine. Procedures may use other procedures, and these procedures or subroutines may either be compiled separately or may be nested within the calling procedure and compiled with it. Each procedure may contain declarations that define names and control allocation of storage.

The rules defining the use of procedures, communication between procedures, the meaning of names, and allocation of storage are fundamental to the proper understanding of PL/I at any level but the most elementary. These rules give the programmer considerable control over the degree of interaction between subroutines. They permit flexible communication and storage allocation, at the same time allowing the definition of names and allocation of storage for private use within a procedure.

By giving the programmer freedom to determine the degree to which a subroutine can be generalized, PL/I makes it possible to write procedures which can freely be used in other environments, while still allowing interaction in procedures where interaction is desirable.

DATA TYPES AND DATA DESCRIPTION

The characteristic of PL/I that most contributes to the range of applications for which it can be used is the variety of data types that can be represented and manipulated. PL/I deals with arithmetic data, string data (bit and character), and program control data, such as labels and pointers (or addresses). Arithmetic data may be represented in a variety of ways; it can be binary or decimal, fixed-point or floating-point, and its precision may be specified.

PL/I provides features to perform arithmetic operations, operations for comparisons, logical manipulation of bit strings, and operations and functions for assembling, scanning, and subdividing character strings.

The compiler must be able to determine, for every name used in a program, the complete set of attributes associated with that name. The programmer may specify these attributes explicitly by means of a DECLARE statement, the compiler may determine all or some of the attributes by context, or the attributes may be assumed by default.

DEFAULT ASSUMPTIONS

An important feature of PL/I is its default philosophy. If all the attributes associated with a name, or all the options permitted in a statement, are not specified

by the programmer, attributes or options may be assigned by the compiler. This default action has two main consequences. First, it reduces the amount of declaration and other program writing required; second, it makes it possible to teach and use levels of the language for which the programmer need not know all possible alternatives, or even that alternatives exist.

Since defaults are based on assumptions about the intent of the programmer, errors or omissions may be overlooked, and incorrect attributes may be assigned by default. To reduce the chance of this, the D-Compiler optionally provides an attribute listing, which can be used to check the names in the program and the attributes asscciated with them.

STORAGE ALLOCATION

PL/I goes beyond most other languages in the flexibility of storage allocation that it provides. Dynamic storage allocation is comparatively difficult for an assembly language programmer to handle for himself; yet it is automatically provided in PL/I. There are three different storage classes: AUTOMATIC, STATIC, and BASED. In general, the default storage class in PL/I is AUTOMATIC. This class of storage is allocated whenever the block in which the variables are declared is activated. AUTOMATIC storage is freed and is available for re-use whenever control leaves the block in which the storage is allocated.

Storage may also be STATIC, in which case, it is allocated when the program is loaded, or it may be BASED, in which case, the address associated with a variable can be controlled by the programmer.

The existence of several storage classes enables the programmer to determine for himself the speed, storage space, or programming economy that he needs for each application. The cost of a particular facility will depend upon the implementation, but it will usually be true that the more dynamic the storage allocation, the greater the overhead in execution time.

EXPRESSIONS

Calculations in PL/I are specified by expressions. An expression has a meaning in PL/I that is similar to that of elementary algebra. For example:

```
A + B * C
```

This specifies multiplication of the value of B by the value of C and adding the value of A to the result. PL/I places some restrictions on the kinds of data that can be used in an expression. For example, A could be a binary floating-point number, B a decimal fixed-point number, and C a bit string, but none could be a character string.

When permissable mixed expressions are specified, the operands will be converted so that the operation can be evaluated meaningfully. Note, however, that the rules for conversion must be considered carefully; converted data may not have the same value as the original. And, of course, any conversion requires additional compiler-generated coding, which increases execution time.

The results of the evaluation of expressions are assigned to variables by means of the assignment statement. An example of an assignment statement is:

```
X = A + B * C;
```

This means: evaluate the expression on the right and store the result in X. If the attributes of X differ from the attributes of the result of the expression, conversion will again be performed.

DATA COLLECTIONS

PL/I permits the programmer many ways of describing and operating on collections of data, or data aggregates. Arrays are collections of data elements, all of the same type, collected into lists or tables of one or more dimensions. Structures are hierarchical collections of data, not necessarily all of the same type. Each level of the hierarchy may contain other structures of deeper levels. The deepest levels of the hierarchy represent elementary data items or arrays.

Arrays cannot contain structures, but structures can contain arrays. Operations can be specified for arrays, structures, or parts of arrays or structures. For example:

```
A = B + C;
```

In this assignment statement, A, B, and C could be arrays or structures.

INPUT AND OUTPUT

Facilities for input and output allow the user to choose between factors such as simplicity, machine independence, and efficiency. There are two broad classes of input/output in PL/I: stream-oriented and record-oriented.

Stream-oriented input/output is almost completely machine independent. On input, data items are selected one by one from what is assumed to be a continuous stream of characters that are converted to internal form and assigned to variables specified in a list. Similarly, on output, data items are converted one by one to external character form and are added to a conceptually continuous stream of characters.

For printing, the output stream may be considered to be divided into lines and pages. An output stream file may be declared to be a print file with a certain line size and page size. The programmer has facilities to detect the end of a page and to specify the beginning of a line or a page. These facilities may be used in subroutines that can be developed into a report generating system suitable for a particular installation or application.

Record input/output is machine dependent. It deals with collections of data, called records, and transmits these a record at a time without any data conversion; the external representation is an exact copy of the internal representation. Because the aggregate is treated as a whole, and because no conversion is performed, this form of input/output is potentially more efficient than stream-oriented input/output, although the actual efficiency of each class will, of course, depend on the implementation.

Stream-oriented input and output usually sacrifices efficiency for ease of handling. Each data item is transmitted separately and is examined to determine if data conversion is required. Record-oriented input and output, on the other hand, provides faster transmission by transmitting data as entire records, without conversion.

INTERRUPT ACTIVITIES

Modern computing systems provide facilities for interrupting the execution of a program whenever an exceptional condition arises. Further, they allow the program to deal with the exceptional condition and to return to the point at which the interrupt occurred.

PL/I provides facilities for detecting a variety of exceptional conditions. It allows the programmer to specify, by means of a condition prefix, whether certain interrupts will or will not occur if the condition should arise. And, by use of an ON statement, he can specify the action to be taken when an interrupt does occur.

There are few restrictions in the format of PL/I statements. Consequently, programs can be written without consideration of special coding forms or checking to see that each statement begins in a specific column. As long as each statement is terminated by a semicolon, the format is completely free. Each statement may begin in the next column or position after the previous statement, or any number of blanks may intervene. The D-Compiler requires that the first column of every card in the source program be blank; columns 73 through 80 of these cards are ignored and can contain any information.

CHARACTER SETS

One of two character sets may be used to write a source program; either a 60-character set or a 48-character set. For a given external procedure, the choice between the two sets is optional. In practice, this choice will depend upon the available equipment.

60-CHARACTER SET

The 60-character set is composed of digits, special characters, and alphabetic characters.

There are 29 alphabetic characters beginning with the currency symbol ($), the number sign (#), and the commercial "at" sign (@), which precede the 26 letters of the English alphabet in the IBM System/360 collating sequence in Extended Binary-Coded-Decimal Interchange Code (EBCDIC). For use with languages other than English, the three alphabetic characters can be used to cause printing of letters that are not included in the standard English alphabet.

There are ten digits. The decimal digits are the digits 0 through 9. A binary digit is either a 0 or a 1.

There are 21 special characters. They are as follows:

Name	Character
Blank	
Equal or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Point or period	.
Single quotation mark or apostrophe	'
Percent symbol	%
Semicolon	;
Colon	:
"Not" symbol	¬
"And" symbol	&
"Or" symbol	|
"Greater than" symbol	>
"Less than" symbol	<
Break character[1]	_
Question mark	?

Special characters are combined to create other symbols. For example, <= means "less than or equal to," ¬= means "not equal to." The combination ** denotes exponentiation (X**2 means X^2). Blanks are not permitted in such composite symbols.

An alphameric character is either an alphabetic character or a digit, but not a special character.

Note: The question mark, at present, has no specific use in the language, even though it is included in the 60-character set. The percent symbol has no meaning in the PL/I subset, although it does have meaning in the fullset.

48-CHARACTER SET

The 48-character set is composed of 48 characters of the 60-character set. In all but five cases, the characters of the reduced set can be combined to represent the missing characters from the larger set. For example, the semicolon (;) is not

---

[1]The break character is the same as the typewriter underline character. It can be used with a name, such as GROSS_PAY, to improve readability.

included in the 48-character set, but a comma followed by a point (,.), with no blanks intervening, can be used to represent it. The five characters that are not duplicated are the commercial "at" sign, the number sign, the break character, the question mark, and the percent symbol.

The restrictions and changes for this character set are described in Part II, Section B, "Character Sets with EBCDIC and Card-Punch Codes."

USING THE CHARACTER SET

All the elements that make up a PL/I program are constructed from the PL/I character sets. There are two exceptions: character-string constants and comments may contain any character permitted by a particular machine configuration.

Certain characters perform specific functions in a PL/I program. For example, many characters perform as operators.

There are four types of operators: arithmetic, comparison, bit-string, and string.

The arithmetic operators are:

+	denoting addition or prefix plus
-	denoting subtraction or prefix minus
*	denoting multiplication
/	denoting division
**	denoting exponentiation

The comparison operators are:

>	denoting "greater than"
¬>	denoting "not greater than"
>=	denoting "greater than or equal to"
=	denoting "equal to"
¬=	denoting "not equal to"
<=	denoting "less than or equal to"
<	denoting "less than"
¬<	denoting "not less than"

The bit-string operators are:

¬	denoting "not"
&	denoting "and"
|	denoting "or"

The string operator is:

||	denoting concatenation

Table 2-1 shows some of the functions of other special characters:

Identifiers

In a PL/I program, names or labels are given to data, files, statements, and entry points of different program areas. In creating a name or label, a programmer must observe the syntactic rules for creating an identifier.

An identifier is a single alphabetic character or a string of up to 31 alphameric and break characters, not contained in a comment or constant, and preceded and followed by a blank or some other delimiter; the initial character of the string must be alphabetic.

Language keywords also are identifiers. A keyword is an identifier that, when used in proper context, has a specific meaning to the compiler. A keyword can specify such things as the action to be taken, the nature of data, the purpose of a name. For example, READ, DECIMAL, and ENDFILE are keywords. A complete list of keywords and their use is contained in Part II, Section C, "Keywords."

Note: Most PL/I keywords are not reserved words. They are recognized as keywords by the compiler only when they appear in their proper context. In other contexts they may be used as programmer-defined identifiers. (Those keywords that are reserved are given in Chapter 7,"Recognition of Names.")

Table 2-1. Some Functions of Special Characters

Name	Character	Use
comma	,	Separates elements of a list
period	.	Indicates decimal point or binary point; connects elements of a qualified name
semicolon	;	Terminates statements
assignment symbol	=	Indicates assignment of values[1]
colon	:	Connects prefixes to statements
blank		Separates elements of a statement
single quotation mark	'	Encloses string constants and picture specifications
parentheses	( )	Enclose lists; specify information associated with various keywords; in conjunction with operators and operands, delimit portions of a computational expression

[1]The character = can be used as an equal sign and as an assignment symbol.

No identifier can exceed 31 characters in length. For the D-Compiler, some identifiers, as discussed in later chapters, cannot exceed six characters in length; this limitation is placed upon certain names, called external names, that may be referred to by the operating system or by more than one separately compiled procedure.

Examples of identifiers that could be used for names or labels:

```
A

FILE2

LOOP_3

RATE_OF_PAY

#32
```

The Use of Blanks

Blanks may be used freely throughout a PL/I program. They may or may not surround operators and most other delimiters. In general, any number of blanks may appear wherever one blank is allowed, such as between words in a statement.

One or more blanks must be used to separate identifiers and constants that are not separated by some other delimiter or by a comment. However, identifiers, constants (except character-string constants) and composite operators (for example, ¬=) cannot contain blanks.

Other cases that require or permit blanks are noted in the text where the feature of the language is discussed. See Figure 2-1 for examples.

AB+BC	is equivalent to	AB + BC
TABLE(10)	is equivalent to	TABLE (10)
FIRST,SECOND	is equivalent to	FIRST, SECOND
ATOB	is <u>not</u> equivalent to	A TO B

Figure 2-1. Examples of the Use of Blanks

Comments

Comments are permitted wherever blanks are allowed in a program, except within data items, such as a character-string constant. A comment is treated as a blank and can therefore be used in place of a required separating blank. Comments do not otherwise affect execution of a program; they are used only for documentation purposes. Comments may be punched into the same cards as statements, either inserted between statements or in the middle of them.

The general format of a comment is:

```
/*   character-string   */
```

The character pair /* indicates the beginning of a comment. The same character pair reversed, */, indicates its end. No blanks or other characters can separate the two characters of either pair; the slash and the asterisk must be immediately adjacent. The comment itself may contain any characters except the */ combination, which would be interpreted as terminating the comment.

Example:

```
/* THIS WHOLE SENTENCE COULD BE
   INSERTED AS A COMMENT */
```

Any characters permitted for a particular machine configuration may be used in comments.

BASIC PROGRAM STRUCTURE

A PL/I program is constructed from basic program elements called statements. There are two types of statements: simple and compound. These statements make up larger program elements called groups and blocks.

SIMPLE AND COMPOUND STATEMENTS

There are three types of simple statements: keyword, assignment, and null, each of which contains a statement body that is terminated by a semicolon.

A keyword statement has a keyword to indicate the function of the statement; the statement body is the remainder of the statement.

The assignment statement contains the assignment symbol (=) and does not have a keyword.

The null statement consists only of a semicolon and indicates no operation; the semicolon is the statement body.

Examples of simple statements are:

```
GOTO LOOP_3; (GOTO is a keyword; a blank
             between GO and TO is optional.
             The statement body is LOOP_3;)

A = B + C;   (assignment statement)
```

A compound statement is a statement that contains one or more other statements as a part of its statement body. There are two compound statements: the IF statement and the ON statement. The final statement of a compound statement is a simple statement that is terminated by a semicolon. Hence, the compound statement is terminated by this semicolon. Examples of the two compound statements are:

1. IF A>B THEN A = B+C; ELSE GO TO LOOP_3;

   This example can also be written as follows:

   ```
   IF A>B
       THEN A=B+C;
       ELSE GO TO LOOP_3;
   ```

2. ON UNDERFLOW GO TO UNFIX;

3. ON UNDERFLOW;

In example 3, the contained statement is the null statement represented by a semicolon only; it indicates that no action is to be taken when an UNDERFLOW interrupt occurs.

Statement Prefixes

Both simple and compound statements may have one or more prefixes. There are two types of prefixes; the label prefix and the condition prefix.

A label prefix identifies a statement so that it can be referred to at some other point in the program. A label prefix is an identifier that precedes the statement and is connected to the statement by a colon. Most statements may have one or more labels. If more than one is specified, they may be used interchangeably to refer to that statement. PROCEDURE and ENTRY statements must have one and only one label.

A condition prefix specifies whether or not program interrupts are to result from the occurrence of the named conditions. Condition names are language keywords, each of which represents an exceptional condition that might arise during execution of a program. Examples are OVERFLOW and SIZE. The OVERFLOW condition arises when the exponent of a floating-point number exceeds the maximum allowed (representing a maximum value of about 10^{75}). The SIZE condition arises when a value is assigned to a variable with loss of high-order digits or bits.

A condition name in a condition prefix may be preceded by the word NO to indicate that, effectively, no interrupt is to occur if the condition arises. If NO is used, there can be no intervening blank between the NO and the condition name.

A condition prefix consists of a list of one or more condition names, separated by commas and enclosed in parentheses. Only one condition prefix may be attached to a statement, and the parenthesized list must be followed by a colon. A condition prefix precedes the entire statement, including any possible label prefixes for the statement.

Example:

```
(SIZE,NOOVERFLOW):COMPUTE:A = B * C ** D;
```

The condition prefix indicates that an interrupt is to occur if the SIZE condition arises during execution of the assignment statement, but that no interrupt is to occur if the OVERFLOW condition arises. Note that the condition prefix precedes the label prefix COMPUTE.

Since intervening blanks between a prefix and its associated statement are ignored, it is often convenient to punch the condition prefix into a separate card that precedes the card into which the statement is punched. Thus, after debugging, the prefix can be easily removed. For example:

```
(SIZE,NOOVERFLOW):

COMPUTE:  A = B * C ** D;
```

Condition prefixes are discussed in Chapter 11, "Exceptional Condition Handling and Program Checkout."

GROUPS AND BLOCKS

A group is a sequence of statements headed by a DO statement and terminated by a corresponding END statement. It is used for control purposes. A group also may be called a DO-group.

A block is a sequence of statements that defines an area of a program. It is used to delimit the scope of a name and for control purposes. A program may consist of one or more blocks. Every statement must appear within a block. There are two kinds of blocks: begin blocks and procedure blocks. A begin block is delimited by a BEGIN statement and an END statement. A procedure block is delimited by a PROCEDURE statement and an END statement. Every begin block must be contained within some procedure block.

Execution passes sequentially into and out of a begin block. However, a procedure block must be invoked by execution of a statement in another block. The first procedure in a program to be executed (sometimes called the main or initial procedure) is invoked automatically by the operating system. For System/360 implementations, this first procedure must be identified by specifying OPTIONS (MAIN) in the PROCEDURE statement.

Data is generally defined as a representation of information or of value.

In PL/I, reference to a data item, arithmetic or string, is made by using either a variable or a constant (the terms are not exactly the same as in general mathematical usage).

A variable is a symbolic name having a value that may change during execution of a program.

A constant (which is not a symbolic name) has a value that cannot change.

The following statement has both variables and constants:

```
AREA = RADIUS**2*3.1416;
```

AREA and RADIUS are variables; the numbers 2 and 3.1416 are constants. The value of RADIUS is a data item, and the result of the computation will be a data item that will be assigned as the value of AREA. The number 3.1416 in the statement is itself the data item; the characters 3.1416 also are written to refer to the data item.

If the number 3.1416 is to be used in more than one place in the program, it may be convenient to represent it as a variable to which the value 3.1416 has been assigned. Thus, the above statement could be written as:

```
PI = 3.1416;

AREA = RADIUS**2*PI;
```

In this statement, only the digit 2 is a constant.

In preparing a PL/I program, the programmer must be familiar with the types of data that are permitted, the ways in which data can be organized, and the methods by which data can be referred to. The following paragraphs discuss these features.

DATA TYPES

The types of data that may be used in a PL/I program fall into two categories: problem data and program control data. Problem data is used to represent values to be processed by a program. It consists of the arithmetic and string data types. Program control data is used by the programmer to control the execution of his program. Statement labels and pointers are the types of program control data.

A constant does more than state a value; it demonstrates various characteristics of the data item. For example, 3.1416 shows that the data type is arithmetic and that the data item is a decimal number of five digits and that four of these digits are to the right of the decimal point.

The characteristics of a variable are not immediately apparent in the name. Since these characteristics, called attributes, must be known, certain keywords and expressions may be used to specify the attributes of a variable in a DECLARE statement. The attributes used to describe each data type are discussed briefly in this chapter. A complete discussion of each attribute appears in Part II, Section I, "Attributes."

PROBLEM DATA

The types of problem data are arithmetic and string.

ARITHMETIC DATA

An item of arithmetic data is one with a numeric value. Arithmetic data items have the characteristics of base, scale, and precision. The characteristics of data items represented by an arithmetic variable are specified by attributes declared for the name, or assumed by default.

The base of an arithmetic data item is either decimal or binary.

The scale of an arithmetic data item is either fixed-point or floating-point. A decimal fixed-point data item is a number in which the position of the decimal point is specified, either by its appearance in a constant or by a scale factor declared for a variable. A binary fixed-point data item cannot, in general, contain a binary point; a binary point is assumed to be at the right of the rightmost digit in the item. (The D-Compiler does not allow the specification of a scale factor for fixed-point

binary items; however, certain mathematical operations involving fixed-point binary operands maintain an actual binary point -- e.g., fixed-point binary division -- so that fractional binary digits can occur in the result of such an operation. These exceptions are discussed in Chapter 4, "Expressions.")

A floating-point data item is a number followed by an optionally signed integer exponent. The exponent specifies the assumed position of the decimal or binary point, relative to the position in which it appears.

The precision of an arithmetic data item is the number of digits the data item can contain, in the case of fixed-point, or the minimum number of significant digits (excluding the exponent) to be maintained, in the case of floating-point. For decimal fixed-point data items, precision can also specify the assumed position of the decimal point, relative to the rightmost digit of the number.

Base and scale of arithmetic variables are specified by keywords; precision is specified by parenthesized decimal integer constants.

Whenever a data item is assigned to a fixed-point variable, the precision declared for that variable is maintained. The assigned item is aligned on the decimal or assumed binary point of the variable. Leading zeros are inserted if the assigned decimal or binary item contains fewer integer digits than declared; trailing zeros are inserted if an assigned decimal item contains fewer fractional digits than declared. A SIZE error may occur if the assigned item contains too many integer digits; truncation on the right may occur if it contains too many fractional digits. Note that since the value represented by a binary fixed-point variable can have no fractional digits, any fractional digits contained in a binary fixed-point item assigned to such a variable are always truncated; thus, a binary fixed-point variable always represents an integer value.

In the following sections, the arithmetic data types discussed are decimal fixed-point, sterling fixed-point, binary fixed-point, decimal floating-point, and binary floating-point.

Decimal Fixed-Point Data

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit. In most uses, a sign may optionally precede a decimal fixed-point constant.

Examples of decimal fixed-point constants as written in a program are:

```
3.1416

455.3

732

003

5280

.0012
```

The keyword attributes for declaring decimal fixed-point variables are DECIMAL and FIXED. Precision is stated by two unsigned decimal integer constants, separated by a comma and enclosed in parentheses. The first specifies the total number of digits; the second, the scale factor, specifies the number of digits to the right of the decimal point. If the variable is to represent integers, the scale factor and its preceding comma can be omitted. The attributes may appear in any order, but the precision specification must follow either DECIMAL or FIXED.

Following are examples of declarations of decimal fixed-point variables:

```
DECLARE A FIXED DECIMAL (5,4);

DECLARE B FIXED (6,0) DECIMAL;
```

The first DECLARE statement specifies that the identifier A is to represent decimal fixed-point items of not more than five digits, four of which are to be treated as fractional, that is, to the right of the assumed decimal point. Any item assigned to A will be converted to decimal fixed-point and aligned on the decimal point. The second DECLARE statement specifies that B is to represent integers of no more than 6 digits. Note that the comma and the zero are unnecessary; it could have been specified B FIXED (6) DECIMAL.

The maximum number of decimal digits allowed for System/360 implementations is 15. Default precision, assumed when no specification is made, is (5,0). The internal coded arithmetic form of decimal fixed-point data is packed decimal. Packed decimal is stored two digits to the byte, with a sign indication in the rightmost four 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

may specify the number of digits (p) as an even number. Any such extra digit is in the high-order position, and it participates in any operations performed upon the data item, such as in a comparison operation. (Note that any arithmetic overflow into such an extra high-order digit position can be detected only if the SIZE condition is enabled.)

Sterling Fixed-Point Data

PL/I has a facility for handling constants stated in terms of sterling currency value. The data may be written in a program with pounds, shillings, and pence fields, each separated by a period. Such data is converted and maintained internally as a decimal fixed-point number representing the equivalent in pence. A sterling data constant ends with the letter L, representing the pounds symbol. All three fields (pounds, shillings, and pence) must be present in a sterling constant. Note that the pence field is one or more decimal digits with an optional decimal point (the integral part must be less than 12 and must contain at least one digit)--see the third example below.

Examples of sterling fixed-point constants as written in a program are:

```
101.13.8L

1.10.0L

0.0.2.5L

2.4.6L
```

The third example represents twopence-halfpenny. The last example represents two pounds, four shillings, and six pence. It is converted and stored internally as 534 (pence).

There are no keyword attributes for declaring sterling variables, but a variable can be declared with a sterling picture, or sterling values may be expressed in pence as decimal fixed-point data. The precision of a sterling constant is the precision of its value expressed in pence.

Binary Fixed-Point Data

A binary fixed-point constant consists of one or more binary digits, followed immediately by the letter B, with no intervening blank. It cannot contain a binary point; a point is always assumed to follow the rightmost binary digit. In most uses, a sign may optionally precede the constant.

Examples of binary fixed-point constants as written in a program are:

```
10110B

11111B

101B
```

The keyword attributes for declaring binary fixed-point variables are BINARY and FIXED. Precision is specified by a decimal integer, enclosed in parentheses, to represent the maximum number of binary digits that the variable can contain. A binary fixed-point variable always represents an integer. The attributes can appear in any order, but the precision specification must follow either BINARY or FIXED.

Following is an example of declaration of a binary fixed-point variable:

```
DECLARE FACTOR BINARY FIXED (20);
```

FACTOR is declared to be a variable that can represent arithmetic data items as large as 20 binary digits.

The maximum number of binary digits allowed for System/360 implementations is 31. The default precision for the D-Compiler is (15). The internal coded arithmetic form of binary fixed-point data is a fixed-point binary full word. A full word is 31 bits plus a sign bit. Any binary fixed-point data item is always stored as 31 digits, even though the declaration of the variable may specify fewer digits. The declared number of digits are considered to be in the low-order positions, but the extra high-order digits participate in any operations performed upon the data item. (Note that any arithmetic overflow into such extra high-order digit positions can be detected only if the SIZE condition is enabled.)

An identifier for which no declaration is made is assumed to be a binary fixed-point variable, with default precision, if its first letter is any of the letters I through N.

Decimal Floating-Point Data

A decimal floating-point constant is written as a field of decimal digits followed by the letter E, followed by an

optionally signed decimal integer exponent that specifies a power of ten. The first field of digits may contain a decimal point. The entire constant may be preceded by a plus or minus sign. Examples of decimal floating-point constants as written in a program are:

```
15E-23

15E23

4E-3

48333E65

438E0

3141593E-6

.003141593E3
```

The last two examples represent the same value.

The keyword attributes for declaring decimal floating-point variables are DECIMAL and FLOAT. Precision is stated by a decimal integer constant enclosed in parentheses. It specifies the minimum number of significant digits to be maintained. If an item assigned to a variable has a field width larger than the declared precision of the variable, truncation may occur on the right. The least significant digit is the first that is lost. Attributes may appear in any order, but the precision specification must follow DECIMAL or FLOAT.

Following is an example of declaration of a decimal floating-point variable:

```
DECLARE LIGHT_YEARS DECIMAL FLOAT(5);
```

This statement specifies that LIGHT_YEARS is to represent decimal floating-point data items with an accuracy of at least five significant digits.

The maximum precision allowed for decimal floating-point data items for System/360 implementations is (16); the exponent cannot exceed two digits. A value range of approximately 10^{-78} to 10^{75} can be expressed by a decimal floating-point data item. Default precision is (6). The internal coded arithmetic form of decimal floating-point data is normalized hexadecimal floating-point, with the point assumed to the left of the first hexadecimal digit. 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), long floating-point form is used.

An identifier for which no declaration is made is assumed to be a decimal floating-point variable if its first letter is any of the letters A through H, O through Z, or one of the alphabetic extenders, $, #, @.

Binary Floating-Point Data

A binary floating-point constant consists of a field of binary digits followed by the letter E, followed by an optionally signed decimal integer exponent followed by the letter B. The exponent is a string of decimal digits and specifies an integral power of two. The field of binary digits may contain a binary point. A binary floating-point constant may be preceded by a plus or minus sign. Examples of binary floating-point constants as written in a program are:

```
101101E5B

101.101E2B

11101E-28B
```

The keyword attributes for declaring binary floating-point variables are BINARY and FLOAT. Precision is expressed as a decimal integer constant, enclosed in parentheses, to specify the minimum number of significant digits to be maintained. The attributes can appear in any order, but the precision specification must follow either BINARY or FLOAT. Following is an example of declaration of a binary floating-point variable:

```
DECLARE S BINARY FLOAT (16);
```

This specifies that the identifier S is to represent binary floating-point data items with 16 digits in the binary field.

The maximum precision allowed for binary floating-point data items for System/360 implementations is (53); default precision is (21). The exponent cannot exceed three decimal digits. A value range of approximately 2^{-260} to 2^{252} can be expressed by a binary floating-point data item. The internal coded arithmetic form of binary floating-point data is normalized hexadecimal floating-point. 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), long floating-point form is used.

Numeric Character Data

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

A numeric picture specification describes a string of characters to which only data that has an arithmetic value is to be assigned. A numeric picture specification cannot contain the picture character X, which is used only for non-numeric pictures. The basic form of a numeric picture specification is one or more occurrences of the picture character 9 and an optional occurrence of the picture character V, to indicate the assumed location of a decimal point. The picture specification must be enclosed in single quotation marks. For example:

```
'999V99'
```

This numeric picture specification describes a data item consisting of up to five decimal digits in character form, with a decimal point assumed to precede the rightmost two digits.

Repetition factors may be used in numeric picture specifications. A repetition factor is a decimal integer constant, enclosed in parentheses, that indicates the number of repetitions of the immediately following picture character. For example, the following picture specification would result in the same description as the example shown above:

```
'(3)9V(2)9'
```

The format for declaring a numeric character variable is:

```
DECLARE identifier PICTURE
    'numeric-picture-specification';
```

For example:

```
DECLARE PRICE PICTURE '999V99';
```

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

The numeric picture specification can specify all of the arithmetic attributes of data in much the same way that they are specified by the appearance of a constant. Only decimal numeric data can be represented by picture characters.

It is important to note that, although numeric character data has arithmetic attributes, it is not stored in coded arithmetic form. In System/360 implementations, numeric character data is stored in zoned decimal format; before it can be used in arithmetic computations, it must be converted either to packed decimal or to hexadecimal floating-point format. Such conversions are done automatically, but they require extra execution 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 data item is assigned to a character string, the editing characters are included in the assignment. If, however, a numeric character item is assigned to another numeric character or arithmetic variable, the editing characters will not be included in the assignment; only the actual digits and the location of the assumed decimal point are assigned. (Note that character-string data cannot be assigned to numeric character variables.)

Consider the following example:

```
DECLARE PRICE PICTURE '$99V.99',
        COST CHARACTER(6),
        VALUE 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. They are not, however, a part of its arithmetic value. After execution of the second assignment statement, the actual internal character representation of PRICE and COST can be considered identical. If they were assigned to character strings, which were then printed, they would look exactly the same. They do not, however, always function the same. For example:

```
VALUE = PRICE;

COST = PRICE;

VALUE = COST;

PRICE = COST;
```

After the first two assignment statements are executed, the value of VALUE would be 001228 (with an assumed decimal point before the last two digits) and the value of COST would be '$12.28'. In the assignment of PRICE to VALUE, the currency symbol and the decimal point are considered to be editing characters, and they are not part of the assignment; the arithmetic 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.

The third and fourth assignment statements are invalid. The value of COST cannot be assigned to VALUE because a character string cannot be converted to coded arithmetic. The value of COST cannot be assigned to PRICE because a character string cannot be converted to numeric character.

Other editing characters, including zero suppression characters, drifting characters, and insertion characters, can be used in numeric picture specifications. For complete discussions of picture characters, see Part II, Section D, "Picture Specification Characters" and the discussion of the PICTURE attribute in Part II, Section I, "Attributes."

STRING DATA

A string is a contiguous sequence of characters (or binary digits) that is treated as a single data item. The length of the string is the number of characters (or binary digits) it contains.

There are two types of strings: character strings and bit strings.

Character-String Data

A character string can include any digit, letter, or special character recognized as a character by the particular machine configuration. Any blank included in a character string is considered an integral character of the data item and is included in the count of length. A comment that is inserted within a character string will not be recognized as a comment. The comment, as well as the comment delimiters (/* and */), will be considered to be part of the character-string data.

Character-string constants, when written in a program, must be enclosed in single quotation marks. If a single quotation mark is a character in a string, it must be written as two single quotation marks with no intervening blank. The length of a character string is the number of characters between the enclosing quotation marks. If two single quotation marks are used within the string to represent a single quotation mark, they are counted as a single character.

Examples of character-string constants are:

```
'LOGARITHM TABLE'

'PAGE 5'

'SHAKESPEARE''S ''''HAMLET'''''

'AC438-19'

(2)'WALLA '
```

The third example actually indicates SHAKESPEARE'S "HAMLET" with a length of 24. In the last example, the parenthesized number is a repetition factor which indicates repetition of the characters that follow. This example specifies the actual constant 'WALLA WALLA ' (the blank is included as one of the characters to be repeated). The repetition factor must be an unsigned decimal integer constant, enclosed in parentheses.

The keyword attribute for declaring a character-string variable is CHARACTER. Length is declared by a decimal integer constant, enclosed in parentheses, which specifies the number of characters in the string. The length specification must follow the keyword CHARACTER. For example:

```
DECLARE NAME CHARACTER(15);
```

This DECLARE statement specifies that the identifier NAME is to represent character-string data items, 15 characters in length. If a character string shorter than 15 characters were to be assigned to NAME, it would be left adjusted and padded on the right with blanks to a length of 15. If a longer string were assigned, it would be truncated on the right. (Note: If such truncation occurs, no interrupt will result

as it might for truncation of arithmetic data; there is no ON-condition in PL/I to deal with string truncation.)

Character-string data in System/360 implementations is maintained internally in character format, that is, each character occupies one byte of storage. The maximum length allowed by the D-Compiler for variables declared with the CHARACTER attribute is 255. The maximum length allowed for a character-string constant after application of repetition factors is also 255. The minimum length in either case is one.

Character-string variables also can be declared using the PICTURE attribute of the form:

```
PICTURE 'character-picture-specification'
```

The character picture specification is a string composed entirely of the picture specification character X. The string of X picture characters must be enclosed in single quotation marks. The character X specifies that any character may appear in the corresponding position in the field. For example:

```
DECLARE PART_NO PICTURE 'XXXXXXXXXX';
```

This DECLARE statement specifies that the identifier PART_NO will represent character-string data items consisting of any ten characters.

Repetition factors are used in picture specifications differently from the way they are used in string constants. They must be placed inside the quotation marks. The repetition factor specifies repetition of the immediately following picture character. For example, the above picture specification could be written:

```
'(10)X'
```

The maximum length allowed for a picture specification is the same as that allowed for character-string constants, as discussed above.

Bit-String Data

A bit-string constant is written in a program as a series of binary digits enclosed in single quotation marks and followed immediately by the letter B.

Examples of bit-string constants as written in a program are:

```
'1'B

'11111010110001'B

(64)'0'B
```

The parenthesized number in the last example is a repetition factor which specifies that the following series of digits is to be repeated the specified number of times. The repetition factor must be an unsigned decimal integer constant enclosed in parentheses. The example shown would result in a string of 64 binary zeros.

A bit-string variable is declared with the BIT keyword attribute. Length is specified by a decimal integer constant, enclosed in parentheses, to specify the number of binary digits in the string. The letter B is not included in the length specification since it is not an actual part of the string. The length specification must follow the keyword BIT. Following is an example of declaration of a bit-string variable:

```
DECLARE SYMPTOMS BIT (64);
```

Like character strings, bit strings are assigned to variables from left to right. If a string is longer than the length declared for the variable, the rightmost digits are truncated; if shorter, padding, on the right, is with zeros.

With System/360 implementations, bit strings are stored eight bits to a byte, and each string is aligned on a byte boundary. The maximum length allowed for a bit-string variable with the D-Compiler is 64. The maximum length allowed for a bit-string constant after application of repetition factors is also 64. The minimum length in either case is one.

PROGRAM CONTROL DATA

The types of program control data are label and pointer.

LABEL DATA

Label data is a type of program control data. A label data item is a label constant or the value of a label variable.

A label constant is an identifier written as a prefix to a statement so that, during execution, program control can be transferred to that statement through a reference to its label. A colon connects the label to the statement.

```
ABCDE: DISTANCE = RATE*TIME;
```

In this example, ABCDE is the statement label. The statement can be executed either by normal sequential execution of instructions or by transferring control to this statement from some other point in the program by means of a GO TO statement.

As used above, ABCDE can be classified further as a statement-label constant. A statement-label variable is an identifier that refers to statement-label constants. Consider the following example:

```
LBL_A:   statement;
            .
            .
            .
LBL_B:   statement;
            .
            .
            .
         LBL_X = LBL_A;
                 .
                 .
                 .
         GO TO LBL_X;
                 .
                 .
                 .
```

LBL_A and LBL_B are statement-label constants because they are prefixed to statements. LBL_X is a statement-label variable. By assigning LBL_A to LBL_X, the statement GO TO LBL_X causes a transfer to the LBL_A statement. Elsewhere, the program may 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.

A statement-label variable must be declared with the LABEL attribute, as follows:

```
DECLARE LBL_X LABEL;
```

POINTER DATA

Pointer data is a type of program control data. A pointer data item is the value of a pointer variable; it cannot be written as a constant.

A pointer variable is the name of a pointer and is used in connection with variables of the based storage class. The value of a pointer variable is, in effect, an address of data in storage.

The keyword attribute for declaring pointer variables is POINTER. For information on the use of pointer variables, see Chapter 8, "Input and Output," and Chapter 12, "Based Variables and Pointer Variables."

DATA ORGANIZATION

In PL/I, data items may be single data elements, or they may be grouped together to form data collections called arrays and structures. A variable that represents a single element is an element variable (also called a scalar variable). A variable that represents a collection of data elements is either an array variable or a structure variable.

Any type of data -- arithmetic, string, label, or pointer -- can be collected into arrays or structures.

ARRAYS

Data elements having the same characteristics, that is, of the same data type and of the same precision or length, may be grouped together to form an array. An array is an n-dimensional collection of elements, all of which have identical attributes. Only the array itself is given a name. An individual item of an array is referred to by giving its relative position within the array.

Consider the following two declarations:

```
DECLARE LIST (8) FIXED DECIMAL (3);

DECLARE TABLE (4,2) FIXED DECIMAL (3);
```

In the first example, LIST is declared to be a one-dimensional array of eight elements, each of which is a fixed-point decimal item of three digits. In the second example, TABLE is declared to be a two-dimensional array, also of eight fixed-point decimal elements.

The parenthesized number or numbers following the array name in a DECLARE statement is the dimension attribute specification. It must follow the array name, with or without an intervening blank. It specifies the number of dimensions of the array and the bound, or extent, of each dimension. Since only one bound specification appears for LIST, it is a one-dimensional array. Two bound specifications, separated by a comma, are listed for TABLE; consequently, it is declared to be a two-dimensional array.

The bound of a dimension is the end of that dimension; the beginning of a dimension is always assumed to be 1. The extent of a dimension is the number of integers between, and including, 1 and the specified end. Thus, the terms bound and extent, while conceptually different, have the same value in the PL/I subset. For example, the one dimension of LIST has a bound of 8, and hence, its extent is 8. The two dimensions of TABLE have bounds of 4 and 2; the extents are also 4 and 2.

The bounds of an array determine the way elements of the array can be referred to. For example, assume that the following data items are assigned to the array LIST, as declared above:

```
20 5 10 30 630 150 310 70
```

The different elements would be 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 numbers following the name LIST is a subscript. A parenthesized subscript following an array name, with or without an intervening blank, specifies the relative position of a data item within the array. 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. In this case, LIST is an array variable. Note the difference between a subscript and the dimension attribute specification. The latter, which appears in a declaration, specifies the dimensionality and the number of elements in an array. Subscripts are used in other references to identify specific elements within the array.

Assume that the same data were assigned to TABLE, which is declared as a two-dimensional array. TABLE can be illustrated as a matrix of four rows and two columns, as follows:

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, in this case, the data item 10.

Note: The use of a matrix to illustrate TABLE is purely conceptual. It has no relationship to the way in which the items are actually organized in storage. Data items are assigned to an array in row major order, that is, with the rightmost subscript varying 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.

Arrays are not limited to two dimensions. The PL/I D-Compiler allows a maximum of three dimensions to be declared for an array. In a reference to an element of any array, a subscripted name must contain as many subscripts as there are dimensions in the array.

Examples of arrays in this section have shown arrays of arithmetic data. Other data types may be collected into arrays. String arrays, either character or bit, are valid, as are arrays of statement labels and arrays of pointers.

Expressions as Subscripts

The subscripts of a subscripted name need not be constants. Any expression that yields a valid arithmetic value can be used. If the evaluation of such an expression does not yield an integer value, the fractional portion is ignored. For System/360 implementations, the integer value is converted, if necessary, to a fixed-point binary number of precision (15,0), since subscripts are maintained internally as binary integers.

Subscripts are frequently expressed as variables or other expressions. Thus, TABLE(I,J*K) could be used to refer to the different elements of TABLE by varying the values of I, J, and K.

Note that although a subscript can be an expression, each bound of a dimension

attribute declaration must be an unsigned decimal integer constant. Also note that the value of a subscript must lie within the extent of the corresponding dimension; otherwise, it is an error.

STRUCTURES

Data items that need not have identical characteristics, but that possess a logical relationship to one another, can be grouped into aggregates called structures.

Like an array, the entire structure is given a name that can be used to refer to the entire collection of data. Unlike an array, however, each element of a structure also has a name.

A structure is a hierarchical collection of names. At the bottom of the hierarchy is a collection of elements, each of which represents a single data item or an array. At the top of the hierarchy is the structure name, which represents the entire collection of elements. For example, the following is a collection of element variables that might be used to compute a weekly payroll:

```
LAST_NAME
FIRST_NAME
REGULAR_HOURS
OVERTIME_HOURS
REGULAR_RATE
OVERTIME_RATE
```

These variables could be collected into a structure and given a single structure name, PAYROLL, which would refer to the entire collection.

PAYROLL

LAST_NAME	REGULAR_HOURS	REGULAR_RATE
FIRST_NAME	OVERTIME_HOURS	OVERTIME_RATE

Any reference to PAYROLL would be a reference to all of the element variables. For example:

```
GET EDIT (PAYROLL) (format-list);
```

This input statement could cause data to be assigned to each of the element variables of the structure PAYROLL.

It often is convenient to subdivide the entire collection into smaller logical collections. In the above examples, LAST_NAME and FIRST_NAME might make a logical subcollection, as might REGULAR_HOURS and OVERTIME_HOURS, as well as REGULAR_RATE and OVERTIME_RATE. In a structure, such subcollections also are given names.

PAYROLL

NAME	HOURS	RATE
FIRST	REGULAR	REGULAR
LAST	OVERTIME	OVERTIME

Note that the hierarchy of names can be considered to have different levels. At the first level is the major structure name; at a deeper level are the minor structure names; and at the deepest level are the elementary names. An elementary name in a structure can represent an array, in which case it is not an element variable, but an array variable.

The organization of a structure is specified in a DECLARE statement through the use of level numbers. A major structure name must be declared with the level number 1. Minor structures and elementary names must be declared with level numbers arithmetically greater than 1; they must be decimal integer constants. A blank must separate the level number and its associated name.

For example, the items of a weekly payroll could be declared as follows:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST,
            3 FIRST,
          2 HOURS,
            3 REGULAR,
            3 OVERTIME,
          2 RATE,
            3 REGULAR,
            3 OVERTIME;
```

Note: In an actual declaration of the structure PAYROLL, attributes would be specified for each of the elementary names. The pattern of indention in this example is used only for readability. The statement could be written in a continuous string as DECLARE 1 PAYROLL, 2 NAME, 3 LAST, etc.

PAYROLL is declared as a major structure containing the minor structures NAME, HOURS, and RATE. Each minor structure contains two elementary names. A programmer can refer to the entire structure by the name PAYROLL, or he can refer to portions of the structure by referring to the minor structure names. He can refer to an element of the structure by referring to an elementary name.

Note that in the declaration, each level number precedes its associated name and is separated from the name by a blank. The numbers chosen for successively deeper

levels need not be the immediately succeeding integers. They are used merely to specify the relative level of a name. 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. A major structure description is terminated by the declaration of another item with a level number 1 (i.e., another major structure), by the declaration of another item with no level number, or by a semicolon terminating the DECLARE statement. PAYROLL might have been declared as follows:

```
DECLARE 1 PAYROLL, 4 NAME, 5 LAST, 5 FIRST,
          2 HOURS, 6 REGULAR, 5 OVERTIME,
          2 RATE, 3 REGULAR, 3 OVERTIME;
```

This declaration would result in exactly the same structuring as the previous declaration.

Level numbers are specified with structure names only in DECLARE statements. In references to the structure or its elements, no level numbers are used. Only structures can be declared with level numbers; a level number cannot be declared with any other identifier.

Qualified Names

A minor structure or a structure element can be referred to by the minor structure name or the elementary name alone if there is no ambiguity. Note, however, that each of the names REGULAR and OVERTIME appears twice in the structure declaration for PAYROLL. A reference to either name would be ambiguous without some qualification to make the name unique.

PL/I allows the use of qualified names to avoid this ambiguity. A qualified name is an elementary name or a minor structure name that is made unique by qualifying it with one or more names at a higher level. In the PAYROLL example, REGULAR and OVERTIME could be made unique through use of the qualified names HOURS.REGULAR, HOURS. OVERTIME, RATE.REGULAR, and RATE.OVERTIME.

The different names of a qualified name are connected by periods. Blanks may or may not appear surrounding the period. Qualification is in the order of levels; that is, the name at the highest level must appear first, with the name at the deepest level appearing last.

Any of the names in a structure, except the major structure name itself, need not be unique within the procedure in which it is declared. For example, the qualified name PAYROLL.HOURS.REGULAR might be required to make the reference unique (another structure, say WORK, might also have the name REGULAR in a minor structure HOURS; it could be made unique with the name WORK. HOURS. REGULAR). All of the qualifying names need not be used, although they may be, if desired. Qualification need go only so far as necessary to make the name unique. Intermediate qualifying names can be omitted. The name PAYROLL.LAST is a valid reference to the name PAYROLL.NAME.LAST.

ARRAYS OF STRUCTURES

Arrays of structures are not supported by the D-Compiler; however, simulation of arrays of structures is possible. The publication IBM System/360 Disk and Tape Operating Systems, PL/I Programmer's Guide, Form C24-9005, offers some techniques for this simulation.

OTHER ATTRIBUTES

Keyword attributes for data variables such as BINARY and DECIMAL are discussed briefly in the preceding sections of this chapter. Other attributes that are not peculiar to one data type may also be applicable. A complete discussion of these attributes is contained in Part II, Section I, "Attributes." Some that are especially applicable to a discussion of data type and data organization are ALIGNED, PACKED, and DEFINED.

The ALIGNED and PACKED Attributes

The ALIGNED and PACKED attributes are used to specify the arrangement in storage of string or numeric character elements within structures or arrays. If the PACKED attribute is specified for an array or a structure, all character string and numeric character elements must, whenever possible, be stored in adjacent character positions. Bit strings cannot be packed; hence, an array or structure containing bit-string elements cannot have the PACKED attribute. Thus, an array or structure containing bit strings must explicitly be given the ALIGNED attribute.

If the ALIGNED attribute is specified for an array or a structure, each bit string, character string, or numeric character element must be aligned on a particular storage boundary, if that alignment is more efficient for program execution.

Packed aggregates can be useful for overlay defining. (See the discussion of the DEFINED attribute immediately following this section.) Aligned aggregates make it possible for the implementation to speed up the execution of the program, but at some cost in data storage. Since System/360 has character-handling instructions, there is no need to align character strings. Furthermore, alignment of character strings or numeric character fields prohibits the use of overlay defining and the STRING built-in function for them.

Arrays are assumed to have the ALIGNED attribute and structures are assumed to have the PACKED attribute, unless they are declared otherwise.

The DEFINED Attribute

The DEFINED attribute specifies that the named data element, structure, or array is to refer to the same storage area as that assigned to other data. For example:

```
DECLARE LIST (100,100),
        LIST_A (100,100) DEFINED LIST;
```

In the above declaration, LIST is a 100 by 100 two-dimensional array. LIST_A is an identical array defined on LIST. The result is that a reference to an element in LIST_A is the same as a reference to the corresponding element in LIST. Thus, a change to an element in LIST_A will be reflected in the corresponding element of LIST, and vice versa. This type of defining is called correspondence defining.

Another type of defining is called overlay defining. This type of defining specifies that the defined item (the item having the DEFINED attribute; e.g., LIST_A above) is to refer to all or part of the storage occupied by the base identifier (the identifier following the keyword DEFINED; e.g., LIST above). For example:

```
DECLARE 1 P, 2 Q CHARACTER (25),
             2 R CHARACTER (50),
       PSTRING1 CHARACTER (60)
                DEFINED P;
```

In this example, PSTRING1 is a character string of length 60 defined on the packed structure P (P has the PACKED attribute by default). Since P is packed, the first character in Q through the last character in R can be considered as one string of 75 characters in length. PSTRING1 refers to the first 60 characters of that string, that is, the 25 characters of Q effectively concatenated with the first 35 characters of R. Note that if P were not packed, the contents of PSTRING1 could not be guaranteed.

An expression is a representation of a value. A single constant or a variable is an expression. Combinations of constants and/or variables, along with operators and/or parentheses, are expressions. An expression that contains operators is an operational expression. The constants and variables of an operational expression are called operands.

Examples of expressions are:

```
27

LOSS

A+B

(SQTY-QTY)*SPRICE
```

Any expression can be classified as an element expression (also called a scalar expression), an array expression, or a structure expression. An element expression is one that represents an element value. An array expression is one that represents an array value. A structure expression is one that represents a structure value.

Array variables and structure variables cannot appear in the same expression. Element variables and constants, however, can appear in either array expressions or structure expressions. An elementary name within a structure or a subscripted name that specifies a single element of an array is an element expression.

Note: If an elementary name of a structure is given the dimension attribute, that elementary name is an array variable and can appear only in array expressions.

In the examples below, assume that the variables have attributes declared as follows:

```
DECLARE A(10,10) BINARY FIXED (31),
   B(10,10) BINARY FIXED (31),
   1 RATE, 2 PRIMARY DECIMAL FIXED (4,2),
           2 SECONDARY DECIMAL FIXED (4,2),
   1 COST, 2 PRIMARY DECIMAL FIXED (4,2),
           2 SECONDARY DECIMAL FIXED (4,2),
   C BINARY FIXED (15),
   D BINARY FIXED (15);
```

Examples of element expressions are:

```
C * D

A(3,2) + B(4,8)

RATE . PRIMARY - COST . PRIMARY

A(4,4) * C

RATE . SECONDARY / 4

A(4,6) * COST . SECONDARY
```

All of these expressions are element expressions because each operand is an element variable or constant (even though some may be elements of arrays or elementary names of structures); hence, each expression represents an element value.

Examples of array expressions are:

```
A + B

A * C - D

B / 10B
```

All of these expressions are array expressions because at least one operand of each is an array variable; hence, each expression represents an array value. Note that the third example contains the binary fixed-point constant 10B.

Examples of structure expressions are:

```
RATE * COST

RATE / 2
```

Both of these expressions are structure expressions because at least one operand of each is a structure variable; hence, each expression represents a structure value.

USE OF EXPRESSIONS

Expressions that are single constants or single variables may appear freely throughout a program. However, the syntax of many PL/I statements allows the appearance of operational expressions, so long as evaluation of the expression yields a valid value.

In syntactic descriptions used in this publication, the unqualified term

"expression" refers to an element expression, an array expression, or a structure expression. For cases in which the kind of expression is restricted, the type of restriction is noted; for example, the term "element-expression" in a syntactic description indicates that neither an array expression nor a structure expression is valid.

Note: Although operational expressions can appear in a number of different PL/I statements, their most common occurrences are in assignment statements of the form:

```
A = B + C;
```

The assignment statement has no PL/I keyword. The assignment symbol (=) indicates that the value of the expression on the right (B + C) is to be assigned to the variable on the left (A). For purposes of illustration in this chapter, some examples of expressions are shown in assignment statements.

DATA CONVERSION IN 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, when the operation is performed, usually must be of the same data type, as specified by the attributes of a variable or the notation used in writing a constant.

The operands of an operation in a PL/I expression are automatically converted, if necessary, to a common representation before the operation is performed. General rules for conversion of different data types are discussed in the following paragraphs and in a later section of this chapter, "Concepts of Data Conversion." Detailed rules for specific cases, including rules for computing precision or length of converted items, can be found in Part II, Section F, "Data Conversion."

Data conversion is confined to conversion of problem data. Program control data, such as statement labels and pointers, is never converted from one type to another.

Bit-String to Character-String

The bit 1 becomes the character 1; and the bit 0 becomes the character 0.

Character-String to Bit-String

The character string should contain the characters 1 and 0 only, in which case the character 1 becomes the bit 1, and the character 0 becomes the bit 0. The CONVERSION condition is raised by an attempt to convert any character other than 1 or 0 to a bit.

Character-String to Arithmetic

Character-string data cannot be converted to coded arithmetic or numeric character type. Any attempt to do so is an error.

Arithmetic to Character-String

Coded arithmetic data cannot be converted to character string type. Any attempt to do so is an error. However, numeric character data can be converted to character string. The numeric character field is interpreted as a character string having the same characters. The length of the string is the same as the length specified in the PICTURE attribute for the numeric character field.

Bit-String to Coded Arithmetic

A bit string is interpreted as an unsigned binary integer and is converted to fixed-point binary of positive value. The base and scale are further converted, if necessary.

Bit String to Numeric Character

The bit string is first converted to coded arithmetic and then to numeric character.

Coded Arithmetic to Bit-String

The absolute value is converted, if necessary, to a fixed-point binary integer. Ignoring the plus sign, the integer is then interpreted as a bit string. The length of the bit string is dependent upon the precision of the original unconverted arithmetic data item.

Numeric Character to Bit String

The numeric character value is converted to coded arithmetic and then to bit string as above.

Numeric Character to Character-String

See "Arithmetic to Character-String" above.

Arithmetic Base and Scale Conversion

The precision of the result of an arithmetic base or scale conversion is dependent upon the precision of the original arithmetic data item. The rules are listed in Part II, Section F, "Data Conversion."

Conversion by Assignment

In addition to conversion performed as the result of an operation in the evaluation of an expression, conversion will also occur when a data item -- or the result of an expression evaluation -- is assigned to a variable whose attributes differ from the attributes of the item assigned. The rules for such conversion are generally the same as those discussed above and in Part II, Section F, "Data Conversion."

EXPRESSION OPERATIONS

An operational expression can specify one or more single operations. The class of operation is dependent upon the class of operator specified for the operation. There are four classes of operations -- arithmetic, bit-string, comparison, and concatenation.

ARITHMETIC OPERATIONS

An arithmetic operation is one that is specified by combining operands with one of the following operators:

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

The plus sign and the minus sign can appear either as prefix operators (associated with and preceding a single operand, such as +A or -A) or as infix operators (associated with and between two operands, such as A + B or A - B). All other arithmetic operators can appear only as infix operators.

An expression of greater complexity can be composed of a set of such arithmetic operations. Note that 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 preceding the variable B indicates that the value of A is to be multiplied by the negative value of B.

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

-A The single minus sign has the effect of reversing the sign of the value that A represents.

--A One minus sign reverses the sign of the value that A represents. The second minus sign again reverses the sign of the value, restoring it to the original arithmetic value represented by A.

---A Three minus signs reverse the sign of the value three times, giving the same result as a single minus sign.

Data Conversion in Arithmetic Operations

The two operands of an arithmetic operation may differ in type, base, precision, and scale. When they differ, conversion takes place according to rules listed below. Certain other rules -- as stated

below -- may apply in cases of exponentiation.

TYPE: Numeric character field operands (digits recorded in character form) and bit-string operands are converted to internal coded arithmetic type. The result of an arithmetic operation is always in coded arithmetic form. Note that 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.

PRECISION: If only precisions differ, no type conversion is necessary.

SCALE: If the scales of the two operands differ, the fixed-point operand is converted to floating-point scale. The exception to this rule 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 scale factor of zero, that is, a fixed-point integer constant or a variable with precision (p,0). In such a case, no conversion is necessary, but the result will be floating-point.

If both operands of an exponentiation operation are fixed-point, conversions may occur, as follows:

1. Both operands are converted to floating-point if the exponent has a precision other than (p,0).

2. The first operand is converted to floating-point unless the exponent is an unsigned fixed-point integer constant.

3. 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 for the implementation (for System/360, 15 decimal digits or 31 binary digits). Further details and examples of conversion in exponentiation are included in the section "Concepts of Data Conversion" in this chapter.

Results of Arithmetic Operations

The "result" of an arithmetic operation, as used in the following text, may refer to an intermediate result if the operation is only one of several operations specified in a single operational expression. Any result may require further conversion if it is an intermediate result that is used as an operand of a subsequent operation or if it is assigned to a variable.

After required conversions have taken place, the arithmetic operation is performed. If maximum precision is exceeded and truncation is necessary, the truncation is performed on low-order fractional digits, regardless of base or scale of the operands. In some cases involving fixed-point data, however, high-order digits may sometimes be lost when scale factors are such that point alignment does not allow for the declared number of digits.

The base, scale, and precision of the result depend upon the operands and the operator involved.

For prefix operations, the result has the same base, scale, and precision as the converted operand. Note that the result of -A, where A is a bit string, is an arithmetic result, since A must first be converted to coded arithmetic form before the operation can be performed.

For infix operations, the result depends upon the scale of the operands in the following ways:

FLOATING POINT: If the converted operands of an infix operation are of floating-point scale, the result is of floating-point scale, and the base of the result is the common base of the operands. The precision of the result is the greater of the precisions of the two operands.

FIXED POINT: If the converted operands of an infix operation are of fixed-point scale, the result is of fixed-point scale, and the base of the result is the common base of the operands. The precision of a fixed-point result depends upon operands, according to the rules listed below.

In the formulas for computing precision, the symbols used are as follows:

- p represents the total number of digits of the result
- q represents the scale factor of the result
- p_1 represents the total number of digits of the first operand
- q_1 represents the scale factor of the first operand
- p_2 represents the total number of digits of the second operand

q_2 represents the scale factor of the second operand

ADDITION AND SUBTRACTION: The total number of digits in the result is equal to 1 plus the number of integer digits of the operand having the greater number of integer digits plus the number of fractional digits of the operand having the greater number of fractional digits. The total number of positions cannot exceed the maximum number of digits allowed (15 decimal digits, 31 binary digits). The scale factor of the result is equal to the larger scale factor of the two operands.

Formulas:

$$p = 1 + \text{maximum}\ (p_1 - q_1,\ p_2 - q_2) + \text{maximum}\ (q_1,\ q_2)$$

$$q = \text{maximum}\ (q_1,\ q_2)$$

Example:

```
12354.2385 + 222.11111
  A    B       C   D
```

The total number of digits in the result would be equal to 1 plus the number of digits in A plus the number of digits in D. The scale factor of the result would be equal to the number of digits in D. Precision of the result would be (11,5).

MULTIPLICATION: The total number of digits in the result is equal to one plus the number of digits in operand one plus the number of digits in operand two. The total number of digits cannot exceed the maximum number of digits allowed for the implementation (15 decimal, 31 binary). The scale factor of the result is the sum of the scale factors of the two operands.

Formulas:

$$p = p_1 + p_2 + 1$$

$$q = q_1 + q_2$$

Example:

```
345.432 * 22.45
 A   B     C  D
```

The total number of digits in the result would be equal to 1 plus the sum of the number of digits in A, B, C, and D. The scale factor of the result would be the sum of the number of digits in B and D. Precision of the result would be (11,5).

DIVISION: The total number of digits in the quotient is equal to the maximum allowed by the implementation (15 decimal, 31 binary). The scale factor of the quotient is dependent upon the number of integer digits of the dividend (A in the example below), and the number of fractional digits of the divisor (D in the example below). The scale factor is equal to the total number of digits of the result minus the sum of A and D.

Formulas:

$$p = 15 \text{ decimal, } 31 \text{ binary}$$

$$q = 15 \text{ (or } 31) - ((p_1 - q_1) + q_2)$$

Example:

```
432.432 / 2
 A   B   C D
```

The total number of digits in the quotient would be 15 (the maximum number allowed). The scale factor would be 15 minus the sum of 3 (A, the number of integer digits in the dividend) and zero (D, the number of fractional digits in the divisor). Precision of the quotient would be (15,12).

Note that any change in the number of integer digits in the dividend or any change in the number of fractional digits in the divisor will change the precision of the quotient, even if all additional digits are zeros. Also note from the above formulas that the result of a fixed-point division can have a scale factor greater than zero even though the operands might have a scale factor of zero (or no scale factor, in the case where the operands are fixed-point binary variables).

Examples:

```
00432.432 / 2

432.432 / 2.0000
```

Precision of the quotient of the first example would be (15,10); scale factor is equal to 15-(5+0). Precision of the quotient of the second example would be (15,8); scale factor is equal to 15-(3+4).

Caution: In the use of fixed-point division operations, care should be taken that declared precision of variables and apparent precision of constants will not give a result with a scale factor that can force the result of subsequent operations to exceed the maximum number of digits allowed by the implementation.

EXPONENTIATION: If the second operand (the exponent) is an unsigned nonzero fixed-point constant of precision (p,0), the total number of positions in the result is equal to one less than the product of a number that is one greater than the number of digits in the first operand multiplied by the value of the second operand (the

exponent). The scale factor of the result is equal to the product of the scale factor of the first operand multiplied by the value of the second operand (the exponent).

Note: In the exponentiation operation x**y, some special cases are defined as follows:

1. If x=0 and y>0, the result is 0.
2. If x=0 and y≤0, the ERROR condition is raised.
3. If x≠0 and y=0, the result is 1.
4. If x<0 and y is not fixed-point with precision (p,0), the ERROR condition is raised.

(As pointed out under "Data Conversion in Arithmetic Operations," if the exponent is not an unsigned fixed-point integer constant, or if the total number of digits of the result would exceed 15 decimal digits or 31 binary digits, the first operand is converted to floating-point scale, and the rules for floating-point exponentiation apply.)

Formulas:

$$p = ((p_1 + 1)*(\text{value-of-exponent}))- 1$$

$$q = q_1 *(\text{value-of-exponent})$$

Example:

```
32 ** 5
```

The total number of digits in the result would be 14. This is arrived at by multiplying a number equal to one plus the number of digits in the first operand (1+2) by the value of the exponent and subtracting one. The scale factor of the result would be zero (0 * 5, scale factor of the first operand multiplied by the value of the exponent).

BIT-STRING OPERATIONS

A bit-string operation is one that is specified by combining operands with one of the following operators:

¬ & |

The first operator, the "not" symbol, can be used as a prefix operator only. The second and third operators, 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-string operation are, if necessary, converted to bit strings before the operation is performed. If the operands of an infix operation are of unequal length, the shorter is extended on the right with zeros.

The result of a bit-string operation is a bit string equal in length to the length of the operands (the two operands, after conversion, always are the same length).

Bit-string operations are performed on a bit-by-bit basis. The effect of the "not" operator is bit reversal; that is, the result of ¬1 is 0; the result of ¬0 is 1. The result of an "and" operation is 1 only if both corresponding bits are 1; in all other cases, the result is 0. The result of an "or" operation is 1 if either or both of the corresponding bits are 1; in all other cases, the result is 0. The following table illustrates the result for each bit position for each of the operators:

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

More than one bit-string operation can be combined in a single expression that yields a bit-string value.

In the following examples, if the value of operand A is '010111'B, the value of operand B is '111111'B, and the value of operand C is '110'B, then

¬ A	yields	'101000'B
¬ C	yields	'001'B
C & B	yields	'110000'B
A | B	yields	'111111'B
C | B	yields	'111111'B
A | (¬C)	yields	'011111'B
¬((¬C)|(¬B))	yields	'110111'B

COMPARISON OPERATIONS

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

```
<  ¬<  <=  =  ¬=  >=  >  ¬>
```

These operators specify "less than," "not less than," "less than or equal to," "equal to," "not equal to," "greater than or equal to," "greater than," and "not greater than."

There are four types of comparisons:

1. Algebraic, which involves the comparison of signed arithmetic values in internal coded arithmetic form. If operands differ in base, scale, or precision, they are converted according to the rules for arithmetic operations. Numeric character data is converted to coded arithmetic before comparison.

2. Character, which involves left-to-right, character-by-character comparisons of characters according to the collating sequence.

3. Bit, which involves left-to-right, bit-by-bit comparison of binary digits.

4. Pointer, for which only the operators = and ¬= are allowed. Both operands must be valid pointer expressions, since there is no type conversion of program control data.

If the operands of a comparison (other than pointer) are of different types, the operand of the lower type is converted to the type of the operand of the higher type. The priority of types is (1) internal coded arithmetic (highest), (2) character string, (3) bit string. (Character strings cannot be compared with arithmetic data.)

If operands of a character-string comparison, after conversion, are of different lengths, the shorter operand is extended on the right with blanks. If operands of a bit-string comparison are of different lengths, the shorter is extended on the right with zeros.

The result of a comparison operation always is a bit string of length one; the value is '1'B if the relationship is true, or '0'B if the relationship is not true.

The most common occurrences of comparison operations are in the IF statement, of the following format:

```
IF A = B
    THEN action-if-true
    ELSE action-if-false
```

The evaluation of the expression A = B yields either '1'B or '0'B. Depending upon the value, either the THEN portion or the ELSE portion of the IF statement is executed.

Comparison operations need not be limited to IF statements, however. The following assignment statement could be valid:

```
X = A < B;
```

In this example, the value '1'B would be assigned to X if A is less than B; otherwise, the value '0'B would be assigned. In the same way, the following assignment statement could be valid:

```
X = A = B;
```

The first symbol (=) is the assignment symbol; the second (=) is the comparison operator. If A is equal to B, the value of X will be '1'B; if A is not equal to B, the value of X will be '0'B.

Only the comparison operations of "equal" and "not equal" are valid for comparisons of pointer variable operands. Comparison operations with labels is not allowed.

CONCATENATION OPERATIONS

A concatenation operation is one that is specified by combining operands with the concatenation symbol:

```
||
```

It signifies that the operands are to be joined in such a way that the last character or bit of the operand to the left will immediately precede the first character or bit of the operand to the right, with no intervening bits or characters.

The concatenation operator can cause conversion to string type since concatenation can be performed only upon strings, either character strings or bit strings. If both operands are character strings or if both operands are bit strings, no conversion takes place. Otherwise both operands are converted to character strings.

The results of concatenation operations are as follows:

Bit string: a bit string whose length is equal to the sum of the lengths of the two bit-string operands.

Character string: a character string whose length is equal to the sum of the lengths of the two character-string operands.

For example, if A has the attributes and value of the constant '010111'B, B of the constant '101'B, C of the constant 'XY,Z', and D of the constant 'AA/BB', then

```
A||B      yields '010111101'B

A||A||B   yields '010111010111101'B

C||D      yields 'XY,ZAA/BB'

D||C      yields 'AA/BBXY,Z'

B||D      yields '101AA/BB'
```

Note that, 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 consisting of eight characters.

COMBINATIONS OF OPERATIONS

Different types of operations can be combined within the same operational expression. Any combination can be used. For example, the expression shown in the following assignment statement is valid:

```
RESULT = A + B < C & D || E;
```

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.

Assume that the variables given above are declared as follows:

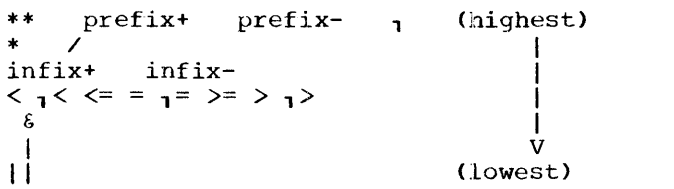
```
DECLARE RESULT CHARACTER(7),
        A FIXED DECIMAL(1),
        B FIXED BINARY (3),
        C BIT(2),
        D BIT(4),
        E CHARACTER(3);
```

- The decimal value of A would be converted to binary base.
- The binary addition would be performed, adding A and B.
- The binary result would be compared with the converted binary value of C.
- The bit-string result of the comparison would be extended to the length of the bit string D, and the "and" operation would be performed.
- The result of the "and" operation, a bit string of length 4, would be converted to a character string and concatenated with the character-string E, giving a length of 7.
- The character-string result would be assigned to RESULT without conversion.

Note: The order of evaluation of an expression depends upon the priority of the operators appearing in the expression. In the above example, the priority of operation is such that evaluation proceeds from left to right.

Priority of Operators

In the evaluation of expressions, priority of the operators is as follows:



If two or more operators of the highest priority appear in the same expression, the order of priority of those operators is from right to left; that is, the rightmost exponentiation or prefix operator has the highest priority. Each succeeding exponentiation or prefix operator to the left has the next highest priority.

For all other operators, if two or more operators of the same priority appear in the same expression, the order of priority of those operators is from left to right.

Note that the order of evaluation of the expression in the assignment statement:

```
RESULT = A + B < C & D || E;
```

is the result of the priority of the operators. It is as if various elements of the expression were enclosed in parentheses as follows:

```
(A) + (B)

(A + B) <(C)

(A + B <C) & (D)

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

The order of evaluation of an expression (and, consequently, the result) can be changed through the use of parentheses. The above expression, for example, might be written as follows:

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

The order of evaluation of this expression would yield a bit string, the result of the "and" operation.

In such an expression, those expressions enclosed in parentheses are evaluated first, to be reduced to a single value, before they are considered in relation to surrounding operators. Within the language, however, no rules specify which of two parenthesized expressions, such as those in the above example, would be evaluated first.

The value of C would be converted to fixed-point binary, and the comparison would be made, yielding a bit string of length one (RESULT_1). The value of D would be converted to a character string and concatenated with E (RESULT_2).

At this point, the expression would have been reduced to:

```
A + RESULT_1 & RESULT_2
```

Since the infix + has a higher priority than the & operator, the addition would be performed first, yielding RESULT_3, and the expression would be:

```
RESULT_3 & RESULT_2
```

The two operands would be converted to bit strings, and the "and" operation would be performed, yielding the bit-string result of the entire expression.

The priority of operators is defined only within operands (or sub-operands). It does not necessarily hold true for an entire expression. Consider the following example (assuming that A, B, C, etc. have been redefined):

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

The priority of the operators specifies, in this case, only that the exponentiation will occur before the "or" operation. It does not specify the order of the operation 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. For instance, in the first example under "Combinations of Operations," the concatenation operator is the operator of the final infix operation. In the second example (because of the use of parentheses), the operator of the final infix operation is the "and" operator, and the evaluation would yield a different value.

In general, unless parentheses are used within the expression, the operator of lowest priority determines the operands of the final operation. For example:

```
A + B ** 3 & C * D - E
```

In this case, the "and" operator indicates that the final operation will be:

```
(A + B ** 3) & (C * D - E)
```

Subexpressions can be analyzed in the same way. The two operands of the expression can be defined as follows:

```
A + (B ** 3)

(C * D) - E
```

ARRAY EXPRESSIONS

An array expression is a single array variable or an expression that includes at least one array operand. Array expressions may also include operators -- both prefix and infix -- element variables, and constants.

Evaluation of an array expression yields an array result. All operations performed on arrays are performed on an element-by-element basis in row-major order (that is with the rightmost subscript varying most rapidly). Therefore all arrays referred to in an array expression must be of identical bounds.

Note: Array expressions other than addition and subtraction are not expressions of conventional matrix algebra.

PREFIX OPERATORS AND ARRAYS

The result of the operation of a prefix operator on an array is an array of identical bounds, each element of which is the result of the operation having been performed upon 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 may have an element or another array as the other operand.

Array and Element Operations

The result of an operation in which an element and an array are connected by an infix operator is an array with bounds identical to the original array, each element of which is the result of the operation performed upon the 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

The element of an array-element operation can be an element of the same array. For example, the expression A*A(2,3) would give the same result in the case of the array A above, since the value of A(2,3) is 3.

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

Note that the original value for A(1,2), which is 10, is used in the evaluation for only the first two elements of A. Since the result of the expression is assigned to A, changing the value of A, the new value of A(1,2) is used for all subsequent operations. The first two elements are multiplied by 10, the original value of A(1,2); all other elements are multiplied by 100, the new value of A(1,2).

Array and Array Operations

If two arrays are connected by an infix operator, the two arrays must have the same number of dimensions and identical 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.

Examples of array infix expressions are:

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

Data Conversion in Array Expressions

The examples in this discussion of array expressions have shown only single arithmetic operations. The rules for combining operations and for data conversion of operands are the same as those for element operations.

STRUCTURE EXPRESSIONS

A structure expression is a single structure variable or an expression that includes at least one structure operand and does not contain an array operand. Element variables and constants can be operands of a structure expression. Evaluation of a structure expression yields a structure result. A structure operand can be a major structure name or a minor structure name.

All operations performed on structures are performed on an element-by-element basis. All structure operands appearing in a structure expression must have identical structuring.

Identical structuring means that the structures must have the same minor structuring and the same number of contained elements and arrays and that 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. Names do not have to be the same. Data types of corresponding elements do not have to be the same, so long as valid conversion can be performed.

PREFIX OPERATORS AND STRUCTURES

The result of the operation of a prefix operator on a structure is a structure of identical structuring, each element of which is the result of the operation having been performed upon each element of the original structure.

Note: Since structures may contain elements of many different data types, a prefix operation in a structure expression would be meaningless unless the operation can be validly performed upon every element represented by the structure variable, which is either a major structure name or a minor structure name.

INFIX OPERATORS AND STRUCTURES

Infix operations that include a structure variable as one operand may have an element or another structure as the other operand.

Structure and Element Operations

When an operation has one structure and one element operand, it is the same as a series of operations, one for each element in the structure. Each sub-operation involves a structure element and the single element.

Consider the following structure:

```
1 A
  2 B
    3 C
    3 D
    3 E
  2 F
    3 G
    3 H
    3 I
```

If X is an element variable, then A * X is equivalent to:

```
A.C * X
A.D * X
A.E * X
A.G * X
A.H * X
A.I * X
```

Structure operands in a structure expression need not be major structure names. A minor structure name, at any level, is a structure variable. Thus, the following are structure expressions:

```
A.B & '1010'B

F * 32
```

Structure and Structure Operations

When an operation has two structure operands, it is the same as a series of element operations, one for each corresponding pair of elements. For example, if A is the structure shown in the previous example and if M is the following structure:

```
1 M
  2 N
    3 O
    3 P
    3 Q
  2 R
    3 S
    3 T
    3 U
```

then A || M is equivalent to:

```
A.C || M.O
A.D || M.P
A.E || M.Q
A.G || M.S
A.H || M.T
A.I || M.U
```

As stated above, structure operands in a structure expression need not be major structure names. A minor structure name, at any level, is a structure variable. Thus, the following is a structure expression:

```
M.N & M.R
```

OPERANDS OF EXPRESSIONS

An operand of an expression can be a constant, an element variable, an array variable, or a structure variable. An operand can also be an expression that represents a value that is the result of a computation, as shown in the following assignment statement:

```
A = B * SQRT(C);
```

In this example, the expression SQRT(C) represents a value that is equal to the square root of the value of C. Such an expression is called a function reference.

FUNCTION REFERENCE OPERANDS

A function reference consists of a name and, usually, a parenthesized list of one or more variables, constants, or other expressions. The name is the name of a block of coding written to perform specific computations upon the data represented by the list and to substitute the computed value in place of the function reference.

Assume, in the above example, that C has the value 16. The function reference SQRT(C) causes execution of the coding that would compute the square root of 16 and replace the function reference with the value 4. In effect, the assignment statement would become:

```
A = B * 4;
```

The coding represented by the name in the function reference is called a function. The function SQRT is one of the PL/I built-in functions. Built-in functions, which provide a number of different operations, are a part of the PL/I language. A complete discussion of each appears in Part II, Section G, "Built-In Functions and Pseudo-Variables." In addition, a programmer may write functions for other purposes (as described in Chapter 10, "Subroutines and Functions"), and the names of those functions can be used in function references.

The use of a function reference is not limited to operands of operational expressions. A function reference is, in itself, an expression and can be used wherever an expression is allowed. It cannot be used in those cases where a variable represents a receiving field, such as to the left of an assignment statement.

There are, however, two built-in functions that can be used as pseudo-variables. A pseudo-variable is a built-in function name that is used in a receiving field. Consider the following 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 name is used both in a normal function reference and as a pseudo-variable.

The SUBSTR built-in function extracts a substring of specified length from the named string. As a pseudo-variable, it indicates the location, within a named string, that is the receiving field.

In the above example, a substring five characters in length, beginning with the 20th character of the string B, is to be assigned to the last five characters of the string A. That is, the last five characters of A are to be replaced by the 20th through the 24th characters of B. The first five characters of A remain unchanged, as do all of the characters of B.

The two built-in functions that can be used as pseudo-variables (SUBSTR and UNSPEC) are discussed in Part II, Section G, "Built-In Functions and Pseudo-

Variables." No programmer-written function can be used as a pseudo-variable.

CONCEPTS OF DATA CONVERSION

Data conversion is the transformation of the representation of a value from one form to another. Although there are some restrictions upon the use of the available forms of data representation and upon the mixing of different representations within an expression, the programmer still has a great deal of freedom in this area.

Programmers who wish to make use of this freedom must understand that mixed expressions imply conversions. If conversions take place at execution time, they will slow down the execution, sometimes significantly. Unless care is taken, conversions can result in loss of precision and can cause unexpected results.

This section is concerned primarily with the concepts of conversion operations. Specific rules for each kind of conversion are listed in Part II, Section F, "Data Conversion." Earlier sections of this chapter discuss circumstances under which conversion can occur during evaluation of expressions. This section deals with the processes of the conversion.

The subject of conversion can be considered in two parts, first, determining the target attributes, and, second, the conversion operation with known source and target attributes. This section deals with determining target attributes. Rules for conversion operations are given in Part II, Section F, "Data Conversion." Within each section, here and in Part II, arithmetic conversion and type conversion are considered separately.

The target of a conversion is the receiving field to which the converted value is assigned. In the case of a direct assignment, such as A = B, in which conversion must take place, the variable to the left of the assignment symbol (in this case, A) is the target. Consider the following example, however:

```
DECLARE   A PICTURE '$9999V.99',
          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 targets, as follows:

1. The compiler-created temporary to which the converted binary equivalent of B is assigned
2. The compiler-created temporary to which the binary result of the addition is assigned
3. The temporary 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 numeric character equivalent of the decimal fixed-point representation of the value is assigned

The attributes of the first target 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 target are determined from the attributes of the source (C and the converted representation of B). The attributes of the third target are determined in part from the source (the second target) and in part from the attributes of the eventual target (A). (The only attribute determined from the eventual target is DECIMAL, since a binary arithmetic representation must be converted to decimal representation before it can be converted to a numeric character.) The attributes of the fourth target (A) are known from the DECLARE statement.

Thus, 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 assumptions may be made, and some implementation restrictions (for example, maximum precision) and conventions exist. After an expression is evaluated, the result may be further converted. In this case, the target attributes usually are independent of the source. Since the process of determining target attributes is different for expression operands and for the results of expression evaluation, the two cases are dealt with separately.

A conversion always involves a source data item and a target data item, that is, the original representation of the value and the converted representation of the value. All of the attributes of both the source data item and the target data item are known, or assumed, at compile time.

It should be realized that constants also have attributes; the constant 1.0 is different from the constants 1, '1'B, '1',

1B, or 1E0. Constants may be converted at compile time or at execution time, but in either case, the rules are the same.

Table 4-1. Target Types for Expression Operands

Operator	Target Type
+ - * / **	coded arithmetic
& | ¬	bit string
||	character string (unless both operands are bit strings)
> < >= <= = ¬= ¬> ¬<	arithmetic, unless both operands are strings then character string unless both operands are bit strings then bit string (Pointers can be compared only by using = and ¬=; both operands must be pointers since no conversion can be performed.)

TARGET ATTRIBUTES FOR TYPE CONVERSION

When an expression operand requires type conversion, some target attributes must be assumed or deduced from the source. Some of these assumptions can be based on the operator, as shown in Table 4-1. Note that numeric character data can always be converted to coded arithmetic and vice versa.

BIT-TO-CHARACTER AND CHARACTER-TO-BIT

In the conversion of bit to character, and character to bit, the length of the target (in bits or characters) is the same as the length of the source (in bits or characters).

CODED ARITHMETIC TO BIT-STRING

In the conversion of coded arithmetic to bit-string data, length of the target is deduced from the precision of the source. Algorithms for determining the length of the target are given below under the heading "Lengths of Bit-String Targets."

BIT-STRING TO CODED ARITHMETIC

The attributes of the target are the attributes that would have been given to the target if a fixed-point binary integer of maximum precision (31) had appeared in place of the bit string.

When converting to fixed-point in System/360 implementations, this operation is performed by first converting the string to a maximum precision integer (BINARY (31)). This integer is then converted to the target attributes.

Target Attributes for Arithmetic Expression Operands

Except for exponentiation, the target attributes for arithmetic conversion are assumed as follows:

BINARY — unless both operands are DECIMAL, in which case no base conversion is performed

FLOAT — unless both operands are FIXED, in which case no scale conversion is performed

precision of source — unless base or scale conversion is performed (see Table 4-2, "Precision for Arithmetic Conversion")

In the case of exponentiation, the base and precision are determined as for other operations. The target scale of the first operand is always FLOAT unless the first operand source is FIXED and the second operand (the exponent) is an unsigned fixed-point integer constant with a value small enough that the result of the exponentiation will not exceed the maximum number of digits allowed (for System/360 implementations, 31, if binary, or 15, if decimal). The target scale of the second operand is FLOAT unless it is an integer constant or a fixed-point variable of precision (p,0).

In the examples of exponentiation shown below, the variables are those named in the following DECLARE statement:

```
DECLARE A FIXED DECIMAL(2),
        B FIXED DECIMAL(3,2),
        C FLOAT DECIMAL(4),
        D FLOAT DECIMAL(7),
        E FIXED DECIMAL(8),
        F FIXED DECIMAL(15);
```

Note: If only one digit appears in the precision attribute specification for a fixed-point variable, the scale factor is, by default, zero; the precision is (p,0).

D ** C — No conversion necessary. Both operands are floating-point.

A ** 4 — No conversion necessary. Second operand is unsigned fixed-point integer constant, and the result will not exceed 15 digits.

D ** 5 — No conversion necessary. First operand is floating-point; second is fixed-point with precision (p,0).

D ** A — No conversion necessary. First operand is floating-point; second is fixed-point with precision (p,0).

E ** A — First operand is converted to floating-point because second operand is not unsigned fixed-point integer constant. Second operand is not converted because it has precision (p,0).

D ** B — Second operand is converted to floating-point because it does not have precision (p,0). Even if B had an integer value with a fractional part of zero, it still would be converted, since its declared precision is (3,2).

Note: All of these examples, except D**B, would be the same if they had been declared binary rather than decimal, except that the maximum number of binary digits allowed is 31. In the case of D**B, B, being binary, could not be declared with a scale factor; hence, if B has a precision of (3), no conversion is necessary.

Precision and Length of Expression Operand Targets

The following rules apply to all calculations of precision and length:

1. Precision and length specifications are always integers. If any of the calculations given below produces a nonintegral value, the next largest integer is taken as the resulting precision.

   The following illustrates how precision would be computed in a conversion from DECIMAL FIXED (8,3) to BINARY FIXED:

   1 + 8 * 3.32 = 27.56 resulting number of digits (p) is 28.

   3 * 3.32 = 9.96 resulting scale factor (q) is 10.

   Note that a scale factor is maintained in conversions to fixed-point binary. However, if the converted result were assigned to a fixed-point binary variable, the fractional binary digits would be truncated since a fixed-point binary variable can have no scale factor declared for it (and hence has an assumed scale factor of zero). Also note that the scale factor can sometimes be negative (e.g., the BINARY and DECIMAL built-in functions). In such cases, the absolute (positive) value is used to take the next largest integer.

2. There is an implementation-defined maximum for the precision of each arithmetic representation. If any calculation yields a value greater than the implementation-defined limit, then the implementation limit is used instead. In System/360 implementations these limits are:

   FIXED DECIMAL -- 15 digits

   FIXED BINARY -- 31 digits

   FLOAT DECIMAL -- 16 digits

   FLOAT BINARY -- 53 digits

   Because of the particular values for these implementations, these limits will usually come into effect only for conversions from fixed-point decimal to fixed-point binary.

   For the D-Compiler, the scale factor for fixed-point decimal variables must lie within 0 and 15, inclusive. The scale factor for binary fixed-point variables cannot be specified and is always assumed to be zero.

Precision for Arithmetic Conversions

Table 4-2 gives the target precision for an operand if base or scale conversion occurs.

The target precision of one operand of an expression is not affected by the precision of the other operand. This can have a

Table 4-2. Precision for Arithmetic Conversions

Source Attributes	Target Attributes	Target Precision
DECIMAL FIXED(p,q)	DECIMAL FLOAT	p
DECIMAL FIXED(p,q)	BINARY FIXED	1+p*3.32,q*3.32
DECIMAL FIXED(p,q)	BINARY FLOAT	p*3.32
DECIMAL FLOAT(p)	BINARY FLOAT	p*3.32
BINARY FIXED(p,q)	BINARY FLOAT	p
BINARY FIXED(p,q)	DECIMAL FLOAT	p/3.32
BINARY FIXED(p,q)	DECIMAL FIXED	1+p/3.32,q/3.32
BINARY FLOAT(p)	DECIMAL FLOAT	p/3.32

significant effect on accuracy, particularly if one of the operands is a constant.

Lengths of Character-String Targets

If the source is a numeric character data item or a bit string and the target is a character string, the length of the target is the same as the length of the source.

Lengths of Bit-String Targets

When converting arithmetic operands to bit string, the arithmetic source is converted to a positive binary integer. The precision of the binary integer target is the same as the length of the bit-string target as given in Table 4-3.

Note that p-q represents the number of binary or decimal digits to the left of the point. For the D-compiler, the target length must lie within 1 and 31, inclusive.

Table 4-3. Lengths of Bit-String Targets

Source Attributes	Target Length
DECIMAL FIXED(p,q)	(p-q)*3.32
DECIMAL FLOAT(p)	p*3.32
BINARY FIXED(p,q)	p-q
BINARY FLOAT(p)	p

Conversion of the Value of an Expression

The result of a completely evaluated expression may require further conversion. The circumstances in which this can occur, and the target attributes for each situation, are given in Figure 4-4. In addition, certain built-in functions cause conversion. Any subscript reference is converted to binary integer.

CONVERSION OPERATIONS

As in the case of determining target attributes, conversion operations may also be considered in two stages: type conversion and arithmetic conversion. For example, when a numeric character source is converted to a coded arithmetic target, the string is first converted to an arithmetic form whose attributes are determined by the constant expressed by the PICTURE specification. This intermediate result is then converted (if necessary) to the attributes of the target. These two stages may not be separated in an actual implementation, but for the purpose of description it is convenient to consider them separately.

There are nine cases of type conversion:

- Numeric character to character-string
- Numeric character to coded arithmetic
- Coded arithmetic to numeric character
- Coded arithmetic to bit-string
- Bit-string to coded arithmetic

Table 4-4. Circumstances that Can Cause Conversion

The following may cause conversion to any target attributes:		
Cause		Target Attributes
Assignment		Attributes of variable to the left of the assignment symbol
RETURN (expression)		Attributes specified in PROCEDURE or ENTRY statement
The following may cause conversion to character string:		
Statement	Option	String Length
DISPLAY		Source, 80-character maximum
RECORD I/O	KEYFROM	Key length specified in ENVIRONMENT attribute
	KEY	Key length specified in ENVIRONMENT attribute (or eight characters in the case of REGIONAL(1))
The following may cause conversion to a binary integer whose precision, as defined for the D-Compiler, is given below:		
Statement	Option/Attribute	Precision
OPEN	PAGESIZE	8
I/O	SKIP	8
	LINE	8

- Character-string to bit-string
- Bit-string to character-string
- Numeric character to bit-string
- Bit-string to numeric character

For specific rules for each of the cases of type conversion and for arithmetic conversion, see Part II, Section F, "Data Conversion."

THE CONVERSION, SIZE, OVERFLOW, AND FIXEDOVERFLOW CONDITIONS

When data is converted from one representation to another, the CONVERSION or SIZE conditions may be raised. The OVERFLOW and FIXEDOVERFLOW conditions are raised only when the result of an arithmetic operation exceeds the implementation-defined limit. When an operand is converted from one representation to another, if the value of the result will not fit in the declared precision for the new representation, the SIZE condition is raised.

The SIZE condition is raised when significant digits are lost from the left-hand side of an arithmetic value. This can occur during conversion within an expression, or upon assigning the result of an expression. It is not raised in conversion to character string or bit string even if the value is truncated. It is raised on conversion to E or F format in edit-directed transmission if the field width specified will not hold the value of the list item. The SIZE condition is normally disabled, so an interrupt will occur only if the condition is raised within the scope of a SIZE prefix.

The CONVERSION condition is raised when the source field contains a character that is invalid for the conversion being performed. For example, CONVERSION would be raised if a character string that is being converted to bit contains any character other than 0 and 1. The CONVERSION condition is normally enabled, so when the condition is raised, an interrupt will occur. It can be disabled by a NOCONVERSION prefix, in which case an interrupt will not occur when the condition is raised.

Note that the OVERFLOW and FIXEDOVERFLOW conditions are raised when an implementation maximum is exceeded, while the SIZE condition is raised when a declared precision is exceeded. Note also that the OVERFLOW condition can be raised for a conversion only when the scale factor specified in an F-format item is too large.

CHAPTER 5: STATEMENT CLASSIFICATION

This chapter classifies statements according to their functions. Statements in each functional class are listed, the purpose of each statement is described, and examples of their use are shown.

A detailed description of each statement is not included in this chapter but may be found in Part II, Section J, "Statements."

CLASSES OF STATEMENTS

Statements can be grouped into the following six classes:

Descriptive

Input/Output

Data Movement and Computational

Control

Exception Control

Program Structure

The names of the classes have been chosen for descriptive purposes only; they have no fundamental significance in the language. Some statements are included in more than one class, since they can have more than one function.

DESCRIPTIVE STATEMENTS

When a PL/I program is executed, it may manipulate many different kinds of data. Each data item, except a constant, is referred to in the program by a name. The PL/I language requires that the properties (or attributes) of data items referred to must be known at the time the program is compiled. There is an exception to this rule: for certain files, the INPUT or OUTPUT attribute can be specified in an OPEN statement and, therefore, can be determined during the execution of the program.

The DECLARE Statement

The DECLARE statement is the principal means of specifying the attributes of a name. Defaults are applied to any name for which a complete set of attributes has not been specified.

DECLARE statements are always needed for fixed-point decimal and floating-point binary variables, character- and bit-string variables, filenames, pointer variables, label variables, arrays and structures, data with the STATIC or BASED attribute, all data with the PICTURE attribute and, in general, data with the EXTERNAL attribute. A RETURNS attribute declaration must be made for the name of any function that returns a value with attributes different from the default attributes that would be assumed for the name -- FIXED BINARY(15) if the first letter of the name is I through N; otherwise, DECIMAL FLOAT (6). (The default precisions are those defined for System/360 implementations.)

DECLARE statements may also be an important part of the documentation of a program; consequently, programmers may make liberal use of declarations, even when default attributes apply or when a contextual declaration is possible. Because there are no restrictions on the number of DECLARE statements, different DECLARE statements can be used for different groups of names. This can make modification easier and the interpretation of diagnostics clearer.

Other Descriptive Statements

As a rule, file description attributes must be specified in a DECLARE statement. However, the OPEN statement allows the INPUT or OUTPUT attribute, as well as the page size, to be specified for certain files. Therefore, the OPEN statement can be classified as a descriptive statement. The FORMAT statement may be thought of as describing the layout of data on an external medium, such as on a page or on an input card.

INPUT/OUTPUT STATEMENTS

The principal statements of the input/output class are those that actually cause a transfer of data between internal storage and an external medium. Other input/output statements that affect such transfers may be considered input/output control statements.

In the following list, the statements that cause a transfer of data are grouped into two subclasses, RECORD I/O and STREAM I/O:

RECORD I/O Transfer Statements

- READ
- WRITE
- REWRITE
- LOCATE

STREAM I/O Transfer Statements

- GET
- PUT

I/O Control Statements

- OPEN
- CLOSE

An allied statement, discussed with these statements, is the DISPLAY statement.

There are two important differences between STREAM transmission and RECORD transmission. In STREAM transmission, each data item is treated individually, whereas RECORD transmission is concerned with collections of data items (records) as a whole. In STREAM transmission, each item may be edited and converted as it is transmitted; in RECORD transmission, the record on the external medium is an exact copy of the record as it exists in internal storage, with no editing or conversion performed.

As a result of these differences, record transmission is particularly applicable for processing large files that are written in an internal representation, such as in binary or packed decimal. Stream transmission may be used for processing keypunched data and for producing readable output, where editing is required. Since files for which stream transmission is used tend to be smaller, the larger processing overhead can be ignored.

RECORD I/O Transfer Statements

The READ statement transmits records directly into working storage or makes records available for processing. The WRITE statement creates new records, transferring collections of data to the output device. The LOCATE statement also creates new records, but it acts by making buffer space available in which the record may be built. The REWRITE statement alters existing records in an UPDATE file.

STREAM I/O Transfer Statements

STREAM transmission files are sequential files that can be processed only with the GET and PUT statements. Record boundaries generally are ignored; data is considered to be a stream of individual data items, either coming from (GET) or going to (PUT) the external medium.

The GET and PUT statements transmit a list of items in the edit-directed mode. In this mode, the data is recorded externally as a string of characters to be treated character by character according to a format list.

Note: The GET and PUT statements can also be used for internal data movement, by specifying the STRING option and omitting the FILE option. Although the facility may be used in association with READ and WRITE statements for moving data to and from a buffer, it is not actually a part of the input/output operation. GET and PUT statements with the STRING option are discussed in the section "Data Movement and Computational Statements," in this chapter (Chapter 9" Editing and String Handling" also touches upon this area).

Input/Output Control Statements

The OPEN statement associates a file name with a data set and prepares the data set for processing. It may also specify additional attributes for the file.

An OPEN statement need not always be written for a STREAM transmission file. Execution of a GET or PUT statement that specifies the name of an unopened file will result in an automatic opening of the file before the data transmission takes place. However, an OPEN statement can be used to save time by opening such a file before it is first required for use. The page size for a file with the PRINT attribute can be

specified only in an OPEN statement. An OPEN statement must always be specified for a RECORD transmission file.

The CLOSE statement dissociates a data set from a file. All files are closed at the termination of a program, so a CLOSE statement is not always required.

The DISPLAY Statement

The DISPLAY statement is used to write messages on the console, usually to the operator. It may also be used, with the REPLY option, to allow the operator to communicate with the program by typing in a code or a message. The REPLY option may be used merely to suspend execution until the operator acknowledges the message.

DATA MOVEMENT AND COMPUTATIONAL STATEMENTS

Internal data movement involves the assignment of the value of an expression to a specified variable. The expression may be a constant or a variable, or it may be an expression that specifies computations to be made.

The most commonly used statement for internal data movement, as well as for specifying computations, is the assignment statement. The GET and PUT statements with the STRING option also can be used for internal data movement. The PUT statement can, in addition, specify computations to be made.

The Assignment Statement

The assignment statement, which has no keyword, is identified by the assignment symbol (=). It generally takes one of two forms:

```
A = B;

A = B + C;
```

The first form can be used purely for internal data movement. The value of the variable (or constant) to the right of the assignment symbol is to be assigned to the variable to the left. The second form includes an operational expression whose value is to be assigned to the variable to the left of the assignment symbol. The second form specifies computations to be made, as well as data movement.

Since the attributes of the variable on the left may differ from the attributes of the result of the expression (or of the variable or constant), the assignment statement can also be used for conversion and editing.

The variable on the left may be the name of an array or a structure; the expression on the right may yield an array or structure value. Thus, the assignment statement can be used to move aggregates of data, as well as single items.

The STRING Option

If the STRING option appears in a GET or PUT statement in place of a FILE option, execution of the statement will result only in internal data movement; neither input nor output is involved.

Assume that NAME is a string of 30 characters and that FIRST, MIDDLE, and LAST are string variables. Consider the following example:

```
GET STRING (NAME) EDIT
    (FIRST,MIDDLE, LAST)
    (A(12),A(1),A(17));
```

This statement specifies that the first 12 characters of NAME are to be assigned to FIRST, the next character to MIDDLE, and the remaining 17 characters to LAST.

The PUT statement with the STRING option specifies the reverse operation, that is, that the values of the specified variables are to be concatenated into a string and assigned as the value of the string named in the STRING option. For example:

```
PUT STRING (NAME) EDIT
    (FIRST,MIDDLE, LAST)
    (A(12),A(1),A(17));
```

This statement specifies that the values of FIRST, MIDDLE, and LAST are to be concatenated, in that order, and assigned to the string variable NAME.

Computations to be performed can be specified in a PUT statement by including operational expressions in the data list. Assume, for the following example, that the variables A, B, and C represent arithmetic data and BUFFER represents a character string:

```
PUT STRING (BUFFER) EDIT
    (A * 3,B + C)
    (F(15), F(15));
```

This statement specifies that the character string assigned to BUFFER is to consist of the character representations of the value of A multiplied by 3 and the value of the sum of B and C. Note that while arithmetic to character-string and character-string to arithmetic conversions are not allowed in the PL/I subset, they can be effectively achieved by the GET STRING and PUT STRING operations, respectively; however, it should also be noted that this can be quite inefficient because of the high overhead in execution time and storage that is required.

Operational expressions in the data list of a PUT statement are not limited to PUT statements with the STRING option. Operational expressions can appear in PUT statements that specify output to a file. In either case, however, such expressions must be element expressions; they cannot involve arrays or structures.

CONTROL STATEMENTS

Statements in a PL/I program, in general, are executed sequentially unless the flow of control is modified by the occurrence of an interrupt or the execution of one of the following control statements:

```
GO TO

IF

DO

CALL

RETURN

END

STOP
```

The GO TO Statement

The GO TO statement is most frequently used as an unconditional branch. If the destination of the GO TO is specified by a label variable, it may then be used as a switch by assigning label constants, as values, to the label variable.

If the label variable is subscripted, the switch may be controlled by varying the subscript. Usually, however, simple control statements are the most efficient.

The keyword of the GO TO statement may be written either as two words separated by a blank or as a single word, GOTO.

The IF Statement

The IF statement provides the most common conditional branch and is usually used with a simple comparison expression following the word IF. For example:

```
IF A = B

   THEN action-if-true

   ELSE action-if-false
```

If the comparison is true, the THEN clause (the "action to be taken") is executed. After execution of the THEN clause, control branches around the ELSE clause (the "alternate action"), and execution continues with the next statement. Note that the THEN clause can contain a GO TO statement or some other control statement that would result in a different transfer of control.

If the comparison is not true, control branches around the THEN clause, and the ELSE clause is executed. Control then continues normally.

The IF statement might be as follows:

```
IF A = B

   THEN C = D;

   ELSE C = E;
```

If A is equal to B, the value of D is assigned to C, and control branches around the ELSE clause. If A is not equal to B, control branches around the THEN clause, and the value of E is assigned to C.

Either the THEN clause or the ELSE clause can contain some other control statement that causes a branch, either conditional or unconditional. If the THEN clause contains a GO TO statement, for example, there is no need to specify an ELSE clause. Consider the following example:

```
IF A = B

   THEN GO TO LABEL_1;

next-statement
```

If A is equal to B, the GO TO statement of the THEN clause causes an unconditional branch to LABEL_1. If A is not equal to B, control branches around the THEN clause to the next statement, whether or not it is an ELSE clause associated with the IF statement.

Note: If the THEN clause does not cause a transfer of control and if it is not followed by an ELSE clause, the next statement will be executed whether or not the THEN clause is executed.

The expression following the IF keyword can be only an element expression; it cannot be an array or structure expression. It can, however, be a logical expression with more than one operator. For example:

```
IF A = B & C = D
   THEN GO TO R;
```

The same kind of test could be made with nested IF statements. The following three examples are equivalent:

```
IF A = B & C = D
   THEN GO TO R;
B = B + 1;
```

```
IF A = B
   THEN IF C = D
           THEN GO TO R;
B = B + 1;
```

```
IF A ¬= B
   THEN GO TO S;
IF C ¬= D
   THEN GO TO S;
GO TO R;
S: B = B + 1;
```

The DO Statement

The most common use of the DO statement is to specify that a group of statements is to be executed a stated number of times while a control variable is incremented each time through the loop. Such a group might take the form:

```
DO I = 1 TO 10;
  .
  .
  .
END;
```

The statements to be executed iteratively must be delimited by the DO statement and an associated END statement. In this case, the group of statements will be executed ten times, while the value of the control variable I ranges from 1 through 10. The effect of the DO and END statements would be the same as the following:

```
   I = 0;
A: I = I + 1;
   IF I > 10 THEN GO TO B;
     .
     .
     .
   GO TO A;
B: next statement
```

Note that the increment is made before the control variable is tested and that, in general, control goes to the statement following the group only when the value of the control variable exceeds the limit set in the DO statement. If a reference is made to a control variable after the last iteration is completed, the value of the variable will be one incrementation beyond the specified limit.

The DO statement can also be used with the WHILE option and no control variable, as follows:

```
DO WHILE (A = B);
```

This statement, heading a group, causes the group to be executed repeatedly so long as the value of A remained equal to the value of B.

The WHILE option can be combined with a control variable of the form:

```
DO I = 1 TO 10 WHILE (A = B);
```

This statement specifies two tests. Each time that I is incremented, a test is made to see that I has not exceeded 10. An additional test then is made to see that A is equal to B. Only if both conditions are satisfied will the statements of the group be executed.

More than one successive iteration specification can be included in a single DO statement. Consider each of the following DO statements:

```
DO I = 1 TO 10, 13 TO 15;
```

```
DO I = 1 TO 10, WHILE (A = B);
```

The first statement specifies that the DO group is to be executed a total of 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. The second DO statement specifies that the group is to be executed at least ten times. After the first ten executions have been completed, execution is to continue so long as A is equal to B. Note that in both statements, a comma is used to separate the two specifications. This indicates that a succeeding specification is to be considered only after the preceding specification has been satisfied.

The control variable of a DO statement can be used as a subscript in statements within the DO-group, so that each iteration deals with successive elements of a table or array. For example:

```
DO I = 1 TO 10;
   A(I)=I;
   END;
```

In this example, each element of A is set to 1, 2,...,10, respectively.

The increment in the iteration specification is assumed to be one unless some other value is stated, as follows:

```
DO I = 2 TO 10 BY 2;
```

This specifies that the loop is to be executed five times, with the value of I equal to 2, 4, 6, 8, and 10.

Noniterative DO Statements

The DO statement need not specify repeated execution of the statements of a DO-group. A simple DO statement, in conjunction with a DO-group can be used as follows:

```
DO;
 .
 .
 .
END;
```

The use of the simple DO statement in this manner merely indicates that the DO-group is to be treated logically as a single statement. It can be used to specify a number of statements to be executed in the THEN clause or the ELSE clause of an IF statement.

The CALL, RETURN, and END Statements

A subroutine may be invoked by a CALL statement that names an entry point of the subroutine. Control is returned to the activating, or invoking, procedure when a RETURN statement is executed in the subroutine or when execution of the END statement terminates the subroutine.

The RETURN statement with a parenthesized expression is used in a function procedure to return a value to a function reference. This form can be used only to return from a procedure that has been invoked by a function reference.

Normal termination of a program occurs as the result of execution of the final END statement of the main procedure or of a RETURN statement in the main procedure, either of which returns control to the operating system.

The STOP Statement

The STOP statement causes abnormal termination of a program.

EXCEPTION CONTROL STATEMENTS

The control statements, discussed in the preceding section, alter the flow of control whenever they are executed. Another way in which the sequence of execution can be altered is by the occurrence of a program interrupt caused by the raising of an exceptional condition.

In general, an exceptional condition is the occurrence of an unexpected action, such as an overflow error, or of an expected action, such as an end of file, that occurs at an unpredictable time. A detailed discussion of the handling of these conditions appears in Chapter 11, "Exceptional Condition Handling and Program Checkout."

The three exception control statements are the ON statement, the REVERT statement, and the SIGNAL statement.

The ON Statement

The ON statement is used to specify action to be taken when any subsequent occurrence of a specified condition causes a program interrupt. ON statements may specify particular action for any of a number of different conditions. For all of these conditions, a standard system action is specified as a part of PL/I, and if no ON statement is in force at the time an interrupt occurs, the standard system action will take place. For most conditions, the standard system action is to print a message and terminate execution.

The ON statement takes the form:

```
ON condition-name {SYSTEM;|on-unit}
```

The "condition name" is one of the keywords listed in Part II, Section H, "ON Conditions." The "on-unit" specifies a

programmer-defined action to be taken when that condition arises and an interrupt occurs; it can only be a null statement or a GO TO statement. The keyword SYSTEM (accompanied by the semicolon) is used in place of an on-unit to specify that the standard system action is to be taken if an interrupt occurs. For example:

```
ON OVERFLOW;
```

This statement has a null statement as its on-unit. It specifies that when an interrupt occurs as a result of an OVERFLOW condition being raised, the interrupt is to be ignored and execution is to continue from the point at which the interrupt occurred. If an ON statement for OVERFLOW were not in force and the condition arose, the standard system action for that condition would be taken.

The effect of an ON statement, the establishment of the on-unit or SYSTEM, can be changed within a block (1) by execution of another ON statement naming the same condition with either another on-unit or the word SYSTEM, which re-establishes standard system action, or (2) by the execution of a REVERT statement naming that condition. The action in effect at the time another block is activated is passed to the activated block and remains in effect in that activated block and in other blocks activated by it, unless another ON statement for the same condition is executed. When control returns to an activating block, actions are re-established as they existed.

The REVERT Statement

The REVERT statement is used to cancel the effect of all ON statements for the same condition that have been executed in the block in which the REVERT statement appears.

The REVERT statement, which must specify the condition name, re-establishes action for that condition as it was in the activating block at the time the current block was invoked.

The SIGNAL Statement

The SIGNAL statement simulates the occurrence of an interrupt for a named condition. It can be used to test the coding of the action established by execution of an ON statement. For example:

```
SIGNAL OVERFLOW;
```

This statement would simulate the occurrence of an overflow interrupt and would cause execution of the action established for the OVERFLOW condition. If an action has not been established, standard system action is taken.

PROGRAM STRUCTURE STATEMENTS

The program structure statements are those statements used to delimit sections of a program into blocks and groups . These statements are the PROCEDURE statement, the END statement, the ENTRY statement, the BEGIN statement, and the DO statement . The concept of blocks and groups is fundamental to a proper understanding of PL/I and is dealt with in detail in Chapters 6, 7, and 10.

Proper division of a program into blocks simplifies the writing and testing of the program, particularly when a number of programmers are co-operating in writing a single program. It may also result in more efficient use of storage, since dynamic storage of the automatic class is allocated on entry to the block in which data of this class is declared.

The PROCEDURE Statement

The principal function of a procedure block, which is delimited by a PROCEDURE statement and an associated END statement, is to define a sequence of operations to be performed upon specified data. This sequence of operations is given a name (the label of the PROCEDURE statement) and can be invoked from any point at which the name is known.

Every program must have at least one PROCEDURE statement and one END statement. A program may consist of a number of separately written procedures linked together. A procedure may also contain other procedures nested within it. These internal procedures may contain declarations that are treated (unless otherwise specified) as local definitions of names. Such definitions are not known outside their own block, and the names cannot be referred to in the containing procedure. The automatic storage associated with these names is allocated upon entry to the block in which such a name is defined, and it is freed upon exit from the block.

The sequence of statements defined by a procedure can be executed at any point at which the procedure name is known. A procedure is invoked either by a CALL statement or by the appearance of its name in an expression, in which case the procedure is called a function procedure. A function reference causes a value to be calculated and returned to the function reference for use in the evaluation of the expression.

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. A procedure may therefore operate upon different data when it is invoked from different points. A value is returned from a function procedure to a function reference by means of the RETURN statement.

The ENTRY Statement

The ENTRY statement is used to provide an alternate entry point to the procedure in which it appears and, possibly, an alternate parameter list to which arguments can be passed, corresponding to that entry point.

Note: The ENTRY statement specifies an entry to the procedure in which it appears; the ENTRY attribute specifies other procedures that are invoked from the procedure in which the ENTRY attribute specification appears.

The BEGIN Statement

Local definitions of names can also be made within begin blocks, which are delimited by a BEGIN statement and an associated END statement. Begin blocks, however, are executed in the normal flow of a program, either sequentially or as a result of a GO TO or an IF statement transfer. It is useful for delimiting a section of a program in which some automatic storage is to be allocated.

Each begin block must be nested within a procedure or another begin block.

The DO Statement

Another kind of program structure is provided by the DO-group, which is delimited by a DO statement and an associated END statement. A DO-group does not have any effect upon the allocation of storage or the meaning of names. A DO-group specifies that the statements contained within it are to be considered as an entity for the purpose of flow of control.

A DO statement may specify repeated execution of a sequence of statements until a criterion is satisfied, or it may indicate within an IF statement that a group of statements is to be taken together as the whole of the THEN clause or of the ELSE clause.

CHAPTER 6: BLOCKS, FLOW OF CONTROL, AND STORAGE ALLOCATION

This section discusses how statements can be organized into blocks to form a PL/I program, how control flows within a program from one block of statements to another, and how storage may be allocated for data within a block of statements.

BLOCKS

A block is a delimited sequence of statements that constitutes a section of a program. It localizes names declared within the block and limits the allocation of variables. There are two kinds of blocks: procedure blocks and begin blocks.

PROCEDURE BLOCKS

A procedure block, simply called a procedure, is a sequence of statements headed by a PROCEDURE statement and ended by an END statement, as follows:

```
label: PROCEDURE;
         .
         .
         .
       END [label];
```

All procedures must be named because the procedure name is the primary point of entry through which control can be transferred to a procedure. A PROCEDURE statement must have one and only one label. An example of a procedure follows:

```
READIN: PROCEDURE;
        statement-1
        statement-2
          .
          .
          .
        statement-n
        END READIN;
```

In general, control is transferred to a procedure through a reference to the name of the procedure. Thus, the procedure in the above example would be given control by a reference to its name READIN.

A PL/I program consists of one or more such procedures, each of which may contain other procedures and/or begin blocks.

BEGIN BLOCKS

A begin block is a set of statements headed by a BEGIN statement and ended by an END statement, as follows:

```
[label:]...BEGIN;
             .
             .
             .
           END [label];
```

Unlike a procedure block, a label is optional for a begin block. If one or more labels are prefixed to a BEGIN statement, they serve only to identify the starting point of the block. (Control may pass to a begin block without reference to the name of that block, although control can be transferred to a labeled BEGIN statement by execution of a GO TO statement.) An example of a begin block follows:

```
B:  CONTRL: BEGIN;
                 statement-1
                 statement-2
                 .
                 .
                 .
                 statement-n
                 END;
```

Unlike procedures, begin blocks generally are not given control through special references to them. The normal sequence of control governing ordinary statement execution also governs the execution of begin blocks. Control passes into a begin block sequentially, following execution of the preceding statement.

Begin blocks are not essential to the construction of a PL/I program. However, there are times when it is advantageous to use begin blocks to delimit certain areas of a program. These advantages are discussed in this chapter and in Chapter 7, "Recognition of Names."

INTERNAL AND EXTERNAL BLOCKS

Any block can contain one or more blocks. That is, a procedure, as well as a begin block, can contain other procedures and begin blocks. However, there can be no overlapping of blocks; a block that contains another block must totally encompass that block.

A procedure block that is contained within another block is called an internal procedure. A procedure block that is not contained within another block is called an external procedure. There must always be at least one external procedure in a PL/I program. (Note: With System/360 implementations, each external procedure is compiled separately.)

Begin blocks are always internal; they must always be contained within another block.

Internal procedure and begin blocks can also be referred to as nested blocks. Nested blocks may have blocks nested within them, and so on. The maximum level of nesting permitted by the D-Compiler is three, with the external procedure considered at level one. (The outermost block always must be an external procedure.) Consider the following example:

```
A:  PROCEDURE;
    statement-a1
    statement-a2
    statement-a3
    B:  BEGIN;
        statement-b1
        statement-b2
        statement-b3
        END;
    statement-a4
    statement-a5
    C:  PROCEDURE;
        statement-c1
        statement-c2
        D:  BEGIN
            statement-d1
            statement-d2
            statement-d3
            statement-d4
            END;
        END;
    statement-a6
    statement-a7
    END;
```

In the above example, procedure block A is an external procedure because it is not contained in any other block. Block B is a begin block that is contained in A; it contains no other blocks. Block C is an internal procedure; it contains begin block D. This example contains three levels of nesting. A is at the first level, B and C are at the second level, and D is at the third level.

Note: The END statement always closes (i.e., ends) that unclosed block headed by the BEGIN or PROCEDURE statement or an unclosed DO-group headed by the DO statement that physically precedes, and appears closest to the END statement. If a label follows END, it must be the label of the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no corresponding END.

ACTIVATION AND TERMINATION OF BLOCKS

ACTIVATION

Although the begin block and the procedure have a physical resemblance and play the same role in the allocation and freeing of storage, as well as in delimiting the scope of names, they differ in the way they are activated and executed. A begin block, like a single statement, is activated and executed in the course of normal sequential program flow, and, in general, can appear wherever a single statement can appear. For a procedure, however, normal sequential program flow passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of that procedure. The only way in which a procedure can be activated is by a procedure reference.

A procedure reference is the appearance of an entry name (defined below) in one of the following contexts:

1. After the keyword CALL in a CALL statement
2. As a function reference (see Chapter 10, "Subroutines and Functions" for details)

This chapter uses examples of the first of these; that is, with the procedure reference of the form:

```
CALL entry-name;
```

The material, however, is relevant to the other form as well.

An entry name is defined as either of the following:

1. The label of a PROCEDURE statement
2. The label of an ENTRY statement appearing within a procedure

The first of these is called the primary entry point to a procedure; the second is known as a secondary entry point to a procedure. (Note that for the D-Compiler an entry name of an external procedure cannot exceed six characters.) The following is an example of a procedure containing secondary entry points:

```
   A: PROCEDURE;
      statement-1
      statement-2
ERRT: ENTRY;
      statement-3
      statement-4
      statement-5
RETR: ENTRY;
      statement-6
      statement-7
      statement-8
      END;
```

In this example, A is the primary entry point to the procedure, while ERRT and RETR specify secondary entry points.

When a procedure reference is executed, the procedure containing the specified entry point is activated and is said to be invoked; control is transferred to the specified entry point. 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 block it invokes.

Whenever a procedure is invoked at its primary entry point, execution begins with the first executable statement in the invoked procedure. However, when a procedure is invoked at a secondary entry point, execution begins with the first executable statement following the ENTRY statement that defines that secondary entry point. Therefore, if all of the numbered statements in the last example are executable, the statement CALL A would invoke procedure A at its primary entry point, and execution would begin with statement-1; the statement CALL ERRT would invoke procedure A at the secondary entry point ERRT, and execution would begin with statement-3; the statement CALL RETR would invoke procedure A at its other secondary entry point, and execution would begin with statement-6. Note that any ENTRY statements encountered during sequential flow are never executed; control flows around the ENTRY statement as though the statement were a comment.

Any procedure, whether external or internal, can always invoke an external procedure, but it cannot always invoke an internal procedure that is contained in some other procedure. Those internal procedures that are at the first level of nesting relative to a containing procedure can always be invoked by that containing procedure, or by each other. For example:

```
PRMAIN: PROCEDURE;
        statement-1
        statement-2
        statement-3
        A: PROCEDURE;
           statement-a1
           statement-a2
           B: PROCEDURE;
              statement-b1
              statement-b2
              END B;
           END A;
        statement-4
        statement-5
        C: PROCEDURE;
           statement-c1
           statement-c2
           END;
        statement-6
        statement-7
        END;
```

In this example, PRMAIN can invoke procedures A and C, but not B; procedure A can invoke procedures B and C; procedure B can invoke procedure C; and procedure C can invoke procedure A, but not B. Note that recursion is not permitted; that is, a procedure cannot be invoked while it is active. Hence, a procedure cannot invoke itself.

The foregoing discussion on the activation of blocks presupposes that a program has been activated in the first place. A program becomes active only when the operating system invokes the initial procedure. For System/360 implementations, this procedure, also called the main procedure, must be an external procedure whose PROCEDURE statement has been specified with the OPTIONS(MAIN) designation, as shown in the following example:

```
CONTRL: PROCEDURE OPTIONS(MAIN);
        CALL A;
        CALL B;
        CALL C;
        END;
```

In this example, CONTRL is the initial procedure and it invokes other procedures in the program.

The following is a summary of what has been stated, or at least implied, about the activation of blocks:

- A program becomes active when the initial procedure is activated by the operating system.
- Except for the initial procedure, external and internal procedures contained in a program are activated only when they are invoked by a procedure reference.
- A procedure cannot be invoked while it is active.
- Begin blocks are activated through normal sequential flow.

- The initial procedure remains active for the duration of the program.

- All activated blocks remain active until they are terminated (see below).

TERMINATION

In general, a procedure block is terminated when control passes back to the invoking block or to some other active block. Similarly, a begin block is terminated when control passes to another active block. There are a number of ways by which such transfers of control can be accomplished, and their interpretations differ according to the type of block being terminated.

Begin Block Termination

A begin block is terminated when any of the following occurs:

1. Control reaches the END statement for the block. When this occurs, control moves sequentially to the statement physically following the END.

2. The execution of a GO TO statement within the begin block (or any block activated from within that begin block) transfers control to a point not contained within the block.

3. A STOP statement is executed (thereby terminating execution).

A GO TO statement of the type described in item 2 can also cause the termination of other blocks as follows:

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 terminated.

For example, if begin block B is contained in begin block A, then a GO TO statement in B that transfers control to a point contained in neither A nor B effectively terminates both A and B. This case is illustrated below:

```
FRST: PROCEDURE OPTIONS(MAIN);
      statement-1
      statement-2
      statement-3
      A: BEGIN;
         statement-a1
         statement-a2
         B: BEGIN;
            statement-b1
            statement-b2
            GO TO LAB;
            statement-b3
            END;
         statement-a3
         END;
      statement-4
      statement-5
 LAB: statement-6
      statement-7
      END;
```

After FRST is invoked, the first three statements are executed and then begin block A is activated. The first two statements in A are executed and then begin block B is activated (A remaining active). When the GO TO statement in B is executed, control passes to statement-6 in FRST. Since statement-6 is contained in neither A nor B, both A and B are terminated. Thus, the transfer of control out of begin block B results in the termination of intervening block A as well as termination of block B.

Procedure Termination

A procedure is terminated when one of the following occurs:

1. Control reaches a RETURN statement within the procedure. The execution of a RETURN statement causes control to be returned 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 will be resumed.

2. Control reaches the END statement of the procedure. Effectively, this is equivalent to the execution of a RETURN statement.

3. The execution of a GO TO statement within the procedure (or any block activated from within that procedure) transfers control to a point not contained within the procedure.

4. A STOP statement is executed (thereby terminating execution).

Items 1, 2, and 3 are normal procedure terminations; item 4 is an abnormal procedure termination.

As with a begin block, the type of termination described in item 3 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. Consider the following example:

```
A: PROCEDURE OPTIONS(MAIN);
   statement-1
   statement-2
   B: BEGIN;
      statement-b1
      statement-b2
      CALL C;
      statement-b3
      END;
   statement-3
   statement-4
   C: PROCEDURE
      statement-c1
      statement-c2
      statement-c3
      D: BEGIN;
         statement-d1
         statement-d2
         GO TO LAB;
         statement-d3
         END;
      statement-c4
      END;
   statement-5
LAB: statement-6
   statement-7
   END;
```

In the above example, 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.

Program Termination

A program is terminated when either of the following occurs:

1. A STOP statement is executed anywhere within the program. This is called abnormal program termination, which, for the D-Compiler, effectively results in an immediate transfer of control to the final END statement in the main procedure.
2. Control reaches a RETURN statement or the final END statement in the main procedure. This is called normal program termination.
3. A null on-unit is executed for the ERROR condition or the standard system action for the ERROR condition is taken. The standard system action for this condition results in a return of control to the operating system control program.

STORAGE ALLOCATION

Storage allocation is the process of associating an area of storage with a variable so that the data item(s) to be represented by the variable may be recorded internally. When storage has been associated with a variable, the variable is said to be allocated. Allocation for a given variable may take place statically, that is, before the execution of the program, or dynamically, during execution. A variable that is allocated statically remains allocated for the duration of the program. A variable that is allocated dynamically will relinquish its storage either upon the termination of the block containing that variable or by pointer manipulation.

The manner in which storage is allocated for a variable is determined by the storage class of that variable. There are three storage classes: static, automatic, and based. Each storage class is specified by its corresponding storage class attribute: STATIC, AUTOMATIC, and BASED, respectively. The last two define dynamic storage allocation.

Storage class attributes may be declared explicitly for element, array, and major structure variables. If a variable is an array or a major structure variable, the storage class declared for that variable applies to all of the elements in the array or structure.

All variables that have not been explicitly declared with a storage class attribute are assumed to have the AUTOMATIC attribute, with one exception: any variable that has the EXTERNAL attribute is assumed to have the STATIC attribute.

Static Storage

All variables that have the STATIC attribute are allocated storage before the

execution of the program begins and they remain allocated for the duration of the program. For example:

```
CNTRL: PROCEDURE OPTIONS (MAIN);
       DECLARE (X,Y,Z) FIXED (5,0)
         STATIC EXTERNAL;
       X=1; Y=1; Z=1;
       CALL OUTP;
       CALL NEXT;
       CALL REVERS;
       END;
       .
       .
       .

OUTP:  PROCEDURE;
       DECLARE X FIXED (5,0)
         STATIC EXTERNAL;
            .
            .
            .
       PUT EDIT ('OUTP INVOCATION#', X)
         (A(17), F(6));
            .
            .
            .
       X=X+1;
       END;
```

Before execution of a program begins, all static variables are allocated. Thus, in the above example, X, Y, and Z are allocated before the initial procedure CNTRL is invoked by the operating system. When CNTRL is invoked, it sets X, Y, and Z to 1. (X is the same variable in both CNTRL and OUTP because it has been declared EXTERNAL in both.) Therefore, the first time that procedure OUTP is invoked, X has the value 1 and execution of the PUT statement causes this value to be written into the stream (along with an identifying character string). Before OUTP is terminated, the value of X is increased by 1 by the assignment statement. If OUTP is invoked a second time, and if the value of X is not changed elsewhere in the program, X has the value 2. Now when the PUT statement is executed for the second time, the new value of X is transmitted, etc. Thus, the static variable X is used to record the number of times that OUTP is invoked.

Note that even though OUTP could be activated and terminated several times, X, being static, retains a value throughout the program. The EXTERNAL attribute is given to X only to allow X to be initialized in the main procedure (CNTRL).

Automatic Storage

A variable that has the AUTOMATIC attribute is allocated storage upon activation of the block in which that variable is declared. The variable remains allocated as long as the block remains active; it is freed when the block is terminated. Once a variable is freed, its value is lost.

Based Storage

A variable that has the BASED attribute is known as a based variable. Storage for a based variable is, in effect, allocated by the programmer through the use of a READ or LOCATE statement with a SET option. This initializes the pointer variable associated with the based variable in such a way that the description of the based variable applies to the storage area "pointed to" by the pointer variable. The pointer variable can be initialized in other ways (e.g., by the ADDR built-in function) so that the description of the based variable can overlay storage that has been allocated for other variables.

The pointer variable can be manipulated so that the description of the based variable applies to different storage areas. That is, the value of the pointer variable can be changed so that the storage area associated with the old pointer value is no longer described by the based variable; the description of the based variable now applies to the storage area associated with the new pointer value. A complete discussion of this topic is given in Chapter 12, "Based Variables and Pointer Variables."

PROLOGUES AND EPILOGUES

Each time a block is activated, certain activities must be performed before control can reach the first executable statement in the block. This set of activities is called a prologue. Similarly, when a block is terminated, certain activities must be performed before control can be transferred out of the block; this set of activities is called an epilogue.

Prologues and epilogues are the responsibility of the compiler and not of the programmer. They are discussed here because knowledge of them may assist the programmer in improving the performance of his program.

Prologues

A prologue is a compiler-written routine logically appended to the beginning of a block and executed as the first step in the activation of a block. In general, activities performed by a prologue are as follows:

- Allocation of storage for automatic variables.
- Establishment of the inheritance of on-units.
- Allocation of storage for dummy arguments that may be passed from the block.

Epilogues

An epilogue is a compiler-written routine logically appended to the end of a block and executed as the final step in the termination of a block. In general, the activities performed by an epilogue are as follows:

- Re-establishment of the on-unit environment existing before the block was activated.
- Release of storage for all automatic variables allocated in the block.

A PL/I program consists of a collection of identifiers, constants, and special characters used as operators or delimiters. Identifiers themselves may be either keywords or names with a meaning specified by the programmer. The PL/I language is constructed so that the compiler can usually determine from context whether or not an identifier is a keyword, so there are very few reserved words that must not be used for programmer-defined names (see note below). Any identifier may be used as a name; the only restriction is that at any point in a program a name can have one and only one meaning. For example, the same name cannot be used for both a file and a floating-point variable.

Note: The 48-character set operation identifiers GT, GE, NE, LE, LT, NG, NL, NOT, OR, AND, and CAT are fully reserved when the 48-character set is being used; when such is the case, these identifiers cannot be declared in any way. The built-in function identifiers TIME, DATE, and NULL are partially reserved and cannot be implicitly declared. No other keywords are reserved. (Although the PL/I Subset Language partially reserves the identifiers SYSIN and SYSPRINT, the D-Compiler does not. However, some care should be taken if the programmer associates these identifiers with the standard system files defined for the D-Compiler. This is covered in detail under "Standard Files" in Chapter 8.)

It is not necessary, however, for a name to have the same meaning throughout a program. A name declared within a block has a meaning only within that block. Outside the block it is unknown unless the same name has also been declared in the outer block. In this case, the name in the outer block refers to a different object. This enables programmers to specify local definitions and, hence, to write procedures or begin blocks without knowing all the names being used by other programmers writing other parts of the program.

Since it is possible for a name to have more than one meaning, it is important to define which part of the program a particular meaning applies to. In PL/I a name is given attributes and a meaning by a declaration (not necessarily explicit). The part of the program for which the meaning applies is called the scope of the declaration of that name. In most cases, the scope of a name is determined entirely by the position at which the name is declared within the program (or assumed to be declared if the declaration is not explicit).

In order to understand the rules for the scope of a name, it is necessary to understand the terms "contained in" and "internal to."

Contained In:

> All of the text of a block, from the PROCEDURE or BEGIN statement through the corresponding END statement, is said to be contained in that block. Note, however, that the label of the BEGIN or PROCEDURE statement heading the block, as well as the label of any ENTRY statement that applies to the block, are not contained in that block. Nested blocks are contained in the block in which they appear.

Internal To:

> 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. Note that entry names of a procedure (or the labels of a BEGIN statement, if the block is a begin block) are not contained in that block. Consequently, they are internal to the containing block. Entry names of an external procedure are treated by the D-Compiler as if they were internal to the external procedure, but declared with the EXTERNAL attribute.

In addition to these terms, the different types of declaration are important. The three different types -- explicit declaration, contextual declaration, and implicit declaration -- are discussed in the following sections.

EXPLICIT DECLARATIONS

A name is explicitly declared if it appears:

1. In a DECLARE statement
2. In a parameter list
3. As a statement label
4. As the label of a PROCEDURE or ENTRY statement

The appearance of a name in a parameter list is the same as if a DECLARE statement for that name appeared immediately following the PROCEDURE statement in which the parameter list occurs (though the same name may also appear in a DECLARE statement internal to the same block).

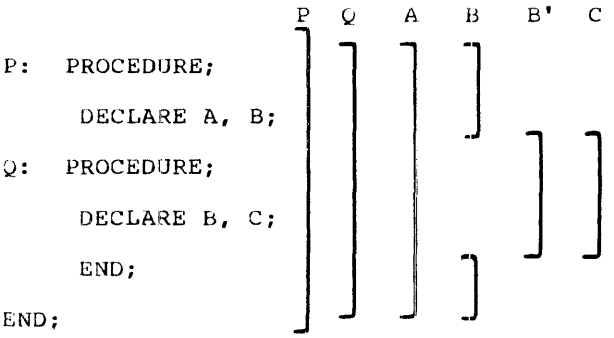
The appearance of a name as the label of either an internal PROCEDURE or an internal ENTRY statement is the same as if it were declared in a DECLARE statement immediately preceding the PROCEDURE statement for the procedure to which it refers. The labels of the PROCEDURE and ENTRY statements of an external procedure are treated by the D-Compiler as if they appeared in a DECLARE statement with the EXTERNAL attribute in the external procedure.

The appearance of a statement label prefix constitutes an explicit declaration equivalent to the declaration of a variable in a DECLARE statement internal to the same block as the statement to which it applies.

SCOPE OF AN EXPLICIT DECLARATION

The scope of an explicit declaration of a name is that block to which the declaration is internal, but excluding all contained blocks to which another explicit declaration of the same identifier is internal.

For example:



The brackets to the right indicate the scope of the names. B and B' indicate the two distinct uses of the name B.

CONTEXTUAL DECLARATIONS

When an identifier appears in a context where only an entry name can appear, its attributes can be determined without explicit declaration of that identifier. Such an identifier is said to be contextually declared as an entry name only if it does not lie within the scope of an explicit declaration for that same identifier. Entry names are the only identifiers that can be so declared.

An identifier that has not been explicitly declared will be recognized and contextually declared as an entry name in either of the following cases:

1. If the identifier immediately follows the keyword CALL in a CALL statement.
2. If the identifier is immediately followed by a parenthesized list in a context where an expression is expected; i.e., if the identifier appears as the function name in a function reference with arguments.

A contextually declared entry name is given the EXTERNAL attribute by default.

SCOPE OF A CONTEXTUAL DECLARATION

The scope of a contextual declaration is determined as if the declaration were made in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name appears.

Note that a contextual declaration has the same effect as if the name were declared in the external procedure, even when the statement that causes the contextual declaration is internal to a block (called B, for example) that is contained in the external porcedure. Consequently, the name is known throughout the entire external procedure, except for any blocks in which the name is explicitly declared. It is as if block B has inherited the declaration from the containing external procedure.

Since a contextual declaration cannot exist within the scope of an explicit declaration, it is impossible for the context of an identifier to add to the attributes established for that identifier in an explicit declaration. Thus, a parameter, since it is explicitly declared by its appearance in a PROCEDURE or ENTRY statement, can never be contextually declared as an entry name. A complementary explicit declaration of the ENTRY attribute must be given for the parameter in its containing procedure if the parameter is to be used as an entry name within that procedure. This rule is illustrated by the example below.

The following is invalid:

```
P:  PROCEDURE (FNAM);
    CALL FNAM;
    .
    .
    .
    END;
```

FNAM appears in the parameter list of the PROCEDURE statement and is therefore explicitly declared. Since no further explicit declarations are given for FNAM, it is given the attributes DECIMAL FLOAT by default, and hence must be an arithmetic variable. Therefore, the appearance of FNAM in the CALL statement is in error because FNAM is not an entry name and it cannot be contextually declared as an entry name. The example could be corrected by adding a DECLARE statement as follows:

```
P:  PROCEDURE (FNAM);
    DECLARE FNAM ENTRY;

    CALL FNAM;
    .
    .
    .

    END;
```

Now the CALL statement is valid because of the complementary explicit declaration of FNAM with the ENTRY attribute.

IMPLICIT DECLARATION

If a name appears in a program and is not explicitly or contextually declared, it is said to implicitly declared. The scope of an implicit declaration is determined as if the name were declared in a DECLARE statement immediately following the first PROCEDURE statement of the external procedure in which the name is used.

An implicit declaration causes default attributes to be applied, depending upon the first letter of the name. If the name begins with any of the letters I through N it is given the attributes FIXED BINARY (15). If the name begins with any other letter including one of the alphabetic extenders $, #, or @, it is given the attributes FLOAT DECIMAL (6). (The default precisions are those defined for System/360 implementations.)

The identifiers TIME, DATE, and NULL cannot be implicitly declared; each is always assumed to refer to the corresponding built-in function, unless, of course, it has been explicitly declared otherwise.

EXAMPLES OF DECLARATIONS

Scopes of data declarations are illustrated in Figure 7-1. The brackets to the left indicate the block structure, the brackets to the right show the scope of each declaration of a name. In the diagram, the scopes of the two declarations of Q and R are shown as Q and Q' and R and R'.

P is declared in the block A and known throughout A since it is not redeclared.

Q is declared in A, and redeclared in B. The scope of the first declaration is all of A except B; the scope of the second declaration is block B only.

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

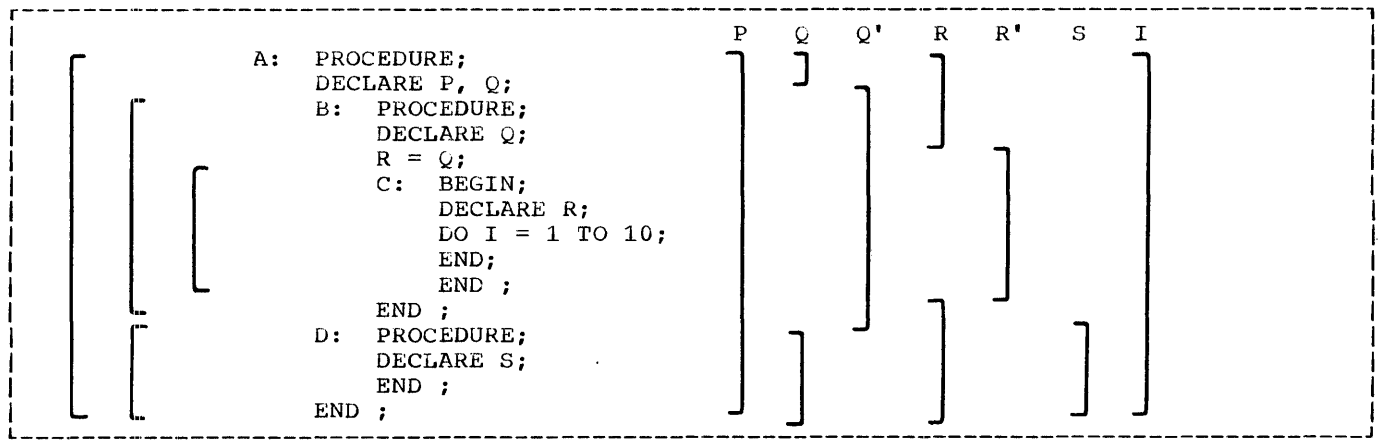


Figure 7-1. Scopes of Data Declarations

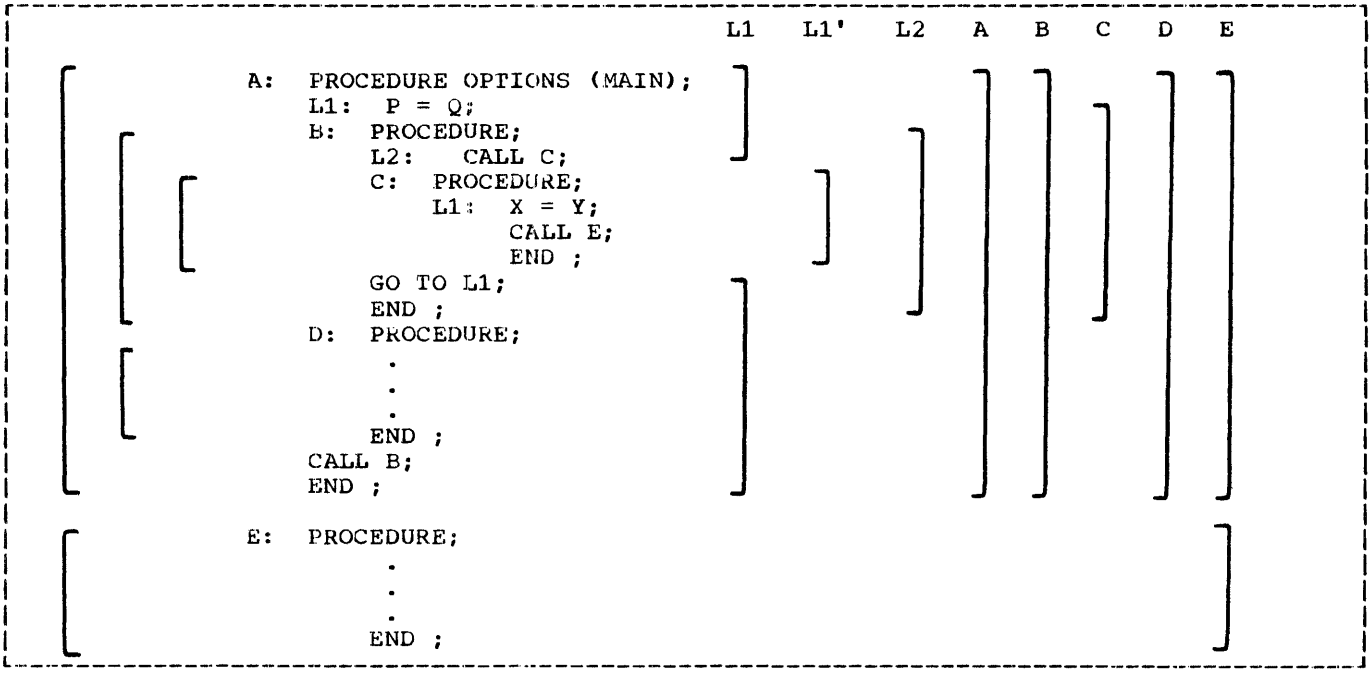


Figure 7-2. Scopes of Entry and Label Declarations

implicit declaration of R in A,the external procedure. Two separate names with different scopes exist, therefore. The scope of the explicitly declared R is C; the scope of the implicitly declared R is all of A except block C.

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.

S is declared within procedure D and is known only within D.

Scopes of entry name and statement label declarations are illustrated in Figure 7-2. The example shows two external procedures. The names of these procedures, A and E, are assumed to be explicitly declared with the EXTERNAL attribute within the procedures to which they apply. In addition, E is contextually declared in A as an EXTERNAL entry name by its appearance in the CALL statement in block C. The contextual declaration of E applies throughout block A and is linked to the explicit declaration of E that applies throughout block E. The scope of the name E is all of block A and all of block E. The scope of the name A is only all of the block A, and not E. Since recursion is not permitted, A could not be called from within E and hence A is not known within E.

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 is executed, control is transferred to L1 in block A, and block B is terminated.

D and B are explicitly declared in block A and can be referred to anywhere within A; but since they are INTERNAL, they cannot be referred to in block E (unless passed as an argument to E).

C is explicitly declared in B and can be referred to from within B, but not from outside B.

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.

APPLICATION OF DEFAULT ATTRIBUTES

The attributes associated with a name comprise those explicitly, contextually, or implicitly declared, as well as those assumed by default. The default for each attribute is given in Part II, Section I, "Attributes."

THE INTERNAL AND EXTERNAL ATTRIBUTES

The scope of a name with the INTERNAL attribute is the same as the scope of its declaration. Any other explicit declaration of that name refers to a new object with a different, non-overlapping scope.

A name with the EXTERNAL attribute may be declared more than once in the same program, either in different external procedures or within blocks contained in external procedures. Each declaration of the name establishes a scope. These declarations are linked together and, within a program, all declarations of the same identifier with the EXTERNAL attribute refer to the same name. The scope of the name is the sum of the scopes of all the declarations of that name within the program.

Since these declarations all refer to the same thing, they must result in the same set of attributes. It may be impossible for the compiler to check this, particularly if the names are declared in different procedures, so care should be taken to ensure that different declarations of the same name with the EXTERNAL attribute do have matching attributes. The attribute listing, which is available as optional output from the D-Compiler, helps to check the use of names.

The D-Compiler restricts a name with the EXTERNAL attribute to six characters or less. This includes names that are EXTERNAL by default, such as file names and entry names of external procedures.

Example: The following example illustrates the points discussed in this chapter:

```
A: PROCEDURE OPTIONS (MAIN);
   DECLARE S CHARACTER(10);
   CALL SET(23168);
E: GET EDIT ...;
   B: BEGIN;
      DECLARE (X,Y) DECIMAL;
      GET EDIT(X,Y,N)...;
      CALL C(X,Y);
      C: PROCEDURE(P,Q);
         DECLARE S BINARY EXTERNAL;
            .
            .
            .
         GET EDIT(I)...;
         IF... THEN GO TO B;
         CALL D(I);
         CALL OUT(E);
      B: END C;
      D: PROCEDURE(N);
         PUT EDIT(N,S)...;
         END D;
      END B;
   GO TO E;
   END A;

OUT: PROCEDURE(R);
     DECLARE R LABEL,
             S BINARY EXTERNAL,
             Z DECIMAL FIXED,
             (M,L) STATIC,
             INTERNAL;
        .
        .
        .
     GO TO R;
SET: ENTRY(Z);
     X=Z;
     RETURN;
     END OUT;
```

A is an external procedure name; its scope is all of block A, plus any other blocks where A is declared (explicitly or contextually) as external.

S is explicitly declared in block A and block C. The character-string declaration applies to all of block A except block C; the 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 string S, and not to the S declared in block C.

N appears as a parameter in block D, but it is also used outside the block. Its appearance as a parameter establishes an explicit declaration of N within D, the reference outside D causes an implicit declaration of N in block A. These two uses of the name N refer to different objects, although in this case the objects have the same data attributes.

X and Y are known throughout B and could be referred to in blocks C or D within B, but not in that part of A outside B. The X used within the entry point SET is an implicit declaration of X within OUT and is not known outside OUT.

P and Q are parameters; their appearance in the parameter list is sufficient to constitute an explicit declaration.

I is not explicitly declared in the external procedure A; it is implicitly declared and is therefore known throughout A, even though it appears only within block C.

Within external procedure A, OUT and SET are contextually declared as entry names, since they follow the keyword CALL. They are therefore considered to be declared in A, and are given the EXTERNAL attribute by default.

The second external procedure in the example has two entry names, SET and OUT. These are considered to be explicitly declared with the EXTERNAL attribute. The two entry names SET and OUT are therefore known throughout the two external procedures.

The label B appears twice in the example, once as the label of a begin block, which is an explicit declaration of B as a label in A. It is redeclared as a label within block C by its appearance as a prefix to the END statement. The reference to B in the GO TO statement within block C refers to the label of the END statement within block C. Outside block C, any reference to B would be to the label of the begin block.

Note that 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, since E is known throughout A, transfers to E may be made from any point within A. Transfers out of a nested block are therefore possible, but, in general, transfers into such a block are not.

An exception to the above rule is shown in the external procedure OUT, where the label E from block A is passed as an argument to the label parameter R. The statement GO TO R causes control to pass to the label E, even though E is declared within A, and not known within OUT (this topic is fully discussed in Chapter 10, "Subroutines and Functions").

The variables M and L are declared within the block OUT to be STATIC, so each value is preserved between calls to OUT.

In order to make the S in OUT the same as the S in C, they have both been declared with the attribute EXTERNAL.

MULTIPLE DECLARATIONS AND AMBIGUOUS REFERENCES

Two or more declarations of the same identifier internal to the same block constitute a multiple declaration, unless at least one of the identifiers is declared within a structure in such a way that name qualification can be used to make the names unique.

Two or more declarations anywhere in a program of the same identifier as different names with the EXTERNAL attribute constitute a multiple declaration.

Multiple declarations are in error.

A name need have only enough qualification to make the name unique. Reference to a name is always taken to apply to the identifier declared in the innermost block containing the reference. An ambiguous reference is a name with insufficient qualification to make the name unique.

The following examples illustrate both multiple declarations and ambiguous references:

```
DECLARE 1 A, 2 C, 2 D, 3 E;
  BEGIN;
  DECLARE 1 A, 2 B, 3 C, 3 E;
  A.C = D.E;
```

In this example, A.C refers to C in the inner block; D.E refers to E in the outer block.

```
DECLARE 1 A, 2 B, 2 B, 2 C, 3 D, 2 D;
```

In this example, B has been multiply declared. A.D refers to the second D, since A.D is a complete qualification of only the second D; the first D would have to be referred to as A.C.D.

```
DECLARE 1 A, 2 B, 3 C, 2 D, 3 C;
```

In this example, A.C is ambiguous because neither C is completely qualified by this reference.

```
DECLARE 1 A, 2 A, 3 A;
```

In this example, A refers to the first A, A.A refers to the second A, and A.A.A refers to the third A.

```
DECLARE X;
DECLARE 1 Y, 2 X, 3 Z, 3 A,
             2 Y, 3 Z, 3 A;
```

In this example, 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.

PL/I provides input and output statements that enable data to be transmitted between the internal and external storage devices 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*, and transmission of data from a program to a data set is called *output*.

Data sets are stored on a variety of external storage media, such as punched cards, reels of magnetic tape, and magnetic disks. Despite their variety, external storage media have many common characteristics that permit standard methods of collecting, storing, and transmitting data. For convenience, thus, the general term *volume* is used to refer to a unit of external storage, such as a reel of magnetic tape or a disk pack, without regard to its specific physical composition.

The data items within a data set are arranged in distinct physical groupings called *blocks*. These blocks allow the data set to be transmitted and processed in portions rather than as a unit. For processing purposes, each block consists of one or more logical subdivisions called *records*, each of which can contain one or more data items.

A block is also called a *physical record*, because it is the unit of data that is physically transmitted to and from a volume. To avoid confusion between a physical record and its logical subdivisions, the logical subdivisions are called *logical records*.

When a block contains two or more records, the records are said to be *blocked*. Blocked records often permit more compact and efficient use of storage. Consider how data is stored on magnetic tape: the data between two successive interrecord gaps is one block, or physical record. If several logical records are contained within one block, the number of interblock gaps is reduced, and much more data can be stored on a full length of tape. For example, on a tape of density 800 characters/inch with an interrecord gap of 0.6 inches, a card image of 80 characters would take up 0.1 inches. If the records were unblocked, each record would require 0.1 inches, plus 0.6 inches for the interrecord gap, making a total of 0.7 inches. 100 records would therefore take up 70 inches of tape. If the records were blocked, however, at, say, 10 records to a block, each block of 10 records would take up 1 inch, plus 0.6 inches for the gap, making a total of 1.6 inches. Thus, 100 records would now take up only 16 inches of tape; this is less than 25 percent of the amount needed for unblocked records.

Most data processing applications are concerned with logical records rather than physical records. Therefore, the input and output statements of PL/I generally refer to logical records; this allows the programmer to concentrate on the data to be processed, without being directly concerned about its physical organization in external storage.

TYPES OF DATA TRANSMISSION

Two different types of data transmission can be used by a PL/I program, stream-oriented transmission and record-oriented transmission.

In *stream-oriented transmission*, the data in the data set is considered to be a continuous stream of data items in character form. Consequently, characters in the input stream are interpreted and converted where necessary to the specified internal form; on output, data items in internal form are converted where necessary to character form and added to the output stream. The GET and PUT statements are the data transmission statements used in stream-oriented transmission. Variables, to which input data items are assigned, and expressions from which output data items are transmitted, are generally specified in a data list with each GET or PUT statement.

Although data in the data set exists in record format, in stream transmission such organization is ignored within the program and the data is treated as though it actually were a continuous stream of individual data items.

In record-oriented transmission, data in the data set is considered to be a collection of discrete logical records, recorded in any format acceptable to the computer. No data conversion is performed during record transmission; on input it is transmitted exactly as it is recorded in the data set; on output it is transmitted exactly as it is recorded internally.

The READ, REWRITE, and WRITE statements cause a single logical record to be transmitted to or from a data variable or, in the case of READ with the SET option, to an intermediate, addressable buffer. The LOCATE statement allocates an area in a buffer to which data for a record can be assigned.

Note that although records may be blocked, in which case the physical record actually is transmitted to or from the data set as an entity, each data transmission statement in record I/O is concerned with a logical record. Blocked records are unblocked automatically.

The following discussion of files and file attributes should be of particular interest to a programmer using record-oriented transmission. File handling is simpler when using stream-oriented transmission, and, as can be noted, fewer attributes are applicable to stream files.

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 a symbolic representation of a data set called a file. This symbolic representation determines how input and output statements access and process the associated data set. Unlike a data set, however, a file has significance only within the source program and does not exist as a physical entity external to the program.

PL/I requires a file name to be declared for a file and allows the characteristics of a file to be described with keywords called file attributes, which are specified for the file name.

FILE ATTRIBUTES

The following lists show file attributes that are applicable to each type of data transmission:

Record Transmission	Stream Transmission
FILE	FILE
RECORD	STREAM
INPUT	INPUT
OUTPUT	OUTPUT
UPDATE	PRINT
ENVIRONMENT	ENVIRONMENT
SEQUENTIAL	
DIRECT	
BUFFERED	
UNBUFFERED	
KEYED	
BACKWARDS	

A detailed description of each of these attributes appears in Part II, Section I, "Attributes." The discussions below give a brief description of each attribute and show how attributes are declared for a file.

The FILE Attribute

The FILE attribute indicates that the associated identifier is a file name. For example, the identifier MASTER is declared to be a file name in the following statement:

```
DECLARE MASTER FILE...;
```

The FILE attribute must be explicitly declared for every file name and file name parameter, and it must always be the first attribute declared in a file declaration.

Alternative and Additive Attributes

The attributes associated with the FILE attribute fall into two categories: alternative attributes and additive attributes. An alternative attribute is one that is chosen from a group of attributes. If no explicit declaration is given for one of the alternative attributes in a group and if one of the alternatives is required, a default attribute is assumed in most cases.

An additive attribute is one that must be stated explicitly or is implied by another explicitly stated attribute. The additive attribute KEYED can be implied by the DIRECT attribute. The ENVIRONMENT attribute must always be declared explicitly for every file. An additive attribute can never be applied by default.

Alternative Attributes

PL/I provides four groups of alternative file attributes. Each group is discussed individually. Following is a list of the groups and the default for each:

Group Type Usage	Alternative Attributes	Default Attribute
	STREAM | RECORD	STREAM
Function	INPUT|OUTPUT|UPDATE	no default
Access	SEQUENTIAL | DIRECT	SEQUENTIAL
Buffering	BUFFERED|UNBUFFERED	BUFFERED

Note: No default is applied for the function attributes; one must always be specified. In the case of an UNBUFFERED file, INPUT or OUTPUT can appear in the OPEN statement rather than in a DECLARE statement. The scope of a file name must always be EXTERNAL. A file name can be explicitly declared to have this attribute; otherwise it is supplied automatically.

The STREAM and RECORD Attributes

The STREAM and RECORD attributes describe the type of data transmission (stream-oriented or record-oriented) to be used in input and output operations for the file.

The STREAM attribute causes a data set associated with a file to be treated as a continuous stream of data items recorded only in character form.

The RECORD attribute causes a data set associated with a file to be treated as a sequence of logical records, each record consisting of one or more data items recorded in any internal form acceptable to the implementation.

```
DECLARE MASTER FILE RECORD...,
        DETAIL FILE STREAM...;
```

The INPUT, OUTPUT, and UPDATE Attributes

The function attributes determine the direction of data transmission permitted for a file. The INPUT attribute applies to files that are to be read only. The OUTPUT attribute applies to files that are to be created, and hence are to be written only. The UPDATE attribute describes a file that is to be used for both input and output; it allows records to be inserted into an existing file and other records already in that file to be altered.

```
DECLARE
   DETAIL FILE INPUT...,
   REPORT FILE OUTPUT...,
   MASTER FILE UPDATE...;
```

The SEQUENTIAL and DIRECT Attributes

The access attributes apply only to a file with the RECORD attribute and describe how the records in the file are to be accessed.

The SEQUENTIAL attribute normally specifies that successive records in the file are to be accessed on the basis of their successive physical positions, such as they are on magnetic tape.

The DIRECT attribute specifies that a record in a file is to be accessed on the basis of its location in the file and not on the basis of its position relative to the record previously read or written. The location of the record is determined by a key; therefore, the DIRECT attribute implies the KEYED attribute. The associated data set must be in a direct-access volume.

The BUFFERED and UNBUFFERED Attributes

The buffering attributes apply only to a file that has the SEQUENTIAL and RECORD attributes. The BUFFERED attribute indicates that logical records transmitted to and from a file must pass through an intermediate internal-storage area. The size of a buffer usually corresponds to the size of the blocks (physical records) in the data set associated with the file (a discussion of block size and buffer allocation appears in this chapter in "ENVIRONMENT Attribute"). The use of buffers may help speed up processing by allowing an overlap of transmission and computing time. It further allows the automatic blocking and unblocking of records.

The UNBUFFERED attribute indicates that a logical record in a data set is not to pass through a buffer but will be transmitted directly to and from the internal storage associated with a variable. The logical records and physical records are the same size in a data set that is associated with an UNBUFFERED file.

Note: In the D-Compiler, the UNBUFFERED attribute always specifies that a record is

not to pass through any buffer or intermediate storage area. So-called "hidden buffers" are never used.

Additive Attributes

The additive attributes are:

PRINT

BACKWARDS

KEYED

ENVIRONMENT (option-list)

The PRINT Attribute

The PRINT attribute applies only to files with the STREAM and OUTPUT attributes. It indicates that the file is eventually to be printed, that is, the data associated with the file is to appear on printed pages, although it may first be written on some other medium. The PRINT attribute specifies that the associated record is to be created with the initial byte reserved for a printer control character.

The BACKWARDS Attribute

The BACKWARDS attribute indicates that a file is to be accessed in reverse order, beginning with the last logical record and proceeding through the file until the first logical record is accessed. The BACKWARDS attribute applies only to RECORD files with the SEQUENTIAL and INPUT attributes and only to data sets on magnetic tape.

The KEYED Attribute

The KEYED attribute indicates that each record in the file has a key and can be accessed using one of the key options (KEY or KEYFROM) of data transmission statements. Note that the KEYED attribute does not necessarily indicate that the actual keys exist or are to be written in the data set. The STREAM and PRINT attributes cannot be applied to a file that has the KEYED attribute. The use of keys is discussed in detail in "Environmental Considerations for Data Sets" and "Record-Oriented Transmission" in this chapter.

The ENVIRONMENT Attribute

The ENVIRONMENT attribute specifies information about the physical organization of the data set associated with a file. These characteristics are indicated in a parenthesized option list in the ENVIRONMENT attribute specification and are dependent upon the implementation. The option list for the D-Compiler is discussed in "Environmental Considerations for Data Sets."

Note: As stated earlier in this chapter, each file must be explicitly declared; the FILE attribute and the ENVIRONMENT attribute must appear in every file declaration.

OPENING AND CLOSING FILES

Before the data associated with a file can be transmitted by input or output statements, certain file preparation activities must occur, such as checking for the availability of external storage media, positioning the medium, and allocating appropriate programming support. Such activity is known as opening a file. Also, when processing is completed, the file must be closed. Closing a file involves releasing the facilities that were established during the opening of the file.

The PL/I Subset provides two statements, OPEN and CLOSE, to perform these functions. All files with the RECORD attribute must be explicitly opened before use. However, with STREAM files, explicit opening is optional. If an OPEN statement is not executed for a STREAM file, the file is opened automatically when the first GET or PUT is executed; in this case, automatic file preparation is exactly the same as if an explicit OPEN had been executed before the GET or PUT. All files, both STREAM and RECORD, not closed before completion of a program will be closed automatically upon completion of the program.

The following discussions show the effect of OPEN and CLOSE statements.

The OPEN Statement

Execution of an OPEN statement causes one or more files to be opened explicitly. The OPEN statement has the following basic format:

```
OPEN FILE(file-name) [option-list]
  [,FILE(file-name) [option-list]]...;
```

The option list of the OPEN statement can include INPUT or OUTPUT provided the file has the UNBUFFERED attribute. These attributes, when included as options in the OPEN statement, are merged with those stated in a DECLARE statement. The same attribute should not be listed in both an OPEN statement and a DECLARE statement for the same file, and, of course, there can be no conflict. The other option that can appear in the OPEN statement is the PAGESIZE option, used to specify layout of a print page. This is discussed later in this chapter.

The OPEN statement is executed by library routines that are loaded dynamically at the time the OPEN statement is executed. Consequently, execution time can be reduced if more than one file is specified in the same OPEN statement, since the routines need be loaded only once, regardless of the number of files being opened.

For a file to be opened explicitly, the OPEN statement must be executed before any of the input and output statements listed below in "Implicit Opening" are executed for the same file.

Implicit Opening

An implicit opening of a file occurs only when a GET or PUT statement is executed without the prior execution of an OPEN statement for that file. The effect of an implicit opening is the same as if an OPEN statement for the file had been executed before the GET or PUT statement. All files implicitly opened by a GET statement must be declared explicitly as INPUT, and all files implicitly opened by a PUT must be declared explicitly as OUTPUT.

Merging of Attributes

There must be no conflict between the attributes specified in a file declaration and the attributes merged as a result of explicit file opening. For example, a conflict exists when a file is given the BACKWARDS attribute in a DECLARE statement and then is given the OUTPUT attribute in an OPEN statement. Since the attributes BACKWARDS and OUTPUT are in conflict, an error message will be generated during compilation of the program.

Associating Data Sets with Files

With the D-Compiler, a file name is associated with a data set by using the MEDIUM option in the PL/I Subset ENVIRONMENT attribute and, if necessary, the ASSGN statement from the DOS/TOS Job Control Language. This method of associating data sets and file names is described later in this chapter in the discussion of the MEDIUM option under the heading "The ENVIRONMENT Attribute."

The CLOSE Statement

The basic form of the CLOSE statement is:

```
CLOSE FILE (file-name)
      [,FILE(file-name)]...;
```

Executing a CLOSE statement dissociates the specified file from the data set with which it became associated when the file was opened. The CLOSE statement also dissociates from the file an INPUT or OUTPUT attribute established for it by an explicit opening. If desired, a new INPUT or OUTPUT attribute may be specified for the file name in a subsequent OPEN statement. However, all attributes explicitly given to the file name in a DECLARE statement remain in effect.

As with the OPEN statement, closing more than one file with a single CLOSE statement may save execution time.

Note: Closing an already closed file or opening an already opened file has no effect.

PAGE LAYOUT FOR PRINT FILES

The overall layout of a page in a file that has the PRINT attribute is controlled by means of the PAGESIZE option of the OPEN statement. For example:

```
DECLARE REPORT FILE OUTPUT PRINT
   ENVIRONMENT (option-list);

OPEN FILE (REPORT) PAGESIZE(55);
```

The specification PAGESIZE(55) indicates that each page should contain a maximum of 55 lines. An attempt to write on a page after 55 lines have already been written (or skipped) will raise the ENDPAGE condition. The standard system action for the ENDPAGE condition is to skip to a new page,

but the programmer can establish his own action through use of the ON statement.

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 the first time. This can be useful, for example, if a footing is to be written at the bottom of each page. Consider the following example:

```
        ON ENDPAGE(REPORT) GO TO FOOT;
          .
          .
          .
  FOOT: PUT FILE(REPORT) SKIP EDIT
            (FOOTING) (A);
        PUT FILE(REPORT) PAGE;
        N = N + 1;
        PUT FILE(REPORT) EDIT ('PAGE ',N)
            (A,F(3));
        PUT FILE(REPORT) SKIP (3);
        GO TO NEXT;
```

Assume that REPORT has been opened with PAGESIZE(55), as shown in the previous example. When an attempt is made to write on line 56 (or to skip beyond line 55), the ENDPAGE condition will arise, and the GO TO FOOT statement will be executed. The first PUT statement specifies that a line is to be skipped, and the value of FOOTING, a character string, is to be printed on line 57 (when ENDPAGE arises, the current line is always PAGESIZE+1). The second PUT statement causes a skip to the next page and the ENDPAGE counter is automatically reset for the new page. The page number is incremented, and the character string 'PAGE ' and the new page number N are printed. Note that a blank is included as part of the character string to separate the word from the page number. The F(3) format item allows the page number to go as high as 999. The final PUT statement causes three lines to be skipped, so that the next printing will be on line 4. The GO TO NEXT statement transfers control to the statement labeled NEXT.

The maximum number of characters to be printed on each line (i.e., the line size) is equal to the fixed length record size specified in the ENVIRONMENT attribute for the file (see the ENVIRONMENT attribute later in this chapter). An attempt to write more than the maximum number of characters specified without skipping to a new line or page will cause the excess characters to be placed on the next line.

The PAGESIZE option can be specified only for a file with the PRINT attribute and it can be specified only in the OPEN statement.

Further details of writing in PRINT files appear later in this chapter in "Data Transmission."

STANDARD FILES

Two standard system files are provided that can be used by any PL/I Subset program. These files are referred to in the PL/I Subset by specifying a GET or PUT with neither the FILE nor the STRING option. For example:

```
GET EDIT...;
PUT EDIT...;
```

For the above GET, the DOS/TOS system input device SYSIPT is referred to; for the above PUT, the DOS/TOS system output device SYSLST is referred to. When these standard DOS/TOS input/output devices are referred to as shown above by specifying neither the FILE nor STRING option in a GET or PUT, no explicit file declaration need be given. The association of the files with SYSIPT and SYSLST is automatic. Indeed, if files are explicitly declared and associated with SYSIPT or SYSLST (using the MEDIUM option of the ENVIRONMENT attribute), certain rules must be observed when referring to the files to ensure that items are transmitted to or from the output or input stream in the proper order. These will be discussed later.

With the PL/I DOS/TOS D-Compiler, the identifiers SYSIN and SYSPRINT are in no way reserved words. They are never recognized as special identifiers in any way. Therefore, they can be declared just as any other legal PL/I identifiers according to the normal rules for declarations. However, in the PL/I language, these identifiers are usually thought of as the standard input/output files. If the programmer desires to explicitly declare them as file names or otherwise, he should be aware of certain implications as discussed below. Note, however, that of the two identifiers SYSIN and SYSPRINT, only SYSIN can be declared as a file name because file names, being external, cannot exceed six characters in length; any attempt to declare the eight-character identifier SYSPRINT as a file name would result in an error.

For example, if one wishes to set up an ENDFILE on-unit for the standard input device, he must explicitly declare a file name and associate it with SYSIPT in the normal manner using the MEDIUM option in the ENVIRONMENT attribute. The identifier SYSIN seems a logical choice for this file name. Once SYSIN has been so declared with the proper attributes, then one may use

either GET FILE (SYSIN) EDIT... or GET EDIT... without the FILE or STRING option to refer to the standard input device SYSIPT. However, within any given program in which SYSIN has been explicitly declared and associated with SYSIPT, one should either consistently include the FILE (SYSIN) option or consistently omit the FILE (SYSIN) option in all GET statements. This is because one buffer will be set up for the explicitly declared SYSIN file, and another buffer will be set up for use with GET statements with no FILE option. Thus, although both GET statements would refer to the DOS/TOS standard input file SYSIPT, intermixed GET EDIT... and GET FILE (SYSIN) EDIT... statements would refer first to one buffer and then the other, and data items would not necessarily be transmitted in the same order in which they originally appeared in the input stream from SYSIPT.

Compatibility Note: In the OS/360 PL/I F-level compiler, a GET or PUT without a FILE or STRING option is exactly equivalent to GET FILE (SYSIN)... or PUT FILE (SYSPRINT).... Thus, if SYSIN and SYSPRINT are declared as variables other than file variables, the F-level compiler does not allow a GET or PUT without a FILE or STRING option. However, in the DOS/TOS D-level compiler, one may declare SYSIN or SYSPRINT as non-file variables and still use GET and PUT with no FILE or STRING option to refer to the standard DOS/TOS input/output devices SYSIPT and SYSLST. Therefore, a good programming practice is to use SYSIN only as a file name referring to the standard input device and to avoid the use of SYSPRINT entirely.

ENVIRONMENTAL CONSIDERATIONS FOR DATA SETS

The PL/I compiled program produced by the D-Compiler is designed to be executed under control of DOS/TOS. It provides data management facilities that control the organization, location, storage, and retrieval of data sets. The PL/I program calls upon these facilities when it is being executed. The following discussions describe the relationship between the input and output statements of a PL/I program and the various data set organizations supported by the data management facilities of DOS/TOS.

DEVICE INDEPENDENCE OF INPUT AND OUTPUT STATEMENTS

The input and output statements of a PL/I Subset program are concerned with the logical organization of a data set and not with its physical characteristics. Some of the detailed information ultimately required by a PL/I program to process a data set -- information such as input/output unit number and recording density -- need not be stated until the PL/I program is ready to be executed. Other information such as input/output device type and buffering technique is isolated in the ENVIRONMENT attribute. Device independence of this type allows changes in this information possibly without requiring changes to the PL/I program itself or at most by making changes only in the ENVIRONMENT attribute. The required information about specific input/output devices is supplied through the MEDIUM option of the ENVIRONMENT attribute. By changing this option, different input/output devices may be specified for a file. Therefore, a PL/I program can be designed without specific knowledge of the input/output devices that will be used when the program is executed. This information can then be added to the ENVIRONMENT attribute at a compilation just prior to execution.

The ENVIRONMENT Attribute

The ENVIRONMENT attribute provides information about the physical organization of the data set associated with a file. This information allows the compiler to determine the method of accessing the data set.

For the D-Compiler, the ENVIRONMENT attribute has the following general form:

```
ENVIRONMENT (option-list)
```

where "option list" is:

```
{V(maxblocksize)                    } [CONSECUTIVE]
{F(blocksize[, recordsize])         } [REGIONAL(1) ]
{U(maxblocksize)                    } [REGIONAL(3) ]

 MEDIUM (logical-device-name,
     physical-device-type)
 [LEAVE] [BUFFERS(n)] [NOLABEL] [VERIFY]
 [KEYLENGTH(decimal-integer-constant)]
```

For ease of discussion, the options of the ENVIRONMENT attribute are divided into five groups: record format, data set organization, device allocation, length of keys associated with data sets, plus a group of other options to facilitate handling of

data sets. The record format -- either V, F, or U -- must be specified. One of the data set organizations may be specified (CONSECUTIVE is applied by default if none is specified). The MEDIUM specification must always appear. All other options may or may not be given depending on the data set configuration and use. Examples of complete file declarations can be found in Chapter 13, "A PL/I Program."

Record Format

Logical records can appear in one of three formats: fixed-length (F-format), variable-length (V-format), or undefined-length (U-format). These formats provide flexibility in the design of data sets and allow the programmer to take advantage of the fixed-length and variable-length features of specific input/output devices.

The block size and record size are specified in number of bytes. For F-format records, if the record size is not specified in the ENVIRONMENT attribute, the records are assumed to be unblocked. Block size must be specified. Record size can be specified for F-format records only. Blocking and unblocking are handled automatically.

With F-format records, unblocking is dependent upon the stated record size. The block size must be evenly divisible by the record size.

With V-format records, unblocking is dependent upon information at the beginning of each block and at the beginning of each logical record. Four bytes are used at the beginning of each block to specify block length, and another four bytes are used at the beginning of each record to specify length of that record. Although insertion of this length information is done automatically by the system when the data set is created, the programmer must include the number of length-specifying bytes in determining his block size specification. When V-format data sets are created, records are always blocked if their lengths allow two or more to be placed into a block smaller than or equal to the maximum that is specified.

With U-format records, each block consists of only one record. The blocks (records) are of varying lengths. No system control bytes appear anywhere within the block. All processing of records is the responsibility of the programmer. If a length specification is included in the record, the programmer must insert it himself, and he must retrieve the information himself.

Data Set Organization

The organization of a data set determines how data is recorded in a data set volume and, once recorded, how data is subsequently retrieved so that it can be transmitted to the program. Logical records are stored in and retrieved from a data set, in either STREAM or RECORD SEQUENTIAL transmission, on the basis of successive physical positions or, in DIRECT RECORD transmission, on the basis of the values of keys specified in data transmission statements. These storage and retrieval methods provide PL/I with two general data set organizations: CONSECUTIVE and REGIONAL. CONSECUTIVE organization is assumed by default.

Each of the different data set organizations is explained in the discussions below.

CONSECUTIVE DATA SET ORGANIZATION: In a data set with CONSECUTIVE organization, the logical records are organized solely on the basis of their successive physical positions, such as they appear on magnetic tape. Such a data set does not use keys to determine the position of each record. Records are retrieved only in sequential order; therefore, the associated file must have the SEQUENTIAL attribute (or be a STREAM file). Records may be F-format, V-format, or U-format. The last two formats (V and U) may be used only for RECORD input/output and only with tape and direct access units.

Input/output devices permitted for CONSECUTIVE data sets include magnetic tape units, card readers and punches, direct-access storage units, and printers.

Later discussions will show that both stream-oriented and record-oriented transmission statements can process data sets with CONSECUTIVE organizations. However, stream-oriented statements are restricted to this type of organization; record-oriented statements are not.

After a CONSECUTIVE data set is created, it may be opened only as an INPUT or UPDATE file. Reading of such a data set may be either forwards or backwards if the data set is recorded on magnetic tape. To read the data set backwards, the associated file must be declared with the BACKWARDS attribute. If a data set is first read or written forwards and then read backwards in the same program, the LEAVE option in the ENVIRONMENT attribute must be specified to prevent the normal rewind when the file is closed or when volume switching occurs with a multi-volume data set. V-format records cannot be read backwards.

Note the difference between the CONSECUTIVE option of the ENVIRONMENT attribute and the SEQUENTIAL attribute. CONSECUTIVE specifies the physical organization of a data set; SEQUENTIAL specifies how a file is to be processed. However, in the PL/I Subset, a data set with CONSECUTIVE organization must be associated with a SEQUENTIAL file, and a data set with REGIONAL organization must be associated with a DIRECT file.

REGIONAL DATA SET ORGANIZATION: REGIONAL organization of a data set provides control of the physical placement of records in the data set. This type of control allows the programmer to optimize the record access time required by a particular application. Such optimization is not available with the CONSECUTIVE organization, in which successive records are written in strict physical sequence and which does not take advantage of the timing characteristics of direct-access storage devices. The input/output devices allowed for REGIONAL data sets are restricted to direct-access storage devices.

Record Keys: The REGIONAL data set organization allow the use of keys to identify specific records. There are two kinds of keys, recorded keys and source keys. A recorded key is a character string that actually appears in the data set, along with the record, as a positive identification of that record. It cannot exceed 255 bytes in length. A source key is a character string (or expression) that appears in a record-oriented data transmission statement to identify the record to which the statement refers.

The way keys are specified and used differs between the two different kinds of REGIONAL organization. For data sets that contain recorded keys, the source key must exactly match the recorded key in order to positively identify a record.

Whenever source keys are used in a program to access or create a data set (using the KEY or KEYFROM option), the KEYED attribute must be specified for the file. In addition, for data sets that contain recorded keys, the KEYLENGTH option in the ENVIRONMENT attribute of the associated file must be used to specify the actual length, in bytes, of the recorded key.

A data set with REGIONAL organization is divided into relative regions, each of which is identified by a region number and each of which may contain one or more records. The regions are numbered in succession, beginning with zero, and a record is accessed by specifying its region number in the source key of a record-oriented transmission statement. Two kinds of regional specifications are used, relative record and relative track. A relative record specification refers to a region of the data set by specifying the number of a particular record, relative to the first record in the data set, which is number zero. A relative track specification refers to a region of the data set by specifying the number of a particular track relative to the first track of the data set, which is track zero. A relative track or relative record specification always refers uniquely to one region in a data set.

There are two types of REGIONAL organization, one of which, REGIONAL(3), permits recorded keys to appear physically in the data set with the logical records. However, these recorded keys are never embedded within a record. When REGIONAL records are accessed by record-oriented statements, the source keys, specified in the statements, represent a region number and may also represent a recorded key.

Direct access of REGIONAL data sets employs the region number, specified in the source key, for direct access of the region. Once the region has been accessed, a sequential search may or may not be performed for a record that contains a recorded key identical to the source key. A search is performed only for REGIONAL(3), and this search extends only throughout the region (relative track) specified by the source key.

Sequential processing of REGIONAL data sets is not allowed. All REGIONAL data sets must be associated with file names that have the DIRECT attribute.

Each of the REGIONAL types is described in the following discussions.

REGIONAL(1) Organization: A data set with REGIONAL(1) organization contains unblocked F-format records that do not have recorded keys. Each region in the data set contains only one logical record; therefore, each region number represents the position of one logical record within the data set. The relative position of the first record is zero.

Since there are no recorded keys to be used for comparison, only a region number, which serves as the sole identification of a particular logical record, is meaningful in a source key. The character-string value of the source key must represent an unsigned decimal integer that does not exceed 16777215. Only the characters 0 through 9 are recognized in the source key (leading blanks of a character-string source key are not interpreted as zeros).

Thus, any source key expression must always result in a character string of length 8 containing only the digit characters 0 through 9. One good way of doing this is to declare all source keys as numeric character variables by using the PICTURE '(8)9' attribute.

REGIONAL(3) Organization: A data set with REGIONAL(3) organization contains unblocked F-format records that have recorded keys. Unlike REGIONAL(1) organization, each region in the data set corresponds to a track on the direct-access storage device, and therefore may contain more than one logical record.

The recorded key associated with each logical record is a character string, recorded in the data set and immediately preceding the record. The recorded key always includes the regional number as its rightmost eight characters. The source key (specified as a constant or some other expression) consists of a character-string value. It may be thought of as having two logical parts, the region specification and the specification of a character-string key to uniquely identify the record within the region.

The actual source key to be used is generated by evaluating the source key expression and converting it to a character string. The source key expression (which, of course, may simply be a single variable) must always result in a character string whose length precisely equals the value of the KEYLENGTH specification in the ENVIRONMENT attribute.

The rightmost eight characters of the source key make up the region specification, which states the region number. (Only the characters 0 through 9 are allowed; blanks are not interpreted as zeros.) A substring beginning at the left of the source key and containing eight less than the number of characters specified in the KEYLENGTH option is the character-string key specification. To retrieve a record, the entire source key must exactly match the recorded key of the record, since both the region specification and the character-string key specification are included in the recorded key. Note that this means that the KEYLENGTH specification must always be 9 or greater; 8 for the region specification plus at least 1 for the character-string key specification.

Consider the following source key example (b represents a blank):

```
KEY('JOHNbDOEbbb00003251')
```

The rightmost eight characters make up the region specification, the relative number of the track. (Note that leading zeros appear, since leading blanks are not treated as zero and would cause an error.) The associated file declaration should have the ENVIRONMENT option KEYLENGTH (19). Any other KEYLENGTH specification would cause the above source key expression to be padded with blanks or truncated on the right, and therefore the proper region, track 3251, would not be accessed.

In retrieving a record with the above KEY specification, the search will start at the beginning of track number 3251, and it will continue until the first record is found in that track having the recorded key of JOHNbDOEbbb00003251. If no record is found in track 3251 having this key, the KEY condition is raised.

If the above KEY option were used with an output operation, the record would be written in the first available space on track 3251. If no space were available on that track, the KEY condition would be raised.

The regional specification for REGIONAL(3) data sets cannot exceed 16777215.

Comparison of REGIONAL Types: Records in a REGIONAL data set are either "actual," representing valid data, or "dummy," representing usable areas prepared when the data set is created. Only F-format records are allowed for REGIONAL files. Dummy records are identical in REGIONAL(1) and REGIONAL(3) data sets.

Before a file can be opened to create a REGIONAL data set, the entire volume to be used must be initialized using the DOS Clear Disk utility program. This program creates dummy records, each of which contains a string filled with user-defined characters and resets the capacity record R0 to reflect that all tracks are empty. For REGIONAL(3), this resetting of R0 insures that the dummy records will not be retrieved as actual data records. (For details, see the publication IBM System/360 Disk and Tape Operating Systems, Utility Program Specifications, Form C24-3465.) Once the format of the volume has been established using this utility program, the file can then be opened and the REGIONAL data set created. The file must, of course, have the DIRECT attribute, since SEQUENTIAL is not allowed with REGIONAL data sets.

For retrieving records, a file associated with a REGIONAL data set can have either INPUT or UPDATE attributes. It must have the DIRECT attribute.

When a REGIONAL data set is associated with a file that has the UPDATE attribute, records can be retrieved, added, and replaced according to the following conventions:

1. Retrieval

   REGIONAL(1): All records, whether dummy or actual, can be retrieved.

   REGIONAL(3): Dummy records cannot be retrieved.

2. Addition

   REGIONAL(1): Addition involves the replacement of existing records, whether dummy or actual (no error condition is raised in either case).

   REGIONAL(3): Addition involves the placement of the record into the specified region.

3. Replacement

   REGIONAL(1): The specified record, whether dummy or actual, is rewritten.

   REGIONAL(3): A record with the specified key must exist. The record is rewritten.

Device Allocation

The MEDIUM option of the ENVIRONMENT attribute and, if necessary, the ASSGN statement of the DOS/TOS Job Control Language are used to associate data sets with file names. The format of the MEDIUM option is:

```
MEDIUM (logical-device-name,
        physical-device-type)
```

The logical device name is the name associated with the file that is known to the system. The physical device type defines the type of input/output device (for example, card reader, disk) which the file requires.

The logical device name is of the form SYSxxx, where xxx can be IPT (system input device), LST (system output device used for listing), PCH (system output device used for punching cards), or 000 through 222 (programmer-defined logical units). The physical device type is a four-digit number giving the device number of the input/output device to be used. For example, for the IBM 2400 Magnetic Tape Unit, the number is 2400; for the IBM 2311 Disk Unit, the number is 2311.

The logical device name is assigned before program execution to a specific physical input/output unit available to the system. This assignment may be accomplished in one of two ways. Certain standard logical device names are automatically associated with specific physical input/output units in any given DOS/TOS system configuration. (Since the automatic association of logical device names with physical input/output units is tailored to fit the needs of a particular installation, and therefore may differ from system to system, one should check the association of logical device names with physical input/output units for the DOS or TOS system he is using.) If the logical device name is not one of those automatically associated with a physical input/output unit or if the automatic association is to be changed, the job control language ASSGN statement is used to effect the assignment. Of course, the physical device type of the MEDIUM option must correspond to the physical input/output unit type assigned to the file either automatically or by using the ASSGN statement. For example, if the physical device type indicates magnetic tape, the file must be assigned to a magnetic tape unit.

Consider the following example:

```
DECLARE MASTER FILE RECORD INPUT
        SEQUENTIAL ENVIRONMENT
        (...MEDIUM(SYS006,2400)... );
```

In the above declaration, the file MASTER is assigned the logical device name SYS006 in the MEDIUM option of the ENVIRONMENT attribute. Also using the MEDIUM option, the physical device type for file MASTER is declared to be an IBM 2400 Magnetic Tape Unit.

If the DOS or TOS system in which the above file declaration is used automatically associates the name SYS006 with a suitable magnetic tape unit, no further assignment is necessary. Otherwise, the ASSGN statement from DOS/TOS job control language must be used to assign a system file to a magnetic tape unit. The job control program used with the execution module must contain the statement partially shown below:

```
// ASSGN SYS006,...
```

This statement associates the logical device name SYS006 with a physical input/output unit which must, of course, be a magnetic tape. The specific tape unit to be used follows the SYS006 on the ASSGN statement. (For the complete format of the ASSGN statement, see the publication, IBM System/360 Disk and Tape Operating Systems, PL/I Programmer's Guide, Form C24-9005.

All files in the PL/I Subset must be explicitly declared with a MEDIUM option in the ENVIRONMENT attribute, and the logical device name must be assigned to a physical input/output unit either automatically by the system or by using the DOS/TOS Job Control Language. Failure to declare the MEDIUM option properly will produce a compiler error message; failure to assign the file properly to a physical input/output unit will result in cancellation of the job at the first attempt to open the file.

Length of Keys

Keys are specified in READ, WRITE, or REWRITE statements for DIRECT files which are associated with REGIONAL data sets. For REGIONAL(1), the key specifies the region number, which is the logical record number of the record to be accessed within the data set. Thus, the key is simply an 8-digit number, in character-string form, which identifies the logical record. The length of the key for REGIONAL(1) data sets is always assumed to be 8. No KEYLENGTH option is ever specified for a REGIONAL(1) file.

For REGIONAL(3) data sets, the key specifies, in character-string form, an 8-digit number that identifies the region (relative track) where the record is to be located, preceded by a character string to uniquely identify the record within the region. The length of keys for REGIONAL(3) files must be specified using the KEYLENGTH option of the ENVIRONMENT attribute and is equal to 8 (for the 8-digit region number) plus the number of characters in the character string that identifies the record. Thus, the KEYLENGTH specification must be 9 or greater since there must be eight characters in the region specification and at least one more character for the record identification.

Other Data Set Handling Options

Data Set Positioning: The LEAVE option in the ENVIRONMENT attribute prevents the normal rewinding of magnetic-tape volumes (reels) when a data set is closed or when a reel is switched while accessing a multi-volume data set. The LEAVE option is normally employed when a data set is alternatively opened for reading or writing forwards and reading backwards.

Buffer Allocation: A buffer is an internal program-storage area that is used for intermediate storage of data transmitted to and from a data set. Allocating two buffers for a data set permits input and output activity to occur concurrently with internal processing.

The option BUFFERS(n) in the ENVIRONMENT attribute specifies the number (n) of buffers to be allocated for a data set. In the D-Compiler, n may be 1 or 2. The BUFFERS(n) option may not be used with UNBUFFERED files. If the BUFFERS(n) option is not specified, the number of buffers is assumed to be one.

Processing Unlabeled Tapes: It may be desired to read or write a magnetic tape which has no label or perhaps a non-standard label. The NOLABEL option is used in the ENVIRONMENT attribute to indicate that no label processing is to be done for the file. On output, a tape mark is automatically written as the first record on the tape. On input, a tape mark is expected as the first record on the tape. e-vices

The VERIFY Option: It may be desired, at the time a record is written, to check that the record is written correctly. The VERIFY option in the ENVIRONMENT attribute causes a read check to be performed after every write operation. This option is allowed only with files that are associated with direct-access storage devices.

DATA TRANSMISSION

As discussed earlier in this chapter, PL/I provides two types of data transmission, stream-oriented and record-oriented.

With stream-oriented transmission, a data set is considered to be a continuous stream of data items in character form; internal bit-string representations and the internal formats of coded arithmetic data do not appear in the stream. Data items are assigned from the stream to program variables or from program variables (or expressions) into the stream, with appropriate conversion from or to character form. Stream-oriented transmission statements ignore the boundaries between records.

With record-oriented transmission, a data set is treated as a collection of logical records, each of which consists of one or more data items. The data items can have any representation, internal or external, that is acceptable to the computer, and there is no data conversion. Each logical record is transmitted as a unit to or from either a program variable or a buffer.

Stream transmission uses only two input and output statements, GET and PUT, which get the next series of data items from the stream or put a specified set of data items into the stream. In record transmission, the corresponding statements are READ and WRITE, which read a logical record from the data set or write a specified logical record into the data set. Other record-transmission statements are REWRITE and LOCATE.

It is possible for the same data set to be processed at different times for either stream transmission or record transmission; however, the data set would have to be in character form acceptable for stream transmission.

One of the attributes, STREAM or RECORD, specified for the file associated with a data set determines which transmission method is applicable to the file at the time it is declared.

STREAM-ORIENTED TRANSMISSION

In the PL/I Subset language, there are two modes of stream transmission: list-directed and edit-directed. However, since edit-directed is the only mode of stream input/output presently implemented by the D-compiler, list-directed will not be explained here or mentioned elsewhere in this publication. For a complete discussion of list-directed input/output, see "PL/I Subset Language Specifications, Form C28-6809.

Edit-directed transmission uses the GET and PUT statements for input and output. These statements, in general, require the following information:

1. The name of the file associated with the data set from which data is to be obtained or to which data is to be assigned.

2. A list of program variables to which data items are to be assigned during input or from which data items are to be obtained during output. This list is called a data list. On output, the data list also can include constants and other expressions.

3. The format of each data item in the stream.

If the file name is not specified, one of the standard files is assumed.

Edit-Directed Transmission

Edit-directed transmission permits the user to specify the variables to which data is to be assigned or to specify data to be transmitted. Edit-directed transmission allows a programmer to specify the format for each item on the external medium.

Input: Data in the stream is a continuous string of characters; different data items are not separated. The variables to which the data is to be assigned is specified by a data list. Format items in a format list in the GET statement specify the number of characters to be assigned to each variable and describe characteristics of the data (for example, the assumed location of a decimal point).

Output: The data values to be transmitted are defined by a data list. The format that the data is to have in the stream is defined by a format list.

EDIT-DIRECTED DATA SPECIFICATION

General format for an edit-directed data specification, either for input or output is as follows:

```
EDIT (data-list) (format-list)
     [(data-list)(format-list)]...
```

1. The data list, which must be enclosed in parentheses, contains one or more variables that are to receive values on input or one or more expressions whose values are to be transmitted on output. Data lists are discussed in more detail in "Data Lists" below. The format list, which also must be enclosed in parentheses, contains one or more format items. There are three types of format items: data format items, which describe data in the stream; control format items, which describe page, line, and spacing operations; and remote format items, which specify the label of a separate statement that contains the format list to be used. Format lists and format

items are discussed in more detail in "Format Lists," below.

2. For input, data in the stream is considered to be a continuous string of characters not separated into individual data items. The number of characters for each data item is specified by a format item in the format list. The characters are treated according to the associated format item.

3. For output, the value of each item in the data list is converted to a format specified by the associated format item and placed in the stream in a field whose width also is specified by the format item.

4. For either input or output, the first data format item is associated with the first item in the data list, the second data format item with the second item in the data list, and so forth. If a format list contains fewer format items than there are items in the associated data list, the format list is re-used; 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. Then the sixth item in the data list will be 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. Then the sixth through the tenth format items will be ignored.

5. An array or structure variable in a list is equivalent to n data items in the data list, where n is the number of element items in the array or structure, each of which will be associated with a separate use of a data format item.

6. If a data list item is associated with a control format item, that control action is executed, and the data list item is paired with the next format item.

7. The specified transmission is complete when the last item in the data list has been processed using its corresponding format item. Subsequent format items, including control format items, are ignored.

8. On output, each data item occupies precisely the field length specified by its corresponding format item in the format list. Thus, arithmetic data items should usually be associated with format items that provide more characters than really necessary to contain the data item, so that leading blanks in each data item will separate it from other data items.

Examples:

```
GET EDIT (NAME, DATA, SALARY)
    (A (20), X(2), A(6), F(6,2));

PUT EDIT ('INVENTORY='||INUM,INVCODE))
    (A,F(5));
```

The first example specifies that the first 20 characters in the stream are to be treated as a character string and assigned to NAME; the next two characters are to be skipped; the next six are to be assigned to DATA in character format; and the next six characters are to be considered as an optionally signed decimal fixed-point constant and assigned to SALARY.

The second example specifies that the character string 'INVENTORY=' is to be concatenated with the value of character string INUM and placed in the stream in a field whose width is the length of the resultant string. Then the value of INVCODE is to be treated as an optionally signed decimal fixed-point integer constant and placed in the stream right-adjusted in a field with a width of five characters (leading characters may be blanks). Note that operational expressions and constants can appear in output data lists only.

Data Lists

Edit-directed data specifications require a data list to specify the data items to be transmitted.

General format:

```
(data-list)
```

where data list is defined as:

```
element [,element]...
```

Syntax rules:

The nature of the elements depends upon whether the data list is used for input or for output. The rules are as follows:

1. On input, a data-list element for edit-directed transmission can be one of the following: an element, array, or structure variable, a pseudo-variable that does not represent a structure or an array, or a repetitive specification (similar to a repetitive specification of a DO-group) involving any of these elements.

2. On output, a data-list element for edit-directed data specifications can be one of the following: an element expression, an array variable, a structure variable, or a repetitive specification involving any of these elements.

3. The elements of a data list must be of arithmetic or string data type.

4. As shown in the general format, a data list must always be enclosed in parentheses.

Repetitive Specification

The general format of a repetitive specification is shown in Figure 8-1.

Syntax rules:

1. An element in the element list of the repetitive specification can be any of those allowed as data-list elements as listed above.

2. The expressions in the specification, which are the same as those in a DO statement, are described as follows:

   a. Each expression in the specification is an element expression.

   b. In the specification, expression-1 represents the starting value of the control variable. Expression-3 represents the increment to be added to the control variable after each repetition of data-list elements in the repetitive specification. Expression-2 represents the terminating value of the control variable. Expression-4 represents a second condition to control the number of repetitions. The exact meaning of the specification is identical to that of a DO statement with the same specification. When the last specification is completed, control passes to the next element in the data list.

3. Each repetitive specification must be enclosed in parentheses as shown in the general format. Note that if a repetitive specification is the only element in a data list, two sets of outer parentheses are required, since the data list must have one set of parentheses and the repetitive specification must have a separate set.

4. As Figure 8-1 shows, the "specification" portion of a repetitive specification can be repeated a number of times, as in the following form:

```
DO I = 1 TO 4, 6 TO 10
```

   Repetitive specifications can be nested; that is, an element in a repetitive specification can itself be a repetitive specification. Each DO portion must be delimited on the right with a right parenthesis (with its matching left parenthesis added to the beginning of the entire repetitive specification).

   When DO portions are nested, the rightmost DO is at the outer level of nesting. For example, consider the following statement:

```
GET EDIT (((A(I,J) DO I = 1 TO 2)
     DO J = 3 TO 4)) (format-list);
```

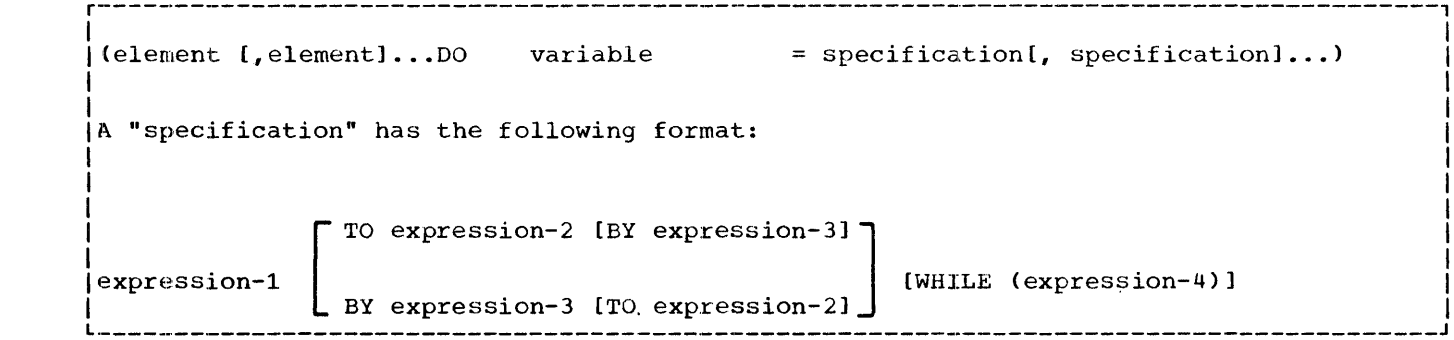


Figure 8-1. General Format for Repetitive Specifications

Note the three sets of parentheses, in addition to the set used to delimit the subscript. The outermost set is the set required by the data list; the next is that required by the outer repetitive specification. The third set of parentheses is that required by the inner repetitive specification. This statement is equivalent to the following nested DO-groups:

```
DO J = 3 TO 4;
   DO I = 1 TO 2;
   GET EDIT (A (I,J))
   (format-list);
   END;
END;
```

It gives values to the elements of the array A in the following order:

A(1,3), A(2,3), A(1,4), A(2,4)

Note: Although the DO keyword is used in the repetitive specification, a corresponding END statement is not allowed.

Transmission of Data-List Elements

If a data-list element is an array variable, 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 element is a structure variable, the elements of the structure are transmitted in the order specified in the structure declaration.

For example, if a declaration is:

```
DECLARE 1 A, 2 B(10), 2 C(10);
```

and if X is a file, then the statement:

```
PUT FILE (X) EDIT (A) (format-list);
```

would result in the output being 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, within a data list used in an input statement for edit-directed transmission, a variable is assigned a value, this new value is used if the variable appears in a later reference in the data list. For example:

```
GET EDIT (N,(X(I) DO I=1 TO N), J,
   SUBSTR (NAME, J,3)) (format-list);
```

When this statement is executed, data is 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 used to specify the number of items to be assigned.
3. A new value is assigned to J.
4. A substring of length 3 is assigned to the string variable NAME, beginning at the Jth character.

Format Lists

Each edit-directed data specification requires its own format list.

General format:

```
(format-list)
```

where format list is defined as:

```
{ item              }[, item              ]
{ n item            }[, n item            ] ...
{ n (format-list)   }[, n (format-list)   ]
```

Syntax rules:

1. Each "item" represents a format item as described below.
2. The letter n represents an iteration factor, which must be an unsigned decimal integer constant. A blank must separate the constant and the following format item. The iteration factor specifies that the associated format item or format list is to be used n successive times. The associated format item is that item or list of items immediately to the right of the iteration factor.

General rule:

There are three types of format items: data format items, control format items, and the remote format item. Data format items specify the external forms that data fields are to take. Control format items specify, for PRINT files, the page, line, column, and spacing operations. The spacing format item can also be used with non-PRINT files, either input or output. The remote format item allows format items to be specified in a separate FORMAT statement elsewhere in the block.

Detailed discussions of the various types of format items appear in Part II, Section E, "Edit-Directed Format Items." The following discussions show how the format items are used in edit-directed data specifications.

Data Format Items

On input, each data format item specifies the number of characters to be associated with the data item and how to interpret the external data. The data item is assigned to the associated variable named in the data list, with necessary conversion to conform to the attributes of the variable. On output, the value of the associated element in the data list is converted to the character representation specified by the format item and is inserted into the data stream.

There are four data format items: fixed-point (F), floating-point (E), character-string (A), and bit (B). They are specified as follows:

F (w[,d[,p]])

E (w,d[,s])

A [(w)]

B [(w)]

In this list, the letter w represents a decimal integer constant that specifies the number of characters in the field. The letter d specifies the number of digits to the right of a decimal point.

A third specification (p) is allowed in the F format item; it is a scaling factor. A third specification (s) is allowed in the E format item to specify the number of digits that must be maintained in the first subfield of the floating-point number. These specifications are discussed in detail in Part II, Section E, "Edit-Directed Format Items."

Note: Fixed-point binary and floating-point binary data items must always be represented in the input stream with their values expressed in decimal digits. The F and E format items then are used to access them, and the values will be converted to binary representation upon assignment. On output, binary items are converted to decimal values and the associated F or E format items must state the field width in terms of the converted decimal number.

The following examples illustrate the use of format items:

1. GET FILE (INFILE) EDIT (ITEM) (A(20));

   This statement causes the next 20 characters in the file called INFILE to be assigned to ITEM, which must be a character-string variable. If it is not a character-string variable, an error results.

   Note: If the data list and format list were used for output, the length of a string item need not be specified in the format item if the field width is to be the same as the length of the string, that is, if no blanks are to follow the string or if no truncation is to occur.

2. PUT FILE (MASKFL) EDIT (MASK) (B);

   Assume MASK has the attributes BIT (25); then the above statement writes the value of MASK in the file called MASKFL as a string of 25 characters consisting of 0's and 1's. A field width specification can be given in the B format item. It must be stated for input. Note that MASK must be a bit-string variable; if it is not, an error results.

3. PUT EDIT (TOTAL) (F(6,2));

   Assume TOTAL has the attributes FIXED (4,2); then the above statement specifies that the value of TOTAL is to be converted to the character representation of a fixed-point number and written into the standard output file. A decimal point is to be inserted before the last two numeric characters, and the number will be right-adjusted in a field of six characters. Leading zeros will be changed to blanks, and, if necessary, a minus sign will be placed to the left of the first numeric character. If a decimal point or a minus sign appears, either will cause one less leading blank to appear. Consequently, the F(6,2) specification will always allow all digits, the point, and a possible sign to appear.

4. GET FILE(A) EDIT (ESTIMATE) (E(10,6));

   This statement obtains the next ten characters from the file called A and interprets them as a floating-point decimal number. A decimal point is assumed before the rightmost six digits of the mantissa. An actual point within the data can override this assumption. The value of the number is converted to the attributes of ESTIMATE and assigned to this variable.

5. GET EDIT (NAME, TOTAL) (A(5),F(4,0));

When this statement is executed, the standard input file is assumed. The first five characters are assigned to NAME. The next four characters must be arithmetic characters with possible leading and/or trailing blanks, and they are assigned to TOTAL.

Control Format Items

Control format items consist of two types: the spacing format item (X) and the printing format items (COLUMN, LINE, PAGE, and SKIP). The spacing format item specifies relative spacing in the data stream. The printing format items can be used only with PRINT files and, consequently, can appear only in PUT statements. All but PAGE generally include decimal integer constants. LINE, PAGE, and SKIP also can appear separately as options in the PUT statement. When they appear as options in a PUT, expressions can be used in place of the decimal integer constants.

The following examples illustrate the use of the control format items:

1.
```
GET EDIT (NUMBER, REBATE)
         (A(5), X(5), A(5));
```

   This statement treats the next 15 characters from the standard input file in the following way: the first five characters are assigned to NUMBER, the next five characters are spaced over and ignored, and the remaining five characters are assigned to REBATE.

2.
```
PUT FILE(OUT) EDIT (PART, COUNT)
          (A(4), X(2), F(5));
```

   This statement places in the file named OUT four characters that represent the value of PART, then two blank characters, and finally five characters that represent the integer value of COUNT.

3. The following examples show the use of the printing format items in combination with one other.

```
PUT EDIT ('QUARTERLY STATEMENT')
    (PAGE, LINE(2), A(19));

PUT EDIT (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));
```

   The first PUT statement specifies that the heading QUARTERLY STATEMENT is to be written on line two of a new page in the standard system output file. The second statement specifies that two lines are to be skipped (that is, "skip to the third following line") and the value of ACCT# is to be written, beginning at the first character of the fifth line; the value of BOUGHT, beginning at character position 14; the value of SOLD, beginning at character position 30; the value of PAYMENT, beginning at character position 45; and the value of BALANCE at character position 60.

Note: Control format items are executed at the time they are encountered in the format list. Any control format list that appears after the data list is exhausted will have no effect.

Remote Format Item

The remote format item (R) specifies the label of a FORMAT statement (or a label variable whose value is the label of a FORMAT statement) located elsewhere; the FORMAT statement and the GET or PUT statement specifying the remote format item must be internal to the same block. The FORMAT statement contains the remotely situated format items. This facility permits the choice of different format specifications at execution time, as illustrated by the following example:

```
DECLARE SWITCH LABEL;
GET FILE(IN) EDIT(CODE)(F(1));
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 to be a label variable; the second GET statement can be made to operate with either of the two FORMAT statements. Another advantage of the remote format item is that it allows many GET/PUT statements to share the same format.

STREAM-ORIENTED DATA TRANSMISSION STATEMENTS

The following provides a summary of the STREAM data transmission statements, along with their options, according to file attributes (the statements are discussed individually in detail in Part II, Section J, "Statements").

STREAM INPUT:

```
GET [FILE (file-name)]
    data-specification;
```

STREAM OUTPUT:

```
PUT [FILE (file-name)]
    data-specification;
```

STREAM OUTPUT PRINT:

```
PUT  [FILE  (file-name)]

   [PAGE [LINE(expression)]]
   [SKIP [(expression)]    ]
   [LINE  (expression)     ]

   [data-specification];
```

Note: The "data specification" can be omitted for STREAM OUTPUT PRINT files only if one of the control options appears.

In all of the above, the data specification has the following form:

```
EDIT (data-list) (format-list)
    [(data-list) (format-list)]..
```

Format lists may use any of the following format items:

A,B,E,F,R,X	which may be used with any STREAM file
PAGE SKIP [(w)] LINE (w) COLUMN (w)	which may be used only with STREAM OUTPUT PRINT files

RECORD-ORIENTED TRANSMISSION

Data sets that contain discrete records or which are to be created as collections of discrete records may be manipulated with record-oriented operation statements. These statements are READ, WRITE, REWRITE, and LOCATE. A general description of these statements is contained in this chapter; they are described completely in Part II, Section J, "Statements." Each record obtained from a data set or dispatched to a data set is defined in terms of the data attributes of a variable (usually a structure). For input operations, the record is obtained from the data set and assigned, without conversion, to the variable. For output operations, the data is transmitted without conversion into the data set.

The variables involved in record transmission must be unsubscripted, of level 1 (element variables and array variables are of level 1 by default), and may be of any storage class. The variables cannot be parameters or defined variables. They may be label or pointer variables, but such data may lose its validity in transmission.

With RECORD transmission, it is possible to operate upon the record in a buffer if the file has the BUFFERED attribute. Operation within the buffer can be accomplished through the use of a based variable, which describes the data attributes of the record, and a pointer variable, which can be set to different values to identify the location of the based variable within the buffer. A based variable and its associated pointer variable are declared with the BASED storage class attribute in the following form:

```
BASED (pointer-variable)
```

The pointer variable itself cannot have the BASED storage class attribute; the default is AUTOMATIC. The pointer variable may be given either INTERNAL or EXTERNAL scope attribute, with default being INTERNAL; but the scope of the based variable is always INTERNAL. The pointer variable must be explicitly declared with the POINTER attribute.

Consider the following declarations:

```
DECLARE REC_ID POINTER;
DECLARE 1  MASTER_RECORD BASED
               (REC_ID),
           2 IDENTIFICATION CHARACTER(10),
           2 NAME CHARACTER(30),
           2 ADDRESS,
             3 STREET CHARACTER(15),
             3 CITY CHARACTER(15),
             3 STATE CHARACTER(15),
             3 ZIP CHARACTER(5);
```

The name MASTER_RECORD is a based variable that can be used to describe a record located in a buffer. Fields of the record must conform to the attributes declared for MASTER_RECORD. REC_ID is a pointer variable that identifies the position of MASTER_RECORD within the buffer. The pointer variable is declared explicitly.

If any attributes other than AUTOMATIC are to be declared for a pointer variable, they must be explicitly declared. For example, the following declaration specifies the STATIC and EXTERNAL attributes for the pointer variable REC_ID:

```
DECLARE REC_ID POINTER STATIC
   EXTERNAL;
```

For input/output operations specifying based variables, the pointer value is set by the SET option in the READ or LOCATE statements.

RECORD-ORIENTED DATA TRANSMISSION STATEMENTS

There are three statements that actually cause transmission of records to or from external storage. They are READ, WRITE, and REWRITE. A fourth statement, LOCATE, causes storage to be allocated in a buffer for subsequent transmission. The attributes of the file determine which statements can be used.

The READ statement can be used with any INPUT or UPDATE file. It causes a record to be transmitted from the data set to the program, either directly to a variable or to a buffer. In the case of blocked records, the READ statement causes a logical record to be transferred from a buffer to the variable; or if the SET option is used, it causes the value of a pointer to be set to point to the logical record in a buffer. For blocked records, consequently, every READ statement may not cause physical input.

The WRITE statement can be used with any OUTPUT file, and with DIRECT UPDATE, but not with SEQUENTIAL UPDATE. It causes a record to be transmitted from the program to the data set. For unblocked records, the transmission may be directly from a variable or from a buffer. For blocked records, the WRITE statement causes a logical record to be placed into a buffer. Only when the blocking of the record is complete is there actual physical output.

The REWRITE statement causes a record to be replaced in an UPDATE file. For SEQUENTIAL UPDATE files, the REWRITE statement specifies that the last record read from the file is to be rewritten; consequently a record must be read before it can be rewritten. For DIRECT UPDATE files, the REWRITE statement must specify a key; consequently, any record can be rewritten whether or not it has first been read.

The LOCATE statement specifies that a based variable be allocated in an output buffer for the specified file and that a pointer be set to identify the location. Both a based variable and a pointer variable must be specified in the LOCATE statement. The based variable is used, in the case of variable length records, to determine the length of the record. The LOCATE statement never specifies immediate data transmission; the contents of the buffer are undefined. Values must be assigned to the based variable. The record will not be written until the next WRITE, LOCATE, or CLOSE statement is executed for the same file. In the case of blocked records, a subsequent LOCATE statement may only cause a pointer to be set to identify a location immediately following the previous record in the buffer.

Options of Record-Oriented Transmission Statements

Options that are allowed for record-oriented data transmission statements differ according to the attributes of the associated file and the purpose of the statement. A list of all of the allowed combinations for each type of file is given later in this chapter.

Each option consists of a keyword followed by a value, which is a file name, a variable, or an expression. This value always must be enclosed in parentheses. In any statement, the FILE option must appear first.

The FILE Option

The FILE option (also called the FILE specification) must appear in every record-oriented statement. It specifies the name of the file upon which the operation is to take place. It consists of the keyword FILE followed by the file name enclosed in parentheses. An example of the FILE option is shown in each of the statements in this section.

The INTO Option

The INTO option can be used in the READ statement for any type of INPUT or UPDATE file. The INTO option specifies a variable to which the logical record is to be assigned. The form is the same whether or not the record passes through an intermediate buffer. The variable can be a based variable.

```
READ FILE (DETAIL) INTO (RECORD_1);
```

This specifies that the next sequential record is to be assigned to the variable RECORD_1.

The SET Option

The SET option can be used in the READ statement for SEQUENTIAL BUFFERED INPUT or UPDATE files. It must appear in every LOCATE statement. The SET option specifies a pointer variable that is to point to the logical record in a buffer.

```
READ FILE (MASTER) SET (REC_IDENT);

LOCATE PAY_REC FILE (PAYROLL)
   SET (P);
```

The first example specifies that the next record from the file MASTER is to be read and that the pointer variable REC_IDENT is to be set to point to that location in the buffer. If the logical record is part of a blocked record, and is not the first record in the block, the actual result of the statement will be merely to set the value of the pointer. The value of REC_IDENT must be associated with a based variable, so that the fields of the record can be accessed.

The second example specifies that the based variable PAY_REC is to be allocated in a buffer and that its location is to be assigned to the pointer variable P, which must have been declared with the based variable PAY_REC. The LOCATE statement must always specify a based variable. Following allocation of the based variable, values must be assigned to it. The record is written when the next WRITE, LOCATE, or CLOSE statement is executed for the file PAYROLL. If the record PAY_REC is part of a blocked record, the next LOCATE statement may only allocate the next logical record in the same block.

The FROM Option

The FROM option must be used in the WRITE statement for any OUTPUT file and for a DIRECT UPDATE file. It also can be used in the REWRITE statement for any UPDATE file. The FROM option specifies the variable from which the record is to be written.

```
WRITE FILE (MASTER) FROM (MAS_REC);

REWRITE FILE (MASTER) FROM (MAS_REC);
```

Both statements specify that the value of the variable MAS_REC is to be written into the file MASTER. In the case of the WRITE statement, it specifies a new record in a SEQUENTIAL OUTPUT file.

The REWRITE statement specifies that MAS_REC is to replace the last record read from a SEQUENTIAL UPDATE file.

The KEY Option

The KEY option applies only to files associated with data sets of REGIONAL organization. It must be used in the READ statement for DIRECT files with the INPUT or UPDATE attribute. The KEY option also must be used in the REWRITE statement for DIRECT UPDATE files. Any file for which the KEY option is used must also have the KEYED attribute.

The KEY option consists of the keyword KEY followed by a parenthesized expression, which is a source key that identifies a particular record. The expression must represent a character string of eight digits for REGIONAL (1) and of length specified by KEYLENGTH for REGIONAL(3).

Following is a summary of what the character string is and what it represents for each of the data set organizations to which it is applicable:

REGIONAL (1)	A string of eight digits that specify the relative record number of the desired record.
REGIONAL (3)	A string of characters, the rightmost eight of which must consist of digits. These rightmost eight characters specify a relative track that is the region to be searched. The record to be accessed is identified by a recorded key that exactly matches the source key which has been converted to a character string of the length specified by KEYLENGTH. This string always includes the rightmost eight characters, which identify the region.

The expression in the KEY option must result in a valid key.

```
READ FILE (MASTER) INTO (MAS_REC) KEY
   ('00003253')

READ FILE (FILEX) INTO (ORDER_REC) KEY
   (NAME||AREA#);
```

The first statement specifies that record number 3253 in the REGIONAL (1) data set associated with the file MASTER is to be read and assigned to the variable MAS_REC.

The second statement, which would be appropriate for a REGIONAL (3) data set, specifies that a record is to be read from the file FILEX into the variable ORDER_REC. The record is to be found in a region

identified by the value of AREA#; the specific record is to be recognized by a recorded key of length specified by KEYLENGTH that matches the character string specified by the expression in the KEY option. Note that the variable AREA# must represent a character-string of eight digits.

The KEYFROM Option

The KEYFROM option must be specified in a WRITE statement used to write a REGIONAL data set. It cannot be used with CONSECUTIVE data set organization. Therefore, it can appear in a WRITE statement only for a DIRECT OUTPUT or DIRECT UPDATE file. Any file for which the KEYFROM option is specified must have the KEYED attribute.

The KEYFROM option specifies the location, within the data set, where the record is to be written. For REGIONAL(1) data sets, it specifies only the region number. For REGIONAL(3) data sets, it specifies a character string to be written as a recorded key (in which the rightmost eight characters represent the region number). It is written with the keyword KEYFROM followed by a parenthesized expression. The expression can be a constant, a variable, or any other expression that can be converted to a character string. For REGIONAL(3), the KEYLENGTH option of the ENVIRONMENT attribute must specify the length of the recorded key to be written.

```
WRITE FILE (PAYROLL) FROM (PAY_REC)
   KEYFROM (NAME||TRACK_NO);
```

The above statement, which could be appropriate for a REGIONAL (3) data set, specifies that the value of PAY_REC is to be written as the next sequential record in the specified region of PAYROLL. The value of TRACK_NO specifies the region in which the record is to be written. The source key is to be a concatenation of the value of NAME and the value of TRACK_NO, and is to be written as the recorded key.

Record-Oriented Transmission Statement Formats

This section provides a summary of the allowed RECORD transmission statements, along with their options, according to file attributes.

SEQUENTIAL BUFFERED INPUT:

```
READ FILE (file-name)
   INTO (variable);

READ FILE (file-name)
   SET (pointer-variable);
```

SEQUENTIAL BUFFERED OUTPUT:

```
WRITE FILE (file-name)
   FROM (variable);

LOCATE variable FILE (file-name)
   SET (pointer-variable);
```

SEQUENTIAL BUFFERED UPDATE:

```
READ FILE (file-name)
   INTO (variable);

READ FILE (file-name)
   SET (pointer-variable);

REWRITE FILE (file-name);

REWRITE FILE (file-name)
   FROM (variable);
```

SEQUENTIAL UNBUFFERED INPUT:

```
READ FILE (file-name)
   INTO (variable);
```

SEQUENTIAL UNBUFFERED OUTPUT:

```
WRITE FILE (file-name)
   FROM (variable);
```

SEQUENTIAL UNBUFFERED UPDATE:

```
READ FILE (file-name)
   INTO (variable);

REWRITE FILE (file-name)
   FROM (variable);
```

DIRECT INPUT:

```
READ FILE (file-name)
   INTO (variable)
   KEY (expression);
```

DIRECT OUTPUT:

```
WRITE FILE (file-name)
   FROM (variable)
   KEYFROM (expression);
```

DIRECT UPDATE:

```
READ FILE (file-name)
   INTO (variable)
   KEY (expression);

REWRITE FILE (file-name)
   FROM (variable)
   KEY (expression);

WRITE FILE (file-name)
   FROM (variable)
   KEYFROM (expression);
```

Summary of Record-Oriented Transmission

The following points cover the salient environmental factors in the use of RECORD transmission:

1. A SEQUENTIAL file specifies that the accessing, creation, or modification of the data set records is performed in a particular order,that is, from the first record of the data set to the last record of the data set (or from the last to the first if the BACKWARDS attribute has been specified).

2. A DIRECT file specifies that the accessing, creation, or modification of the data set records may be performed in random order. The particular record of the data set to be operated upon is identified by a specified key.

3. A data set that is accessed, created, or modified in the SEQUENTIAL access method may not have recorded keys.

4. Existing records of a data set in a SEQUENTIAL UPDATE file can be modified and rewritten, but the number of records cannot be increased. Operation with a DIRECT UPDATE file, however, may specify that records are to be added to the data set, through use of the WRITE statement. An existing record in an UPDATE file can be replaced through use of a REWRITE statement.

5. If the READ INTO option is used in referring to a SEQUENTIAL BUFFERED UPDATE file and the next REWRITE statement does not make use of a FROM option, the record in the data set is replaced from the buffer and not from the variable that had been specified in the INTO option of the READ statement. The FROM option in a REWRITE statement must specifically name the variable into which the data has been read if that data is to be rewritten.

6. A WRITE statement adds a record to a data set, while a REWRITE statement replaces a record. Thus, a WRITE statement may be used with OUTPUT files, and DIRECT UPDATE files, but a REWRITE statement may be used with UPDATE files only. Moreover, for DIRECT files, a REWRITE statement uses the KEY option to identify the existing record to be replaced; a WRITE statement uses the KEYFROM option, which not only specifies where the record is to be written in the data set, but also specifies, except for REGIONAL (1), an identifying key to be recorded in the data set.

The data manipulation performed by the arithmetic, comparison, and bit-string operators are extended in PL/I by a variety of string handling and editing features. These features are specified by data attributes, statement options, built-in functions, and pseudo-variables.

The following discussions give general descriptions of each feature, along with illustrative examples.

EDITING BY ASSIGNMENT

The most fundamental form of editing performed by the assignment statement involves converting the data type of the value on the right side of the assignment symbol to conform to the attributes of the receiving variable. Because the assigned value is made to conform to the attributes of the receiving field, the precision or length of the assigned value may be altered. Such alteration can involve the addition of digits or characters to and the deletion of digits or characters from the converted item. The rules for data conversion are discussed in Chapter 4, "Expressions," and in Part II, Section F, "Data Conversion."

ALTERING THE LENGTH OF STRING DATA

When a value is assigned to a string variable, it is converted, if necessary, to the same string type (character or bit) as the receiving string and also, if necessary, is truncated or extended on the right to conform to the declared length of the receiving string. For example, assume SUBJECT has the attributes CHARACTER (10), indicating a character string of ten characters. Consider the following statement:

```
SUBJECT='TRANSFORMATIONS';
```

The length of the string on the right is fifteen characters; therefore, five characters will be truncated from the right end of the string when it is assigned to SUBJECT. This is equivalent to executing:

```
SUBJECT='TRANSFORMA';
```

If the assigned string is shorter than the length declared for the receiving string variable, the assigned string is extended on the right either with blank characters, in the case of a character-string variable, or with zero bits, in the case of a bit-string variable. Assume SUBJECT still has the attributes CHARACTER (10). Then the following two statements assign equivalent values to SUBJECT:

```
SUBJECT='PHYSICS';

SUBJECT='PHYSICSbbb';
```

The letter b indicates a blank character.

Let CODE be a bit-string variable with the attributes BIT(10). Then the following two statements assign equivalent values to CODE:

```
CODE='110011'B;

CODE='1100110000'B;
```

Note, however, that the following statements do not assign equivalent values to SUBJECT if it has the attributes CHARACTER (10):

```
SUBJECT='110011'B;

SUBJECT='1100110000'B;
```

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

```
SUBJECT='110011bbbb';
```

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

```
SUBJECT='1100110000';
```

OTHER FORMS OF ASSIGNMENT

In addition to the assignment statement, PL/I provides other ways of assigning values to variables. Among these are two methods that involve input and output statements, one in which actual input and output operations are performed, and one in which data movement is entirely internal.

Input and Output Operations

Although the assignment statement is concerned with the transmission of data between storage locations internal to a computer, input and output operations can also be treated as related forms of assignment in which transmission occurs between the internal and external storage facilities of the computer.

Record-oriented operations, however, do not cause any data conversion of items in a logical record when it is transmitted. Required editing of the record must be performed within internal storage either before the record is written or after it is read.

Stream-oriented operations, on the other hand, do provide a variety of editing functions that are applied when data items are read or written. These editing functions are similar to those provided by the assignment statement, except that any data conversion always involves character type, conversion from character type on input, and conversion to character type on output.

The STRING Option in GET and PUT Statements

The STRING option in GET and PUT statements allows the statements to be used to transmit data between internal storage locations rather than between the internal and external storage facilities. In GET and PUT statements, the FILE option, specified by FILE (file-name), is replaced by the STRING option, as shown in the following formats:

```
GET STRING (character-string-variable)
    data-specification;

PUT STRING (character-string-variable)
    data-specification;
```

The GET statement specifies that data items to be assigned to variables in the data list are to be obtained from the specified character-string variable. The PUT statement specifies that data items of the data list are to be assigned to the specified character-string variable. The "data specification" is the same as described for input and output. In general, it takes the following form:

```
EDIT (data-list) (format-list)
```

The STRING option is used with edit-directed transmission, which considers the input stream to be a continuous string of characters. This option permits data gathering or scattering operations to be performed with a single statement, and it allows stream-oriented processing of character strings that are transmitted by record-oriented statements.

Consider the following statement:

```
PUT STRING (RECORD) EDIT
   (NAME, PAY#, HOURS*RATE)
   (A(12), A(7), F(8));
```

This statement specifies that the character-string value of NAME is to be assigned to the first (leftmost) 12 character positions of the string named RECORD, and that the character-string value of PAY# is to be assigned to the next seven character positions of RECORD. The value of HOURS is then to be multiplied by the value of RATE, and the product is to be handled like F-format output and assigned to the next eight character positions of RECORD.

Frequently, it is necessary to read records of different formats, each of which gives an indication of its format within the record by the value of a data item. The STRING option provides an easy way to handle such records; for example:

```
READ FILE (INPTR) INTO (TEMP);
GET STRING (TEMP) EDIT (CODE) (F(1));
IF CODE ¬=1 THEN GO TO OTHER_TYPE;
GET STRING (TEMP) EDIT (X,Y,Z)
   (X(1), 3 F(10,4));
```

The READ statement reads a record from the input file INPTR. The first GET statement uses the STRING option to extract the code from the first byte of the record and to assign it to CODE under the control of F-format input. The code is tested to determine the format of the record. If the code is 1, the second GET statement then uses the STRING option to assign the items in the record to X,Y, and Z. Note that the second GET statement specifies that the first character in the string TEMP is to be ignored (the X(1) format item in the format list). Each GET statement with the STRING option always specifies that the scanning is to begin at the first character of the string. Thus, the character that is ignored in the second GET statement is the same character that is assigned to CODE by the first GET statement.

In a similar way, the PUT statement with a STRING option can be used to create a record within internal storage. In the following example, assume that the file OUTPRT is eventually to be printed.

```
PUT STRING (RECORD) EDIT
   (NAME, PAY#, HOURS*RATE)
   (X(1), A(12), X(10),
   A(7), X(10), F(8));

WRITE FILE (OUTPRT) FROM (RECORD);
```

The X(1) in the format list of the PUT statement specifies that the first character assigned to the character-string variable RECORD is to be a single blank. Following that, the values of the variables NAME and PAY# and of the expression HOURS*RATE are assigned. The format list specifies that ten blank characters are to be inserted between NAME and PAY# and between PAY# and the expression value. The WRIIE statement is used to write the record into the file OUTPRT.

THE PICTURE SPECIFICATION

The editing capabilities associated with data assignment, namely, conversion to a specified data type with accompanying truncation and/or padding, can be extended by use of the picture specification. A picture specification consists of a sequence of character codes (picture characters) that specify editing operations tc be performed on numeric character values. (A detailed discussion of each picture character, together with examples of its use, appears in Part II, Section D, "Picture Specification Characters." The following discussions are concerned with general principles that govern the use of the picture specification.)

A picture specification can be used to describe ordinary character-string data, or it can be used to describe numeric character data, which is data that represents a numeric value.

A picture specification is always enclosed in quotation marks and is used with a PICTURE attribute in a DECLARE statement:

```
DECLARE CODE PICTURE 'XXXXX';

DECLARE PAYMT PICTURE '$999V.99';
```

Character-String Picture Specifications

A character-string picture specification describes a character string; the number of picture characters in the specification determines the length of the string (only the X picture character can be used in a character-string picture specification). For example, the PICTURE attribute in the first DECLARE statement above describes CODE as a character string of length five and is equivalent to the attribute CHARACTER (5). The picture character X also specifies that any character recognized by the computer can occur in the corresponding position of the character string.

Any value assigned to CODE will be converted, if necessary and possible, to a character string and will be truncated or extended on the right as required, to meet the five-character length of CODE. Consider the following examples:

```
CODE='A2B9C8';

CODE='4F';
```

In the first assignment, one character is truncated from the right end of the assigned character string. In the second assignment, three blank characters are appended to the right end of the assigned character string.

Numeric Character Picture Specifications

In addition to the picture character 9, numeric character specifications can contain other picture characters that are used to edit numeric character data. (The picture character X cannot appear in a numeric character picture specification.) The general functions performed by these additional picture characters are described in "Editing Numeric Character Data" below.

Assignment to character-string variables is always from left to right; padding and truncation are on the right. Assignment to a numeric character variable, however, depends upon the location of an assumed decimal point (specified by the picture character V). Values assigned to numeric character variables are always point aligned.

Values of Numeric Character Variables

The value of a numeric character variable can be interpreted in two ways, either as an arithmetic value or as a character-string value.

For a numeric character variable described with a picture specification that contains only one or more occurrences of the character 9, the arithmetic value is the value expressed by the character string, that is, a decimal integer.

If, however, editing characters are included in the picture specification, the arithmetic value and the character-string value generally would be different. Editing characters are actually stored internally in the specified positions of the data item. The editing characters then are considered to be part of the character-string value of the variable. The editing characters are not, however, a part of the variable's arithmetic value, which involves only decimal digits, the assumed location of a decimal point, and a sign (if one is present).

If the value of a numeric character variable is assigned to another numeric character variable or to a coded arithmetic variable, only the arithmetic value is assigned. In the assignment to a coded arithmetic variable (or in the appearance of a numeric character variable in an arithmetic expression operation), conversion to coded arithmetic is performed.

If the value of a numeric character variable is assigned to a character-string variable, no actual conversion is necessary, and any specified editing characters are included in the assignment.

An ordinary character-string variable (specified with the CHARACTER attribute) can be defined on a numeric character variable, using the DEFINED attribute specification. Any reference to the character-string variable is a reference to the character-string value of the numeric character variable. For example:

```
DECLARE A PICTURE '$999V.99',
        B CHARACTER(7) DEFINED A,
        C DECIMAL FIXED (5,2);

A = 128.76;

C = A;
```

After the constant is assigned to A, its arithmetic value is 128.76. This is the value that is assigned to C (after conversion to internal coded arithmetic). The character-string value of A, however, is $128.76; if it were assigned to a character-string variable with a length of 7 or greater, this is the value that would be assigned. The same value, $128.76, is the value of B, since a character string defined on a numeric character variable represents the character-string value of the numeric character variable.

No arithmetic variable (except another numeric character variable) can be defined on a numeric character variable without causing an error.

Editing Numeric Character Data

Because the picture specification of a numeric character field cannot contain the character X, the value of a numeric character data item can always be given a numeric interpretation. Consider the following declaration:

```
DECLARE COUNT PICTURE '99999';
```

Although COUNT is a string of five characters, it can only contain numeric digits; therefore, it is a numeric character variable whose value can be interpreted as a five-digit unsigned fixed-point decimal integer. Unless specified otherwise (with the picture character V), a decimal point is always assumed to be at the right end of a numeric character data item. For example, let COUNT, as declared above, appear in the following assignment statement:

```
COUNT=123.45;
```

When the assignment is performed, the decimal point of the constant is aligned on the assumed point declared for the numeric character variable, and the two rightmost digits are truncated. Two zero digits are then appended on the left end. The effect of the above assignment therefore, is equivalent to the following statement:

```
COUNT=00123;
```

The picture character V allows an assumed decimal point to be specified anywhere in a numeric data item, and not just at the right end:

```
DECLARE TOTAL PICTURE '999V99';
```

Here the value of TOTAL is interpreted as a string of five characters representing a five-digit unsigned fixed-point decimal number with two fractional places. The decimal point of a value assigned to TOTAL will be aligned between the third and fourth digits as specified by the picture character V. Consequently, the following two assignment statements are equivalent:

```
TOTAL=123;

TOTAL=123.00;
```

Note, however, that TOTAL contains only five characters. The picture character V does not specify an actual character position in the numeric character field; it is used only to align decimal points. And if TOTAL were converted to a character string and then printed, no decimal point would appear in the printed field; its character-string value does not include a decimal point.

A decimal point picture character(.) can appear in a numeric picture specification. It merely indicates that a point is to be included in the character representation of the value. Therefore, the decimal point is part of its character-string string value. The decimal point picture character does not cause decimal point alignment during assignment since it is not a part of the variable's arithmetic value. Only the character V causes alignment of decimal points. For example:

```
DECLARE SUM PICTURE '999V.99';
```

SUM is a numeric character variable representing numbers of five digits with a decimal point assumed between the third and fourth 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-string value. (The decimal point picture character can appear on either side of the character V. See Part II, Section D, "Picture Specification Characters.") The following two statements assign the same value to SUM:

```
SUM=123;

SUM=123.00;
```

In the first statement, two zero digits are added to the right of the digits 123.

Note the effect of the following declaration:

```
DECLARE RATE PICTURE '9V99.99';
```

Let RATE be used as follows:

```
RATE=7.62;
```

When this statement is executed, decimal point alignment occurs on the character V and not on the decimal point picture character that appears in the picture specification for RATE. If RATE were converted to a character string and then printed, it would appear as 762.00, but its arithmetic value would be 7.6200.

Unlike the character V, which can appear only once in a picture specification, the decimal point picture character can appear more than once; this allows digit groups within the numeric character data item to be separated by points, as is common in Dewey decimal notation and in the numeric notations of some European countries.

Because a decimal point picture character causes a period character to be inserted into the character-string value of a numeric character data item, it is called an insertion character. PL/I provides two other insertion characters: comma (,) and blank(B), which are used in the same way as the decimal point picture character except that a comma or blank is inserted into the character string. Consider the following statements:

```
DECLARE RESULT PICTURE '9.999.999,V99';

RESULT=1234567;
```

The character-string value of RESULT would be '1.234.567,00'. Note that decimal point alignment occurs before the two rightmost digit positions as specified by the character V. If RESULT were assigned to a coded arithmetic field, the value of the data converted to arithmetic would be 1234567.00.

Besides supplying insertion characters, PL/I also provides replacement characters that allow zeros in specified positions to be replaced by blanks or asterisks. One such picture character is the character Z, which is used to replace leading (leftmost) zeros with blanks:

```
DECLARE TALLY PICTURE 'ZZZ9';

TALLY=0012;
```

The character-string value of TALLY is equivalent to the character-string constant 'bb12' (where the letter b indicates a blank character).

Other picture characters control the appearance of signs and the currency symbol ($) in specified positions of the numeric character data items. For example, a dollar sign can be appended to the left of a numeric character item, as indicated in the following statements:

```
DECLARE PRICE PICTURE '$99V.99';

PRICE=12.45;
```

The character-string value of PRICE is equivalant to the character-string constant '$12.45'. Its arithmetic value, however, would be 1245 with precision of (4,2), or 12.45.

The picture specification can also specify floating-point and British sterling formats. These formats are discussed in Part II, Section D, "Picture Specification Characters."

Using Numeric Character Data

One purpose of a numeric character picture specification is to edit data that is

to be printed. For example, in a payroll application, the digits representing an employee's salary might be 0017250. These digits would be much more meaningful on a paycheck in an edited form, such as $**172.50; the asterisks would also discourage an attempt to alter the amount. This could be done, for example, with the specification '$****9.99'.

PL/I, however, does not restrict the use of numeric character data to output purposes. Numeric character variables can be used wherever arithmetic expressions are permitted. Consider the following example:

```
DECLARE RESULT PICTURE 'XXXXXX',
        COST PICTURE '$9V.99';
        COST=7.15;
        RESULT=COST;
```

In this example, the arithmetic value of COST would be 7.15. When COST is assigned to RESULT, however, the insertion characters ($ and .) appear as part of the character string, and the value of RESULT is '$7.15b'. The only differences between the numeric character data and the character-string data is that the character-string value is left adjusted (hence the blank at the right end) and the insertion characters are actually a part of the data, while with a numeric character variable, data is point aligned and insertion characters, though actually present, are not considered to be a part of the arithmetic value.

If specified in an arithmetic expression, the value of a numeric character data item is converted to coded arithmetic. Note, however, that this conversion will always require the compiler to insert extra coding. Note also, that any editing characters in the picture specification will be lost in the conversion. Consider the following example:

```
DECLARE RESULT FIXED DECIMAL(3,2),
            COST PICTURE '$9V.99';

  COST=1.10;

  RESULT=2*COST;
```

The character-string value of COST is $1.10. The editing characters ($ and .) are present in the item. However, when the expression 2* COST is evaluated, the arithmetic value of COST is converted to coded arithmetic. When the value of the expression is assigned to RESULT, the value of RESULT will be 2.20 (i.e., 220 with precision (3,2)).

CHARACTER-STRING AND BIT-STRING BUILT-IN FUNCTIONS

PL/I provides a number of built-in functions, two of which also can be used as pseudo-variables, to add power to the string-handling facilities of the language. Following are brief descriptions of these functions (more detailed descriptions appear in Part II, Section G, "Built-In Functions and Pseudo-variables").

The STRING built-in function specifies that the elements of a PACKED structure are to be concatenated into a single character string. All elements must be either character strings or numeric character fields.

The BIT built-in function specifies that a data item is to be converted to a bit string. The built-in function allows a programmer to specify the length of the converted string, overriding the length that would result from the standard rules of data conversion.

The CHAR built-in function is exactly the same as the BIT built-in function, except that the conversion is to a character string of a specified length.

The SUBSTR built-in function, which can also serve as a pseudo-variable in a receiving field, allows a specific substring to be extracted from (or assigned to, in the case of a pseudo-variable) a specified string value.

The INDEX built-in function allows a string, either a character string or a bit string, to be searched for the first occurrence of a specified substring, which can be a single character or bit. The value returned is the location of the first character or bit of the substring, relative to the beginning of the string. The value is expressed as a binary integer. If the substring does not occur in the specified string, the value returned is zero.

The HIGH built-in function provides a string of a specified length that consists of repeated occurrences of the highest character in the collating sequence. For System/360 implementations, the character is hexadecimal FF.

The LOW built-in function provides a string of a specified length that consists of repeated occurrences of the lowest character in the collating sequence. For System/360 implementations, the character is hexadecimal 00.

The REPEAT built-in function permits a string to be formed from repeated occurren-

ces of a specified substring. It is used to create string patterns.

The BOOL built-in function allows one of 16 different Boolean operations to be applied to two specified bit strings.

The UNSPEC built-in function, which can also be used as a pseudo-variable, specifies that the internal coded representation of a value is to be regarded as a bit string with no conversion.

ARGUMENTS AND PARAMETERS

Data can be made known to an invoked procedure by extending the scope of the names identifying that data to include the invoked procedure. This extension of scope is accomplished by nesting procedures or by specifying the EXTERNAL attribute for the names.

There is yet another way in which data can be made known to an invoked procedure, and that is to specify the names as arguments in a list in the invoking statement. Each argument in the list is an expression, a file name, a statement label constant or variable, or an entry name that is to be passed to the invoked procedure.

Since arguments are passed to it, the invoked procedure must have some way of accepting them. This is done by the explicit declaration of one or more parameters in a list in the PROCEDURE or ENTRY statement that is the entry point at which the procedure is invoked. A parameter is a name used within the invoked procedure to represent another name (or expression) that is passed to the procedure as an argument. Each parameter in the parameter list of the invoked procedure has a corresponding argument in the argument list of the invoking statement. This correspondence is taken from left-to-right; the first argument corresponds to the first parameter, the second argument corresponds to the second parameter, and so forth. In general, any reference to a parameter within the invoked procedure is treated as a reference to the corresponding argument. The number of arguments and parameters must be the same.

The example below illustrates how parameters and arguments may be used:

```
PRMAIN: PROCEDURE;
     DECLARE NAME CHARACTER (20),
       ITEM BIT(5);
     .
     .
     .
     CALL OUTSUB (NAME, ITEM);
     .
     .
     .
END;

OUTSUB: PROCEDURE (A,B);
     DECLARE A CHARACTER (20),
       B BIT(5);
     .
     .
     .
     PUT EDIT(A,B)(A(20),B(5));
     .
     .
     .
END;
```

In procedure PRMAIN, NAME is declared as a character string, and ITEM as a bit string. The CALL statement in PRMAIN invokes the procedure called OUTSUB, and the parenthesized list included in this procedure reference contains the two arguments being passed to OUTSUB. The PROCEDURE statement defining OUTSUB declares two parameters, A and B. When OUTSUB is invoked, NAME is associated with A and ITEM is associated with B. Each reference to A in OUTSUB is treated as a reference to NAME, and each reference to B is treated as a reference to ITEM. Therefore, the PUT statement causes the values of NAME and ITEM to be written into the standard system output file.

Note that the passing of arguments usually involves the passing of names and not merely the values represented by these names. (In general, the name that is passed is usually the address of the value.) As a result, storage allocated for a variable before it is passed as an argument is not duplicated when the procedure is invoked. Any change of value specified for a parameter actually is a change in the value of the argument. Such changes are in effect when control is returned to the invoking block.

A parameter can be thought of as indirectly representing the value that is directly represented by an argument. Thus, since both the argument and the parameter represent the same value, the attributes of a parameter and its corresponding argument must agree. For example, an error exists if a parameter has the attribute FILE and its corresponding argument has the attribute FLOAT.

A name is explicitly declared to be a parameter by its appearance in the parameter list of a PROCEDURE or ENTRY statement. However, its attributes, unless defaults apply, must be explicitly stated within that procedure in a DECLARE statement.

The parameters specified in an ENTRY statement must also have been specified either in the PROCEDURE statement for the containing procedure, or in a DECLARE statement within that procedure.

Parameters, therefore, provide the means for generalizing procedures so that data whose names may not be known within such procedures can, nevertheless, be operated upon. There are two types of generalized procedures that can be written in PL/I: subroutine procedures (called simply, subroutines) and function procedures (functions).

SUBROUTINES

A subroutine is a procedure that usually requires arguments to be passed to it in an invoking CALL statement. It can be either an external or internal procedure. A reference to such a procedure is known as a subroutine reference. The general format of a subroutine reference is as follows:

```
CALL entry-name[(argument[,argument]...)];
```

Whenever a subroutine is invoked, the arguments of the invoking statement are associated with the parameters of the entry point, and control is then passed to that entry point. The subroutine is thus activated, and execution begins.

Upon termination of a subroutine, control normally is returned to the invoking block. A subroutine can be terminated normally in any of the following ways:

1. Control reaches the final END statement of the subroutine. Execution of this statement causes control to be returned to the first executable statement logically following the statement that invoked the subroutine. This is considered to be a normal return.

2. Control reaches a RETURN statement in the subroutine. This causes the same normal return caused by the END statement.

3. Control reaches a GO TO statement that transfers control out of the subroutine. The GO TO statement may specify a label in a containing block, which must be known within the subroutine, or it may specify a parameter that has been associated with a label argument passed to the subroutine. Although this is considered to be normal termination of the subroutine, it is not normal return of control, as effected by an END or RETURN statement.

A STOP statement encountered in a subroutine abnormally terminates execution of that subroutine and of the entire program associated with the procedure that invoked it.

The following example illustrates how a subroutine interacts with the procedure that invokes it:

```
A:  PROCEDURE;
    DECLARE RATE FIXED(10,3),
            TIME FIXED(5,2),
            DISTANCE FIXED(15,5),
            MASTER FILE...;
          .
          .
          .
    CALL READCM (RATE, TIME, DISTANCE,
                 MASTER);
          .
          .
          .
    END;

READCM: PROCEDURE (W,X,Y,Z);
    DECLARE W FIXED(10,3),X FIXED(5,2),
         Y FIXED(15,5), Z FILE...;
          .
          .
          .
    GET FILE (Z) EDIT (W,X,Y) (F(10,3),
           F(5,2),F(15,5));
          .
          .
          .
    Y=W*X;
    IF Y > 0 THEN RETURN;
             ELSE PUT EDIT('ERROR READCM')
               (A(12));
    END;
```

The arguments RATE, TIME, DISTANCE, and MASTER are passed to the parameters W, X, Y, and Z. Consequently, in the subroutine, 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.

FUNCTIONS

A function is a procedure that usually requires arguments to be passed to it when it is invoked. Unlike a subroutine, which is invoked by a CALL statement, a function is invoked by the appearance of the function name (and associated arguments) in an expression. Such an appearance is called a function reference. Like a subroutine, a function can operate upon the arguments

passed to it and upon other known data. But unlike a subroutine, a function is written to compute a single value which is returned, with control, to the point of invocation, the function reference. This single value can be an arithmetic, string, picture, or pointer value. An example of a function reference is contained in the following procedure:

```
MAINP: PROCEDURE;
          .
          .
          .
       X=Y**3+SPROD(A,B,C);
          .
          .
          .
END;
```

In the above procedure, the assignment statement

```
X=Y**3+SPROD(A,B,C);
```

contains a reference to a function called SPROD. The parenthesized list following the function name contains the arguments that are being passed to SPROD. Assume that SPROD has been defined as follows:

```
SPROD: PROCEDURE (U,V,W);
          .
          .
          .
       IF U>V+W
             THEN RETURN (0);
             ELSE RETURN (U*V*W);
          .
          .
          .
END;
```

When SPROD is invoked by MAINP, the arguments A, B, and C are associated with the parameters U, V, and W, respectively. Since attributes have not been explicitly declared for the arguments and parameters, default attributes of FLOAT DECIMAL (6) are applied to each argument and parameter. Hence, the attributes are consistent, and the association of the arguments with the parameters produces no error.

During the execution of SPROD, the IF statement is encountered and a test is made. If U is greater than V + W, the statement associated with the THEN clause is executed; otherwise, the statement associated with the ELSE clause is executed. In either case, the executed statement is a RETURN statement.

The RETURN statement is the usual way by which a function is terminated and control is returned to the invoking procedure. Its use in a function differs somewhat from its use in a subroutine; in a function, not only does it return control but it also returns a value to the point of invocation. The general form of the RETURN statement, when it is used in a function, is as follows:

```
RETURN (element-expression);
```

The expression must be present and must represent a single value; i.e., it cannot be an array or structure expression. It is this value that is returned to the invoking procedure at the point of invocation. Thus, for the above example, SPROD returns either 0 or the value represented by U*V*W, along with control to the invoking expression in MAINP. The returned value then effectively replaces the function reference, and evaluation of the invoking expression continues.

A function can also be terminated by execution of a GO TO statement. If this method is used, evaluation of the expression that invoked the function will not be completed, and control will go to the designated statement. As in a subroutine, the transfer point specified in a GO TO statement may be a parameter that has been associated with a label argument. For example, assume that MAINP and SPROD have been defined as follows:

```
MAINP: PROCEDURE;
          .
          .
          .
       X=Y**3+SPROD(A,B,C,LAB1);
          .
          .
          .
LAB1:  CALL ERRT;
          .
          .
          .
END;

SPROD: PROCEDURE (U,V,W,Z);
       DECLARE Z LABEL;
          .
          .
          .
       IF U >  V + W
            THEN GO TO Z;
            ELSE RETURN (U*V*W);
              .
              .
              .
            END;
```

In MAINP, LAB1 is explicitly declared to be a statement label constant by its appearance as a label for the CALL ERRT statement. When SPROD is invoked, LAB1 is associated with parameter Z. Since the attributes of A must agree with those of LAB1, Z is declared to have the LABEL attribute. When the IF statement in SPROD is executed, a test is made. If U is greater than V + W, the THEN clause is executed, control returns to MAINP at the statement labeled LAB1, and evaluation of the expression that invoked SPROD is discontinued. If U is not greater than V + W, the ELSE clause is executed and a return to MAINP is made in the normal fashion. Additional information about the use of label arguments and label parameters is contained in the section "Relationship of Arguments and Parameters" in this chapter.

Note: In some instances, a function may be so defined that it does not require arguments. In such cases, the appearance of the function name within an expression will be recognized as a function reference only if the function name has been declared as an entry name elsewhere in the block. See "The ENTRY Attribute" in this chapter for additional information.

Attributes of Value Returned by Function

The attributes of the value returned by a function may be declared in two ways:

1. They may be declared by default according to the first letter of the function name.

2. They may be explicitly declared following the parameter list in the function PROCEDURE (or ENTRY) statement.

Regardless of which method is used, the data attributes for a secondary entry point, including any default attributes, must be identical with those established for the primary entry point. In other words, the attributes specified in an ENTRY statement (explicitly or by default) must in no way conflict with those specified in the PROCEDURE statement of the containing procedure.

Note that the value of the expression in the RETURN statement is converted within the function, wherever necessary, to conform to the attributes specified by one of the two methods above.

In the previous examples of MAINP and SPROD, the PROCEDURE statement of SPROD contains no attributes declared for the value it returns. Thus, these attributes must be determined from the first letter of its name, S. The attributes of the returned value are therefore FLOAT DECIMAL(6). Since these are the attributes that the returned value is expected to have, no conflict exists.

Note: Unless the invoking procedure provides the compiler with information to the contrary, the attributes of the value returned by a function to the invoking procedure are always determined from the first letter of the function name.

The way in which attributes can be declared for the returned value in the PROCEDURE or ENTRY statement is illustrated in the following example. Assume that the PROCEDURE statement for SPROD has been specified as follows:

```
SPROD: PROCEDURE (U,V,W,Z) FIXED BINARY;
```

With this declaration, the value returned by SPROD will have the attributes FIXED and BINARY. However, since these attributes differ from those that would be determined from the first letter of the function name, this difference must be stated in the invoking procedure to avoid a possible error. The PL/I programmer communicates this information to the compiler by specifying the RETURNS attribute in the invoking procedure.

The RETURNS attribute is specified in a DECLARE statement for a function entry name within the procedure invoking that function. It specifies the attributes of the value returned by that function. Unless default attributes for the entry name apply, any invocation of a function must appear within the scope of a RETURNS attribute declaration for the entry name. For an internal function, the RETURN attribute can be specified only in a DECLARE statement that is internal to the same block as the function procedure. If the RETURNS attribute is declared for an internal function, the INTERNAL attribute must be specified in the same declaration.

The general format of the RETURNS attribute is as follows:

```
RETURNS (attribute-list)
```

A RETURNS attribute specifies that within the invoking procedure the value returned from the named entry point is to be treated as though it had the attributes given in the attribute list. The word treated is used because no conversion is performed in an invoking block upon any value returned to it. Therefore, if the attributes of the returned value do not agree with those in the attribute list of the RETURNS attribute, an error will probably result.

Thus, in order to specify to the compiler that coding for MAINP is to handle the FIXED BINARY value being returned by SPROD, the following declaration must be given within MAINP:

```
DECLARE SPROD RETURNS (FIXED BINARY);
```

It is important to note some of the things that are implied in the above discussion. Principally, it should be remembered that during compilation of the invoking block, there is no way for the compiler to check a function procedure to determine the attributes of the value it returns. In the absence of explicit information in a RETURNS attribute specification, the compiler can only assume that the attributes will be consistent with the attributes implied by the first letter of the function name. This is true even if the function procedure is contained in the invoking procedure. If the returned value does not have the attributes that the invoking procedure is prepared to receive, no conversion can be performed. The RETURNS attribute must be declared for a function that returns any value with attributes not consistent with default attributes for the function name.

Built-In Functions

Similar to function procedures that a programmer can define for himself is a comprehensive set of pre-defined functions called built-in functions.

The set of built-in functions is an intrinsic part of PL/I. It includes not only the commonly used arithmetic functions but also other necessary or useful functions related to language facilities, such as functions for manipulating strings and arrays.

Built-in functions are invoked in the same way that programmer-defined functions are invoked. However, many built-in functions can return array values, whereas a programmer-defined function can return only an element value.

Note: Some built-in functions actually are compiled as in-line code rather than as procedure invocations. All are referred to in a PL/I source program, however, by function references, whether or not they result in an actual procedure invocation.

Neither the ENTRY attribute nor the RETURNS attribute can be specified for any built-in function name. The use of the name in a function reference is recognized without need for any further identification; attributes of values returned by built-in functions are known by the compiler.

But since built-in function names are PL/I keywords, they are not reserved; the same identifiers can be used as programmer-defined names (exceptions are TIME, DATE, and NULL; they cannot be implicitly declared). Consequently, ambiguity might occur if a built-in function (or pseudo-variable) reference were to be used in a block that is contained in another block in which the same identifier is declared for some other purpose. To avoid this ambiguity, the BUILTIN attribute can be declared for a built-in function name in any block that has inherited, from a containing block, some other declaration of the identifier. Consider the following example.

```
A: PROCEDURE;
      .
      .
      .
      B: BEGIN;
       DECLARE SQRT FLOAT BINARY;
            .
            .
            .
            C: BEGIN;
             DECLARE SQRT BUILTIN;
                  .
                  .
                  .
                  END;

            .
            .
            .
            END;

      .
      .
      .
      END;
```

Assume that in external procedure A, SQRT is neither explicitly nor contextually declared for some other use. Consequently, any reference to SQRT would refer to the built-in function of that name. In B, however, SQRT is declared to be a floating-point binary variable, and it cannot be used in any other way. Finally, in C, SQRT is declared with the BUILTIN attribute so that any reference to SQRT will be recognized as a reference to the built-in function and not to the floating-point binary variable declared in B.

A programmer can even use a built-in function name as the entry name of a programmer-written function and, in the same program, use both the built-in func-

tion and the programmer-written function. This can be accomplished by use of the BUILTIN attribute and the ENTRY attribute. (The ENTRY attribute, which is used in a DECLARE statement to specify that the associated identifier is an entry name, is discussed in a later section of this chapter.)

The following example illustrates use of the ENTRY attribute in conjunction with the BUILTIN attribute.

```
SQRT: PROCEDURE (PARAM) FIXED (6,2);
      DECLARE PARAM FIXED (12);
            .
            .
            .
      END;

A: PROCEDURE;
      DECLARE SQRT ENTRY RETURNS
            (FIXED(6,2)),Y FIXED(12);
            .
            .
            .
      X = SQRT(Y);
            .
            .
            .
            B: BEGIN;
                  DECLARE SQRT BUILTIN;
                        .
                        .
                        .
                  Z = SQRT (P);
                        .
                        .
                        .
                  END;
            .
            .
            .
      END;
```

The use of SQRT as the label of the first PROCEDURE statement is an explicit declaration of the identifier as an entry name. Since, in this case, SQRT is not the built-in function, the entry name must be explicitly declared in A (and the RETURNS attribute is specified because the attributes of the returned value are not apparent in the function name). The function reference in the assignment statement in A thus refers to the programmer-written SQRT function. In the begin block, the identifier SQRT is declared with the BUILTIN attribute. Consequently, the function reference in the assignment statement in B refers to the built-in SQRT function.

If a programmer-written function using the name of a built-in function is external, any procedure containing a reference to that function name must also contain an entry declaration of that name; otherwise a reference to the identifier would be a reference to the built-in function. In the above example, if B were not contained in A, there would be no need to specify the BUILTIN attribute; so long as the identifier SQRT is not known as some other name, the identifier would refer to the built-in function.

If a programmer-written function using the name of a built-in function is internal, any reference to the identifier would be a reference to the programmer-written function as long as its name is known in the block in which the reference is made. No entry name attributes would have to be specified if attributes to the returned value could be inferred from the entry name.

THE ENTRY ATTRIBUTE

As mentioned earlier, the ENTRY attribute is used to indicate that the associated identifier is an entry name. Such an indication is necessary if an identifier is not otherwise recognizable as an entry name, that is, if it is not explicitly or contextually declared to be an entry name in one of the following ways:

1. By appearing as a label of a PROCEDURE or ENTRY statement (explicit).

2. By appearing immediately following the keyword CALL (contextual).

3. By appearing as the function name in a function reference that contains an argument list (contextual).

Therefore, if a reference is made to an entry name in the block in which it does not appear in one of these three ways, the identifier must be given the ENTRY attribute explicitly, or by implication (see "Note" below), in a DECLARE statement within that block. For example, assume that the following has been specified:

```
A: PROCEDURE;
      .
      .
      .
   PUT EDIT (RANDOM) (E(10,5));
      .
      .
      .
END;
```

Assume also that A is an external procedure and RANDOM is an external function that requires no arguments and returns a random number. As the procedure is shown

above, RANDOM is not recognizable within A as an entry name, and the result of the PUT statement, therefore, is undefined. In order for RANDOM to be recognized within A as an entry name, it must be declared to have the ENTRY attribute. For example:

```
A: PROCEDURE;
     DECLARE RANDOM ENTRY;
     .
     .
     .
     PUT EDIT (RANDOM) (E(10,5));
     .
     .
     .
END;
```

Now, RANDOM is recognized as an entry name, and the appearance of RANDOM in the PUT statement cannot be interpreted as anything but a function reference. Therefore, the PUT statement results in the output transmission of the random number returned by RANDOM.

Note: The ENTRY attribute can be explicitly declared by implication. Any identifier that is explicitly declared to have the RETURNS attribute, is given the ENTRY attribute by implication. Thus, RETURNS implies ENTRY.

Entry Names as Arguments

An argument of a function or subroutine reference can itself be an entry name. When this is the case, one of the following pertains:

1. If the entry name argument, call it FUNC, is specified with an argument list of its own, it is recognized as a function reference; FUNC is invoked, and the value returned by FUNC effectively replaces the appearance of FUNC and its argument list in the containing argument list.

2. If the entry name argument appears without an argument list, but within an operational expression or within parentheses, then it is taken to be a function reference with no arguments. For example, the statement:

   ```
   CALL A((B));
   ```

   where B is known as an entry name, passes, as the argument to A, the value returned by the function procedure B.

3. If the entry name argument appears without an argument list and neither within an operational expression nor within parentheses, the entry name itself is always passed to the function or subroutine being invoked. In such cases, the entry name is never interpreted as a function reference, even if it is the name of a function that does not require arguments. For example, the statement:

   ```
   CALL A(B);
   ```

   when B is known as an entry name, passes the entry name B as an argument to A.

Consider the following example:

```
CALLP: PROCEDURE;
          DECLARE RREAD ENTRY;
          .
          .
          .
          GET EDIT (R,S) (2 F(10,5));
          .
          .
          .
          CALL SUBR (RREAD, ASQRT(R),
       S, LAB1);
          .
          .
          .
     LAB1: CALL ERRT(S);
          .
          .
          .
     END;

SUBR: PROCEDURE (NAME, X, Y, TRANPT);
          DECLARE NAME ENTRY, TRANPT
            LABEL;
          .
          .
          .
          IF X > Y THEN CALL NAME(Y);
                   ELSE GO TO TRANPT;
          .
          .
          .
     END;
```

In this example, assume that CALLP, SUBR, ASQRT, and RREAD are external procedures. In CALLP, RREAD is explicitly declared to have the ENTRY attribute and SUBR is contextually declared to have the ENTRY attribute. Four arguments are specified in the CALL SUBR statement. These are interpreted as follows:

1. The first argument, RREAD, is recognized as an entry name (because of the ENTRY attribute declaration). Since it does not have an argument list of its own, and since it does not appear in an operational expression or within parentheses, the entry name itself is passed at invocation.

2. The second argument, ASQRT(R), is recognized as a function reference because of the argument list accompanying the entry name. ASQRT is invoked and the value returned by ASQRT is assigned to a dummy argument (see "Dummy Arguments"), which effectively replaces the reference to ASQRT. When SUBR is invoked, the dummy argument is passed to it.

3. The third argument, S, is simply a decimal floating-point element variable which is passed as it is.

4. The fourth argument, LAB1, is a statement-label constant. Being a constant, a dummy argument must be created for it. When SUBR is invoked, the dummy argument is passed.

In SUBR, four parameters are explicitly declared in the PROCEDURE statement. If no further explicit declarations were given for these parameters, arithmetic default attributes would be supplied for each. Therefore, since NAME must represent an entry name, it is explicitly declared with the ENTRY attribute, and since TRANPT must represent a statement label, it is explicitly declared with the LABEL attribute. X and Y are arithmetic, so the defaults are allowed to apply.

Note that the appearance of NAME in the CALL statement does not constitute a contextual declaration of NAME as an entry name. Such a contextual declaration exists only if no explicit declaration applies, but, in this case, one does apply since the appearance of NAME in the parameter list constitutes an explicit declaration of NAME as a parameter. If attributes of a parameter are not explicitly declared in a complementary DECLARE statement, arithmetic defaults apply. Consequently, NAME must be explicitly declared to have the ENTRY attribute; otherwise, it would be assumed to be a binary fixed-point variable, and its use in the CALL statement would result in an error.

RELATIONSHIP OF ARGUMENTS AND PARAMETERS

When a function or subroutine is invoked, a relationship is established between the arguments of the invoking statement or expression and the parameters of the invoked entry point. This relationship is dependent upon whether or not dummy arguments are created.

DUMMY ARGUMENTS

In the introductory discussion of arguments and parameters it is pointed out that the name of argument and not its value is passed to a subroutine or function. However, there are times when an argument has no name. A constant, for example, has no name; nor does an operational expression. But the mechanism that associates arguments with parameters cannot handle such values directly. Therefore, the compiler must provide storage for such values and assign an internal name for each. These internal names are called dummy arguments. They are not accessible to the PL/I programmer, but he should be aware of their existence because any change to a parameter will be reflected only in the value of the dummy argument and not in the value of the original argument from which it was constructed.

A dummy argument is always created for any of the following cases:

1. If an argument is a constant

2. If an argument is an expression involving operators

3. If an argument is itself a function reference containing arguments.

4. If an argument is an expression in parentheses.

In all other cases, the argument name is passed directly. The parameter becomes identical with the passed argument; thus, changes to the value of a parameter will be reflected in the value of the original argument only if a dummy argument is not passed.

ARGUMENT AND PARAMETER TYPES

In general, an argument and its corresponding parameter may be of any data organization and type. For example, an argument may be a pointer provided that the corresponding parameter is also a pointer; it may be a bit string, provided that the corresponding parameter is a bit string, and so on. However, not all parameter/argument relationships are so clear-cut. Some need further definition and clarification. Such cases are given below.

If a parameter is an element, i.e., a variable that is neither a structure nor an array, the argument must be an element expression. If the argument is a subscripted variable, the subscripts are

evaluated before the subroutine or function is invoked and the name of the specified element is passed.

If a parameter is an array, the argument must be an array name. The data attributes of the argument must agree with those of the parameter. The bounds of the array argument must agree with the bounds of the array parameter.

If a parameter is a structure, the argument must be a structure name. The relative structuring of the argument and the parameter must be the same; the level numbers need not be identical. The data attributes of the elements of the structure argument must match those of the corresponding elements of the parameter.

If a parameter is an element label variable, the argument must be either an element-label variable or a label constant. If the argument is a label constant, a dummy argument is constructed.

If the parameter is an array label variable, the argument must be an array label variable with identical bounds.

If a parameter is an entry name, the argument must be an entry name. The name of a built-in function cannot be passed. (However, built-in function references can appear in argument lists because the value of the function reference and not the function name is passed.)

If a parameter is a file name, the argument must be a file name. In general, the attributes of the file name argument must match those of the file name parameter. However, for the D-Compiler, in some cases, a match is not required. This is true only for the BACKWARDS attribute and the following options of the ENVIRONMENT attribute:

```
BUFFERS(n)
LEAVE
NOLABEL
VERIFY
MEDIUM
```

In the case of the MEDIUM option, only the logical device name can be different; the physical device type must be the same.

When a file name argument does not match its corresponding parameter in any of the above cases, the argument prevails and the nonmatching ENVIRONMENT options or BACKWARDS attribute of the parameter are overridden. In all other cases, a match is always required and it is an error if any attributes do not match. Consider the following example:

```
A: PROCEDURE OPTIONS(MAIN);
   DECLARE X FILE RECORD INPUT
             BACKWARDS ENVIRONMENT
             (F(80)MEDIUM(SYS001,2400)
              BUFFERS(1) LEAVE),
           Y FILE RECORD INPUT
             ENVIRONMENT (F(80)
              MEDIUM(SYS002,2400)
              BUFFERS(2));
     .
     .
     .
   CALL B(X);
     .
     .
     .
   CALL B(Y);
     .
     .
     .
   B: PROCEDURE(Z);
      DECLARE Z FILE RECORD INPUT
                ENVIRONMENT (F(80)
                MEDIUM(SYS000,2400));
        .
        .
        .
      OPEN FILE (Z);
        .
        .
        .
      END B;
     .
     .
     .
   END A;
```

In this example, X has the BACKWARDS attribute but its corresponding parameter does not. Since this is one of the cases given above, a match is not required and Z effectively is given the BACKWARDS attribute the first time B is invoked. Similarly, the logical device name SYS001, the BUFFERS(1) specification, and the LEAVE option in the ENVIRONMENT attribute for X prevail over those given or assumed for Z. The OPEN statement therefore results in the opening of Z, with all of the attributes of X.

The second time that B is invoked, the action is the same, except that Z now corresponds to Y and, therefore, the attributes of Y prevail. Thus, for this invocation Z does not have the BACKWARDS attribute, its logical device name is SYS002, BUFFERS(2) applies, and LEAVE does not apply.

If a parameter is an element pointer variable, the argument must be an element pointer variable or an element pointer expression.

If a parameter is a pointer array, the argument must be a pointer array with identical bounds.

A parameter has no storage class and therefore cannot be declared with any storage class attribute. All arguments must be either STATIC or AUTOMATIC; they cannot be BASED.

If a parameter is an array or a string, the bounds of the array or the length of the string must be specified in the same way that they must be specified for non-parameters; i.e., as decimal integer constants. They must be the same as the bounds and lengths for the corresponding arguments.

Note that the base, scale, and precision of an arithmetic constant passed as an argument must be the same as that of its corresponding parameter. Similarly, the length of a string constant passed as an argument must be the same as that of its corresponding parameter.

When a PL/I program is executed, a large number of exceptional conditions are monitored by the system and their occurrences are automatically detected whenever they arise. These exceptional conditions may be errors, such as overflow or an input/output transmission error, or they may be conditions that are expected but infrequent, such as the end of a file or the end of a page when output is being printed.

Each of the conditions for which a test may be made has been given a name, and these names are used by the programmer to control the handling of exceptional conditions. The list of condition names is part of the PL/I language. For keyword names and descriptions of each of the conditions, see Part II, Section H, "ON Conditions."

ENABLED CONDITIONS AND ESTABLISHED ACTION

A condition that is being monitored, and the occurrence of which will cause an interrupt, is said to be enabled. Any action specified to take place when an occurrence of the condition causes an interrupt, is said to be established.

Most conditions are checked for automatically, and when they occur, the system will take control and perform some standard action specified for the condition. These conditions are enabled by default, and the standard system action is established for them.

The most common system action is to raise the ERROR condition. This provides a common condition that may be used to check for a number of different types of errors, rather than checking each error type individually. Standard system action for the ERROR condition is to terminate the program.

The programmer may specify whether or not some conditions are to be enabled, that is, are to be checked for so that they will cause an interrupt when they arise. If a condition is disabled, an occurrence of the condition will not cause an interrupt.

All input/output conditions and the ERROR condition are always enabled and cannot be disabled. All of the computational conditions may be enabled or disabled. The SIZE condition must be explicitly enabled if it is to cause an interrupt; all other conditions are enabled by default and must be explicitly disabled if they are not to cause an interrupt when they occur.

Condition Prefixes

Enabling and disabling can be specified for certain conditions by a condition prefix. A condition prefix is a list of one or more condition names, enclosed in parentheses and separated by commas, and connected to a statement (or a statement label) by a colon. The prefix always precedes the statement and any statement labels. A condition name in a prefix list indicates that the corresponding condition is enabled within the scope of the prefix. Some condition names can be preceded by the word NO, without a separating blank or connector, to indicate that the corresponding condition is disabled.

Scope of the Condition Prefix

The scope of the prefix, that is, the part of the program throughout which it applies, is usually the statement to which the prefix is attached. The prefix does not apply to any functions or subroutines that may be invoked in the execution of the statement.

A condition prefix to an IF statement applies only to the evaluation of the expression following the IF; it does not apply to the statements in the THEN or ELSE clauses, although these may themselves have prefixes. Similarly, a prefix to the ON statement has no effect on the associated on-unit. A condition prefix to a DO statement applies only to the evaluation of any expressions in the DO statement itself and not to any other statement in the DO-group.

Condition prefixes to the PROCEDURE statement and the BEGIN statement are special (though commonly used) cases. A condition prefix attached to a 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. It does not apply to any procedures

lying outside that block, which may be invoked during execution of the program.

The enabling or disabling of a condition may 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). Such a 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. When control passes out of the scope of the redefining prefix, the redefinition no longer applies. A condition prefix can be attached to any statement except a DECLARE or ENTRY statement.

Consider the following example:

```
(SIZE): A: PROCEDURE;
             .
             .
             .
 (NOSIZE): B: BEGIN;
                .
                .
                .
              END B;
           .
           .
           .
         END A;
```

In this example, the condition prefix SIZE enables that condition for procedure A and specifies that if a SIZE error occurs during any calculation in A, an interrupt is to take place. Ordinarily, the scope of the SIZE prefix would include begin block B; however, the NOSIZE prefix on the BEGIN statement disables SIZE within B and precludes any interrupt for a SIZE error therein.

The ON Statement

A system action exists for every condition, and if an interrupt occurs, the system action will be performed unless the programmer has specified an alternate action in an ON statement for that condition. The purpose of the ON statement is to establish the action to be taken when an interrupt results from an exceptional condition that has been enabled, either by default or by a condition prefix.

Note: The action specified in an ON statement will not be executed during any portion of a program throughout which the condition has been disabled.

The form of the ON statement is:

```
ON condition-name {SYSTEM;|on-unit}
```

(see Part II, Section J, "Statements" for a full description).

The keyword SYSTEM followed by a semicolon specifies standard system action whenever an interrupt occurs. It re-establishes standard system action for a condition for which some other action has been established. The on-unit is used by the programmer to specify an alternate action to be taken whenever an interrupt occurs. An on-unit must be either a null statement or a GO TO statement; it cannot be labeled.

A null statement on-unit effectively ignores the interrupt and, in general, returns control to the point logically following the point at which the interrupt occurred. Thus, the effect of a null on-unit is to say "When an interrupt occurs as a result of this condition, do nothing except continue."

Use of the null on-unit is not the same as disabling a condition, for two reasons: first, a null on-unit can be specified for any condition (except ENDFILE, KEY, and CONVERSION), but not all conditions can be disabled; and, second, disabling a condition, if possible, may save time by avoiding any checking for this condition. If a null on-unit is specified, the system must still check for the occurrence of the condition, transfer control to the on-unit whenever an interrupt occurs, and then, after doing nothing, return from the on-unit.

Note: The specific point to which control returns from a null on-unit varies for different conditions. In most cases, it returns to the point that immediately follows the action in which the condition arose. Section H, "ON-Conditions" gives the point of return for all conditions for which a null on-unit can be specified. The return from a null on-unit is called a normal return.

If an on-unit is a GO TO statement, then when an interrupt occurs, control is transferred to the label specified in the GO TO statement. Linkage to the point at which the interrupt occurred is thus lost and a normal return cannot occur.

Scope of the ON Statement

The execution of an ON statement associates an action specification with the named condition. Once this association is established, it remains until it is overridden or until termination of the block in which the ON statement is executed.

An established interrupt action passes from a block to any block it activates, and the action remains in force for all subsequently activated blocks unless it is overridden by the execution of another ON statement for the same condition. If it is overridden, the new action remains in force only until that block is terminated. When control returns to the activating block, all established interrupt actions that existed at that point are re-established. This makes it impossible for a subroutine to alter the interrupt action established for the block that invoked the subroutine.

If more than one ON statement for the same condition appears in the same block, each subsequently executed ON statement permanently overrides the previously established condition. No re-establishment is possible, except through execution of another ON statement with an identical action specification (or re-execution, through some transfer of control, of an overridden ON statement).

```
A: PROCEDURE;
   ON CONVERSION GO TO AERR;
   ON ZERODIVIDE GO TO BERR;
     .
     .
     .
   CALL B;
     .
     .
     .
   END A;

(NOOVERFLOW): B: PROCEDURE;
                 DECLARE Z BIT(1),
                         X CHARACTER(1);
                   .
                   .
                   .
                 ON CONVERSION GO TO CERR;
                   .
                   .
                   .
 (NOCONVERSION): Z = X;
                   .
                   .
                   .
                 RETURN;
                 END B;
```

The ON statements in procedure A establish the actions to be taken for the CONVERSION and ZERODIVIDE errors occurring within A. (Note that CONVERSION and ZERODIVIDE are enabled by default and therefore do not require condition prefixes to enable them.) These action specifications carry over into procedure B, because it is invoked by A, and remain in force until the ON statement in B is executed. This ON statement establishes a new action for the CONVERSION condition, which new action remains in force for the remainder of B. When control returns to A, the action specification for CONVERSION within A is re-established (the action specification for ZERODIVIDE, not having been changed in B, does not need to be re-established).

Note that the scope of the ON statement within B does not include the assignment statement since the NOCONVERSION prefix disables the CONVERSION condition for that statement. Thus, a CONVERSION error occurring during execution of the assignment statement does not cause an interrupt.

If a CONVERSION error occurs before the ON statement in B is executed, the action established in A is taken; that is, control is transferred to AERR. Similarly, a ZERODIVIDE error occurring anywhere within B results in a transfer to BERR.

The OVERFLOW condition is enabled by default and, since there is no ON statement for OVERFLOW within A, an OVERFLOW error within A causes the standard system action for OVERFLOW to be taken. However, within B, no action is taken for an OVERFLOW error because a NOOVERFLOW condition prefix has been attached to the PROCEDURE statement for B, and, as a result, OVERFLOW is disabled in B. When control returns to A, OVERFLOW is enabled once again.

The REVERT Statement

The REVERT statement is used to cancel the effect of one or more previously executed ON statements. It can affect only ON statements that are internal to the block in which the REVERT statement occurs and which have been executed in the same invocation of that block. The effect of the REVERT statement is to cancel the effect of any ON statement for the named condition that has been executed in the same block in which the REVERT statement is executed. It then re-establishes the action that was in force at the time of activation of that block.

A REVERT statement that is executed in a block in which no action has been esta-

blished for the named condition is treated as a null statement.

```
(SIZE): A: PROCEDURE;
           ON SIZE GO TO AERR;
             .
             .
             .
           CALL B;
             .
             .
             .
           END A;

(SIZE): B: PROCEDURE;
           ON SIZE GO TO BERR;
             .
             .
             .
           ON SIZE GO TO CERR;
             .
             .
             .
           REVERT SIZE;
             .
             .
             .
           RETURN;
           END B;
```

In this example, if a SIZE error occurs in procedure B after the execution of the first ON statement in B but before the execution of the second ON statement, an interrupt occurs and control is transferred to BERR; if a SIZE error occurs in B after the execution of the second ON statement but before the execution of the REVERT statement, an interrupt occurs and control is transferred to CERR; if a SIZE error occurs in B after the execution of the REVERT statement, an interrupt occurs and control is transferred to AERR. Thus, the REVERT statement re-establishes the action specification for SIZE as it existed at the point of invocation of B (that is, as it existed in A when the CALL B statement was executed).

The SIGNAL Statement

The programmer may simulate the occurrence of an ON condition by means of the SIGNAL statement. An interrupt will occur unless the named condition is disabled. This statement has the form:

```
SIGNAL condition-name;
```

The SIGNAL statement causes execution of the interrupt action currently established for the specified condition. The principal use of this statement is in program checking, to test the action of an on-unit, and to determine that the correct action is associated with the condition. If the signalled condition is not enabled, the SIGNAL statement is treated as a null statement.

For each identifier used in a PL/I program, the compiler must be able to determine the attributes associated with the name in order to generate correct code. For example:

```
A = B + C;
```

If A, B, and C are floating-point variables, then floating-point instructions will be compiled; if they are fixed-point, fixed-point instructions will be compiled.

In addition to determining the type of operation, the compiler must also be able to determine the address of each operand. In some cases, the compiler must generate code that will determine the address when the program is executed. The storage class of a variable determines the way in which the address is obtained. There are three distinct cases:

1. Static storage. The offset from a fixed origin can be determined when the program is loaded.

2. Automatic storage. The origin and the offset of the address are determined upon entry to the block.

3. Based storage. With each of the other classes of storage, the address used when an element is referred to is determined by the system. Indeed, one of the main advantages of using a language such as PL/I is that the programmer need not concern himself with addresses and address computation. However, in keeping with the general design of PL/I, the facility is available for the programmer to exercise direct control over addresses.

It is this third class, based storage, and address manipulation with which this chapter is concerned.

POINTER VARIABLES

A special type of variable, the pointer variable, is used to specify addresses in PL/I. While a pointer variable may not, in some implementations, actually contain an address, it is used to locate data in storage; consequently, it may be thought of as an address.

BASED VARIABLES

A based variable is a description of data that can be applied to different locations in storage, depending upon the value of the associated pointer variable.

Using based storage, the programmer can (1)explicitly specify the address to be used when accessing a variable, and (2)locate the storage area of a variable to be transmitted by RECORD-oriented input/output.

When a based variable is declared, it must be associated with a pointer that has been explicitly declared. The form of the declaration is:

```
identifier BASED (pointer-variable)
```

For example:

```
DECLARE P POINTER;

DECLARE A BASED (P);
```

Whenever a reference is made to A, the address must be derived using the value of a pointer variable. The pointer variable used is the one that appears in the declaration of the based variable, in this case, P. For example:

```
A = A + 1;
```

In this statement, the pointer used to determine the address of A will, in both cases, be P.

So long as an associated pointer variable has a valid value, any reference to the based variable is treated as if it had been allocated in the location identified by the pointer variable.

POINTER SPECIFICATION

A pointer variable must be associated with the based variable in the DECLARE statement that names the based variable. The pointer variable specified must be one that is explicitly declared elsewhere with the POINTER attribute.

A restriction imposed by the D-Compiler is that the pointer name used in the declaration of a based variable must be an unsubscripted, unqualified element variable.

Arrays of pointers are allowed, and pointers can be elements of structures, but those pointers cannot be associated with a based variable in a declaration.

VALUES OF POINTER VARIABLES

Before a reference can be made to a based variable, a value must be given to the pointer with which it is associated. This can be done in any of four different ways: with the SET option of a READ or LOCATE statement; by assignment of the value of another pointer; or by assignment of the value returned by the ADDR built-in function.

READ and SET

The READ statement with a SET option causes a record to be read into a buffer and the specified pointer variable to be set to point to the buffer. A based variable, declared with the same pointer, can then be used to refer to the fields of the record.

A based variable used to describe a record in a buffer has the effect of being overlaid on the buffer. The result of a reference to an element of the based variable is the same as if the record had been read directly into the structure described.

LOCATE and SET

The LOCATE statement, which always must have a SET option, allocates storage for a based variable in an output buffer. The action is similar to that of the READ and SET, in that the based variable is, in effect, overlaid on the buffer. In this case, however, the description is used to move data into the output buffer in locations relative to the descriptions of the elements of the based variable.

Assignment of Pointer Value

The value of a pointer variable may be assigned to another pointer variable in a simple assignment statement. Assume that Q and P are pointer variables and that P has a valid pointer value.

```
Q = P;
```

This statement specifies that Q is to be set to point to the same location that P points to. Reference to a based variable using either P or Q as the associated pointer is a reference to the same location in storage. Note that a pointer variable can be assigned a pointer value also by a reference to a programmer-defined function that returns a pointer value. Thus, in the above example, P could be a programmer-defined function that returns a pointer value.

Assignment of the ADDR Function Value

The value returned to an ADDR built-in function reference is a valid pointer value that specifies the location of a data variable named as the argument of the function reference. For example:

```
P = ADDR(A);
```

Execution of this assignment statement will give the pointer variable P a value so that it points to the location of the data variable A. The value of an ADDR function reference can be assigned to a pointer variable only.

The argument of the ADDR function reference can be a variable that represents an element, an array, an element of an array, a major structure, a minor structure, or an element of a structure. The argument may be a based variable or a nonbased variable.

Since the ADDR function can be used to set a pointer to point to a nonbased variable, this facility allows the use of a based variable to refer to the value of a nonbased variable.

DECLARATION OF POINTER VARIABLES

A pointer variable must be explicitly declared with the POINTER attribute in a DECLARE statement. Arrays of pointers can

be declared, or an elementary name of a structure can be declared to be a pointer variable. By default, a pointer variable is given the AUTOMATIC storage class attribute, but STATIC may be declared for it. A pointer variable cannot have the BASED attribute. Following are examples of pointer declarations:

```
DECLARE A POINTER,
     1 ELEMENT,
          2 P POINTER,
          2 C CHARACTER (10),
     X(10) POINTER STATIC;
```

Note: A pointer array variable must be subscripted to indicate a single element when it is used in a SET option.

POINTER VARIABLE RESTRICTIONS

Because a pointer is very closely related to an address, its value is strongly dependent upon the implementation in which it is used. In order to reduce implementation dependence, some restrictions are made on the use of pointer variables.

1. Pointer variables may not be operands of any operations except the comparison operations specified by the operators = and ¬=.

2. Assignment of a pointer variable value may be made only to another pointer variable.

3. Pointer variables cannot be used for STREAM input and output. When used in RECORD input and output, a pointer value written as output cannot be assumed to locate the same data if it is read back in.

THE USE OF BASED STORAGE AND POINTERS

The based storage and pointer handling facilities provided by the D-Compiler are primarily intended to permit the processing of records in input and output buffers. This can result in a significant saving of storage, particularly when many different record types exist in the same file.

Many different declarations of based variables can be associated with the same pointer. The effect of this is that once the pointer has been given a value, say by a READ statement with a SET option, then any of the record descriptions associated with the pointer may be used to refer to the record in the buffer. For example:

```
DECLARE P POINTER;
DECLARE 1 ISSUE BASED (P),
            2 CODE CHARACTER(1),
            2 PART_NO PICTURE '9999999',
            2 QTY PICTURE '9999',
            2 DEPT PICTURE '99',
            2 JOB_NO PICTURE '9999',
          1 RECEIPT BASED (P),
            2 CODE CHARACTER(1),
            2 PART_NO PICTURE '9999999',
            2 QTY PICTURE '9999',
            2 SUPPLIER PICTURE '99999';
READ FILE (TRANS) SET (P);
IF ISSUE.CODE = 'R' THEN GO TO RL1;
IF SUPPLIER >1000 THEN GO TO INHS1;
```

In this example, the two record descriptions ISSUE and RECEIPT are associated with the same pointer. Once P has been given a value by execution of a READ statement with a SET option, either of the two records can be referred to. The records do not require working storage, since the pointer refers to a position within the buffer.

The records can also contain variables other than character strings and numeric character fields. Any number of records can be associated with the same pointer. When the pointer is given a value, all of the records will refer to the same storage and will effectively be overlaid. Such overlaying of record descriptions can be machine dependent and should be used with care.

VARIABLE-LENGTH PARAMETER LISTS

In PL/I, a programmer-written procedure can have only a fixed number of parameters, all of which must be specified. Arguments are associated with parameters by passing addresses (which may be addresses of dummy arguments). By passing an array of pointers as a single argument, it is possible to simulate a variable-length parameter list, since some of the array elements may be null.

The following procedure checks if a value lies between two limits. Either the upper limit or the lower limit may be specified. The procedure has two parameters, a value that is to be checked and an array of two pointers. The first pointer specifies the upper limit, the second the lower limit. If either limit is not to be checked, the associated pointer is null when passed (see "Pointer Manipulation" below for a discussion of the NULL built-in function). The procedure returns the value

'1'B (or true) if the value lies between the limits, or the value '0'B if it does not.

```
LIMIT:  PROCEDURE (X,P) BIT(1);
   DECLARE  P(2) POINTER,
            (P1,P2) POINTER,
            TOP BASED (P1),
            BOTTOM BASED (P2);
   IF P(1) ¬= NULL
      THEN DO;
            P1 = P(1);
            IF X >= TOP
                        THEN RETURN('0'B);
            END;
   IF P(2) ¬=NULL
      THEN DO;
            P2 = P(2);
            IF X <= BOTTOM
                        THEN RETURN('0'B);
            END;
   RETURN('1'B);
   END LIMIT;
```

A procedure that invokes LIMIT might contain:

```
DECLARE LIMIT RETURNS BIT(1),
        Q(2) POINTER;
Q(1) = ADDR(HIGH);
Q(2) = NULL;
IF LIMIT (Y,Q) THEN DO;
```

Note that since a pointer in a based variable declaration cannot be subscripted, it is necessary to define two other pointer variables that are used to refer to the limits TOP and BOTTOM.

Since the procedure LIMIT does not return a fixed-point binary value of precision 15 (as would otherwise be implied by its initial letter), it must be declared with the RETURNS attribute in the invoking procedure.

In the invoking procedure, values are assigned to Q(1) and Q(2) using the built-in functions ADDR and NULL. The procedure LIMIT is then invoked by the function reference in the IF statement.

POINTER MANIPULATION

Two important built-in functions are provided by PL/I which can be used in manipulating pointer variables. They are the ADDR built-in function and the NULL built-in function.

The first of these, the ADDR built-in function, has already been discussed briefly. It requires one argument, the name of a variable, and it returns a value that points to the variable. It can be used to find the address of an element variable, an array variable, an element of an array, a major structure, a minor structure, or an element of a structure.

The ADDR function returns a value that identifies the address of a nonbased or based variable argument.

When using the ADDR function with arrays and structures, it is important to note that the ADDR of the first element of an array or structure is the same as the ADDR of the array or structure itself.

For example, given the following declarations:

```
DECLARE P POINTER;

DECLARE B(10,10) BASED (P),
        A(10,10);
```

ADDR(A(1,1)) is the same as ADDR(A) and, with the following assignment:

```
P = ADDR(A);
```

B(1,1) will refer to the first element of A.

It is entirely up to the programmer to ensure that such references do access meaningful storage locations, which must have been allocated in some other way and whose attributes are correct. It is well worth emphasizing that the power provided by the facility can be offset by extreme implementation dependence unless it is used carefully.

The second built-in function for pointer manipulation is the NULL function. The NULL function requires no arguments in a function reference. It returns a pointer value which is null; that is, a value that does not refer to a valid address.

```
C72A3:  PROCEDURE OPTIONS (MAIN);                                       01
        DECLARE PAGE_NO FIXED DECIMAL,                                  02
             1 CARDIN,                                                  03
                2 ACCNT_NO PICTURE '(8)9',                              04
                2 NAME CHARACTER (25),                                  05
                2 ADDRESS CHARACTER (25),                               06
                2 PAYMENT PICTURE '$$$$9V.99',                          07
                2 REST CHARACTER (14),                                  08
             1 B,                                                       09
                2 BALANCE PICTURE '$$$$$9V.9R',                         10
                2 REST1 CHARACTER (71),                                 11
             WRKA CHARACTER (8) DEFINED CARDIN,                         12
             WRKB CHARACTER (9) DEFINED B,                              13
             PAYMNT FILE INPUT RECORD ENVIRONMENT                       14
                        (CONSECUTIVE F(80) MEDIUM(SYS003,2540)),        15
             ACCNTS FILE UPDATE RECORD DIRECT KEYED                     16
                        ENVIRONMENT (REGIONAL(1) F(80)                  17
                                     MEDIUM(SYS001,2311)),              18
             EXCP FILE STREAM OUTPUT PRINT                              19
                        ENVIRONMENT (MEDIUM(SYS002,1403) F(133));       20
        OPEN FILE (PAYMNT), FILE (ACCNTS);                              21
        ON ENDFILE (PAYMNT) GO TO EOF;                                  22
  /* SET UP PAGE CONTROL */                                            23
        ON ENDPAGE (EXCP) GO TO NEW_PAGE;                               24
  /* SET UP HEADINGS AND THEN PRINT HEADINGS FOR FIRST PAGE */          25
        PAGE_NO = 0;                                                    26
NEW_PAGE: PAGE_NO = PAGE_NO + 1;                                        27
        PUT FILE (EXCP) PAGE LINE(3) EDIT ('PAGE ', PAGE_NO)            28
                                          (X(10), A(5), F(5));          29
        PUT FILE (EXCP) LINE(5) EDIT ('NO PAYMENT RECEIVED FROM')       30
                                     (X(30), A(24));                    31
        PUT FILE (EXCP) LINE(10)                                        32
            EDIT ('ACCOUNT NO', 'NAME', 'ADDRESS', 'BALANCE DUE')       33
                 (COLUMN(5), A(11), COLUMN(16), A(5),                   34
                  COLUMN(41), A(7), COLUMN(66), A(11));                 35
  /* TEST TO SEE IF NEW_PAGE ENTERED ON INTERRUPT */                    36
        IF PAGE_NO = 1 THEN GO TO NEWCARD;                              37
                       ELSE GO TO ZZ;                                   38
  /* MAIN UPDATE LOOP */                                               39
NEWCARD: READ FILE (PAYMNT) INTO (CARDIN);                              40
        READ FILE (ACCNTS) INTO (B) KEY(ACCNT_NO);                      41
        IF PAYMENT = 0 THEN                                             42
                       IF BALANCE<= 0 THEN GO TO NEWCARD;               43
                                      ELSE                              44
                                       ZZ: PUT FILE (EXCP) SKIP EDIT    45
                                              (WRKA,NAME,ADDRESS,WRKB)  46
                                              (COLUMN(5), A(8),         47
                                               COLUMN(16), A(25),       48
                                               COLUMN(41), A(25),       49
                                               COLUMN(66), A(9));       50
                       ELSE DO;                                         51
                           BALANCE=BALANCE-PAYMENT;                     52
                           REWRITE FILE(ACCNTS) FROM(B) KEY(ACCNT_NO);  53
                           END;                                         54
         GO TO NEWCARD;                                                 55
    EOF: CLOSE FILE(PAYMNT), FILE(ACCNTS), FILE(EXCP);                  56
END C72A3;                                                              57
```

Figure 13-1. A PL/I Program

Figure 13-1 is an example of a complete PL/I program. Note, however, that it is intended merely to illustrate how certain features of PL/I can be used; it is not intended that the program be used to solve a problem.

This example illustrates the use of PL/I for some common operations. The program reads a card and tests to see whether a payment has been made. If one has, the amount of the payment is subtracted from the balance in the account, which is in another file. If a payment has not been made, and if there is a balance due, the person's name, address, account number, and balance are printed.

The pattern of indention illustrates the free format allowed by PL/I. The D-Compiler requires that the first column of every card in the source program be blank; columns 73 through 80 of these cards are ignored and can contain any information (in this example, card sequence numbers appear in columns 79 and 80). So long as these margin restrictions are followed, statements can begin and end at any place. Statements can be continued from card to card without any continuation notation, as are the DECLARE statement and the PUT statements in this example. Constants can be continued from card to card provided that the last character in the first card is in column 72 and the first character in the next card is in column 2.

The PROCEDURE statement in card 1 names the procedure; the MAIN option is an implementation-defined option that specifies the initial procedure of the program (which may consist of more than one external procedure).

The DECLARE statement in cards 2 through 20 declares the attributes for the identifiers used in the procedure.

PAGE_NO in card 2 is given the attributes FIXED and DECIMAL. Note that since no precision is specified, PAGE_NO is given the default precision, which is (5,0).

The structure declaration in cards 3 through 8 describes the input record CARDIN. This record has an account number in the first 8 columns (note that this account number is also the key that will be used to find the account in the accounts file), a name in columns 9 through 33, an address in the next 25 columns, and a three- to six-digit quantity with a leading dollar sign and an embedded decimal point in the next 8 columns (the quantity is right-adjusted). The remainder of the record does not contain any information. REST, which is associated with this part of the record, is declared so that the remainder of the 80 columns will be accounted for.

Cards 9, 10, and 11 declare a structure that describes the records that are read from the accounts file (ACCNTS) and that contain the balance due for each amount. Each record has the balance due (BALANCE) in columns 1 through 9. The remaining 71 columns are not used but are accounted for by REST1. Each record in the file is identified by a key, which is the account number. Note that the R picture character in the declaration of BALANCE provides for the storing of a minus sign should the contents of BALANCE become negative.

In cards 12 and 13, WRKA and WRKB are declared as character strings defined on the first eight character positions of CARDIN (and, hence, effectively, ACCNT_NO) and the first nine character positions of B (and, hence, effectively, BALANCE), respectively. This is done because the value of a character string that has been defined on a numeric character variable is always the character-string value of that variable. This eliminates the need to assign the numeric character variable to a character-string variable in order to print the character-string value of the numeric character variable.

Cards 14 and 15 contain a declaration of PAYMNT as a buffered, record-oriented input file. This file has fixed-length records of length 80 and is to be read from the logical device SYS003 which is attached to an IBM 2540 Card Reader. The organization of the data set that is associated with the file is CONSECUTIVE, which means that the (n+1)th record of the file is located after the nth record of that file.

Cards 16, 17, and 18 contain a declaration of the accounts file, ACCNTS. This file is a direct-access update file that has a key associated with each record. REGIONAL(1) specifies that a key is used to refer to a record by its relative location with respect to the first record in the file. The file contains fixed-length records of length 80. The logical device name for the file is SYS001 and the physical device type is 2311 (that is, an IBM 2311 Disk Storage Drive).

Cards 19 and 20 contain a declaration of the file EXCP, which is to be used to record those accounts for which a balance is due but no payment has been made. It is an output print file. Its logical device name is SYS002 and its physical device type is 1403, an IBM 1403 Printer.

The first executable statement in the program is the OPEN statement in card 21. This statement opens the files PAYMNT and

ACCNTS. EXCP is implicitly opened by the first PUT statement, which is in card 28.

Card 22 contains an ON statement that establishes the action to be taken when the last payment card has been read.

The ON ENDPAGE statement in card 24 establishes an action for any ENDPAGE interrupt occurring for the file EXCP. Note that this action (that is, GO TO NEW_PAGE) is taken only when an ENDPAGE interrupt for the file occurs; the execution of the ON statement merely establishes the action.

The assignment statement in card 26 initializes PAGE_NO to zero. The assignment statement in card 27 increments PAGE_NO by 1 in order to update the page number for the print file and also to ensure that the test in cards 37 and 38 will produce the desired result.

The statements in cards 28 through 35 print the page headings. The first PUT statement (cards 28 and 29) starts a new page and then spaces two lines. The page headings are then printed on the third line. The interaction of the data list and the format list proceeds as follows:

> Space ten columns; assign the constant 'PAGE' to a five-character field; convert the value in PAGE_NO to an integer and place it, right-adjusted, into a five-character field.

The second PUT statement (cards 30 and 31) prints the general heading NO PAYMENT RECEIVED FROM on the fifth line of the page.

The third PUT statement (cards 32 through 35) prints the column headings. In this statement, the COLUMN format item is used to position the fields, so that the first heading starts in column 5, the second heading in column 16, the third heading in column 41, and the fourth heading in column 66.

Cards 37 and 38 contain an IF statement that tests to see if the NEW_PAGE routine was entered from an interrupt or from the normal sequence of program execution. If PAGE_NO is 1, then the NEW_PAGE routine has been entered normally. In all other cases, the routine has been entered because of an interrupt and the only possible point at which this interrupt could have occurred is the PUT statement labeled ZZ (see card 45). Any such interrupt would occur before the data list could be transmitted so a transfer of control back to that PUT statement (after printing headings for the new page) results in the printing of the data that was to have been transmitted when ENDPAGE was raised.

The main loop of the program starts with the statement labeled NEWCARD in card 40. This READ statement specifies that a record from the file PAYMNT is to be read into the structure CARDIN.

The next READ statement (card 41) uses the value of ACCNT_NO (obtained by the preceding READ statement) as the key identifying the record to be read into the structure B.

The IF statement in card 42 tests to see whether a payment has been made. If none has been made, the THEN clause (which contains another IF statement in card 43) is executed. This IF statement checks to see whether the account balance is less than or equal to zero; if the payment is zero and the account balance is less than or equal to zero, a new card is read. If the payment is equal to zero, but the account balance is greater than zero, the PUT statement in line 45 prints the information for the delinquent account. The format list in this statement uses the COLUMN format item to line up the data under the headings described in the NEW_PAGE routine (cards 27 through 37). The SKIP option in the PUT statement (card 45) insures that a new line is started each time the PUT statement is executed.

If the payment is not equal to zero, the DO-group of the ELSE clause starting in card 51 is executed. The assignment statement in this group subtracts the payment from the balance in order to form a new balance. The next statement (card 53) rewrites the record into the file ACCNTS, using the value of ACCNT_NO as the key. Note that the record is rewritten only if the value of BALANCE has been changed. Note also that the REWRITE statement is used to rewrite the record; a WRITE statement would attempt to create a new record with the same key, and this is an error. Control then returns to NEWCARD.

Card 56 contains a CLOSE statement that is executed only as a result of the action taken for an ENDFILE interrupt for the file PAYMNT. All files are closed and the program is terminated by the END statement in card 57.

PART II: RULES AND SYNTACTIC DESCRIPTIONS

Throughout this publication, wherever a PL/I statement -- or some other combination of elements -- is discussed, the manner of writing that statement or phrase is illustrated with a uniform system of notation.

This notation is not a part of PL/I; it is a standardized notation that may be used to describe the syntax -- or construction -- of any programming language. It provides a brief but precise explanation of the general patterns that the language permits. It does not describe the meaning of the language elements, merely their structure; that is, it indicates the order in which the elements may (or must) appear, the punctuation that is required, and the options that are allowed.

The following rules explain the use of this notation for any programming language; only the examples apply specifically to PL/I:

1. A notation variable is the name of a general class of elements in the programming language. A notation variable must consist of:

   a. Lower-case letters, decimal digits, and hyphens and must begin with a letter.

   b. A combination of lower-case and upper-case letters. There must be one portion all in lower-case letters and one portion all in upper-case letters, and the two portions must be separated by a hyphen.

   All such variables used are defined in the manual either syntactically, using this notation, or are defined semantically.

   Examples:

   a. digit. This denotes the occurrence of a digit, which may be 0 through 9 inclusive.

   b. file-name. This denotes the occurrence of the notation variable named file name. An explanation of file name is given elsewhere in the publication.

   c. DO-statement. This denotes the occurrence of a DO statement. the upper-case letters are used to indicate a language keyword.

2. A notation constant denotes the literal occurrence of the characters represented. A notation constant consists either of all capital letters or of a special character.

   Example:

   ```
   DECLARE identifier FIXED;
   ```

   This denotes the literal occurrence of the word DECLARE followed by the notation variable "identifier," which is defined elsewhere, followed by the literal occurrence of the word FIXED followed by the literal occurrence of the semicolon (;).

3. The term "syntactic unit," which is used in subsequent rules, is defined as one of the following:

   a. a single notation variable or notation constant, or

   b. any collection of notation variables, notation constants, syntax-language symbols, and keywords surrounded by braces or brackets.

4. Braces {} are used to denote grouping of more than one element into a syntactic unit.

   Example:

   ```
   identifier { FIXED }
              { FLOAT }
   ```

   The vertical stacking of syntactic units indicates that a choice is to be made. The above example indicates that the variable "identifier" must be followed by the literal occurrence of either the word FIXED or the word FLOAT.

5. The vertical stroke | indicates that a choice is to be made.

   Example:

   ```
   identifier {FIXED|FLOAT}
   ```

   This has exactly the same meaning as the above example. Both methods are used in this publication to display alternatives.

6. Square brackets [ ] denote options. Anything enclosed in brackets may appear one time or may not appear at all. Brackets can serve the additional purpose of delimiting a syntactic unit.

   Example:

   ```
   FILE(file-name) [KEY(expression)]
   ```

   This denotes the literal occurrence of the word FILE followed by the notation variable "file-name" enclosed in parentheses and optionally followed by the literal occurrence of the word KEY with its notation variable "expression" enclosed in parentheses. If, in rule 4, the two alternatives also were optional, the vertical stacking would be within brackets, and there would be no need for braces.

7. Three dots ... denote the occurrence of the immediately preceding syntactic unit one or more times in succession.

   Example:

   ```
   [digit] ...
   ```

   The variable "digit" may or may not occur since it is surrounded by brackets. If it does occur, it may be repeated one or more times.

8. Underlining is used to denote an element in the language being described when there is conflict between this element and one in the syntax language.

   Example:

   ```
   operand {&|⊥} operand
   ```

   This denotes that the two occurrences of the variable "operand" are separated by either an "and" (&) or an "or" (|). The notation constant | is underlined in order to distinguish the "or" symbol in the PL/I language from the "or" symbols in the syntax language.

60-CHARACTER SET

Character	Card-Punch	8-Bit Code
blank	no punches	0100 0000
.	12-8-3	0100 1011
<	12-8-4	0100 1100
(	12-8-5	0100 1101
+	12-8-6	0100 1110
|	12-8-7	0100 1111
&	12	0101 0000
$	11-8-3	0101 1011
*	11-8-4	0101 1100
)	11-8-5	0101 1101
;	11-8-6	0101 1110
¬	11-8-7	0101 1111
-	11	0110 0000
/	0-1	0110 0001
,	0-8-3	0110 1011
%	0-8-4	0110 1100
_	0-8-5	0110 1101
>	0-8-6	0110 1110
?	0-8-7	0110 1111
:	8-2	0111 1010
#	8-3	0111 1011
@	8-4	0111 1100
'	8-5	0111 1101
=	8-6	0111 1110
A	12-1	1100 0001
B	12-2	1100 0010
C	12-3	1100 0011
D	12-4	1100 0100
E	12-5	1100 0101
F	12-6	1100 0110
G	12-7	1100 0111
H	12-8	1100 1000
I	12-9	1100 1001
J	11-1	1101 0001
K	11-2	1101 0010
L	11-3	1101 0011
M	11-4	1101 0100
N	11-5	1101 0101
O	11-6	1101 0110
P	11-7	1101 0111
Q	11-8	1101 1000
R	11-9	1101 1001
S	0-2	1110 0010
T	0-3	1110 0011
U	0-4	1110 0100
V	0-5	1110 0101
W	0-6	1110 0110
X	0-7	1110 0111
Y	0-8	1110 1000
Z	0-9	1110 1001
0	0	1111 0000
1	1	1111 0001
2	2	1111 0010
3	3	1111 0011
4	4	1111 0100
5	5	1111 0101
6	6	1111 0110
7	7	1111 0111
8	8	1111 1000
9	9	1111 1001

Composite Symbols	Card Punch
<=	12-8-4, 8-6
||	12-8-7, 12-8-7
**	11-8-4, 11-8-4
¬<	11-8-7, 12-8-4
¬>	11-8-7, 0-8-6
¬=	11-8-7, 8-6
>=	0-8-6, 8-6
/*	0-1, 11-8-4
*/	11-8-4,0-1

48-CHARACTER SET

Character	Card-Punch	8-Bit Code
blank	no punches	0100 0000
.	12-8-3	0100 1011
(	12-8-5	0100 1101
+	12-8-6	0100 1110
$	11-8-3	0101 1011
*	11-8-4	0101 1100
)	11-8-5	0101 1101
-	11	0110 0000
/	0-1	0110 0001
,	0-8-3	0110 1011
'	8-5	0111 1101
=	8-6	0111 1110
A	12-1	1100 0001
B	12-2	1100 0010
C	12-3	1100 0011
D	12-4	1100 0100
E	12-5	1100 0101
F	12-6	1100 0110
G	12-7	1100 0111
H	12-8	1100 1000
I	12-9	1100 1001
J	11-1	1101 0001
K	11-2	1101 0010
L	11-3	1101 0011
M	11-4	1101 0100
N	11-5	1101 0101
O	11-6	1101 0110
P	11-7	1101 0111
Q	11-8	1101 1000
R	11-9	1101 1001
S	0-2	1110 0010
T	0-3	1110 0011
U	0-4	1110 0100
V	0-5	1110 0101
W	0-6	1110 0110
X	0-7	1110 0111
Y	0-8	1110 1000
Z	0-9	1110 1001
0	0	1111 0000
1	1	1111 0001
2	2	1111 0010
3	3	1111 0011
4	4	1111 0100
5	5	1111 0101
6	6	1111 0110
7	7	1111 0111
8	8	1111 1000
9	9	1111 1001

Composite Symbols	Card Punch	60-Character Set Equivalent
..	12-8-3, 12-8-3	:
LE	11-3, 12-5	<=
CAT	12-3, 12-1, 0-3	||
**	11-8-4, 11-8-4	**
NL	11-5, 11-3	¬<
NG	11-5, 12-7	¬>
NE	11-5, 12-5	¬=
,.	0-8-3, 12-8-3	;
AND	12-1, 11-5, 12-4	&
GE	12-7, 12-5	>=
GT	12-7, 0-3	<
LT	11-3, 0-3	>
NOT	11-5, 11-6, 0-3	¬
OR	11-6, 11-9	|
/*	0-1, 11-8-4	/*
*/	11-8-4, 0-1	*/

Note: when using the 48-character set, the following rules should be observed:

1. The two periods that replace the colon must be immediately preceded by a blank if the preceding character is a period.

2. The two slashes that replace the percent symbol must be immediately preceded by a blank if the preceding character is an asterisk, or immediately followed by a blank if the following character is an asterisk.

3. The sequence "comma period" represents a semicolon except when it occurs in a comment or character string, or when it is immediately followed by a digit.

Keyword	Use of Keyword
ABS(x)	built-in function
ADDR(x)	built-in function
ALIGNED	attribute
ALL(x)	built-in function
ANY(x)	built-in function
ATAN(x[,y])	built-in function
ATAND(x[,y])	built-in function
ATANH(x)	built-in function
AUTOMATIC	attribute
BACKWARDS	attribute
BASED (pointer-variable)	attribute
BEGIN	statement
BINARY	attribute
BINARY(x[,p[,q]])	built-in function
BIT(length)	attribute
BIT(value[,size])	built-in function
BOOL(x,y,w)	built-in function
BUFFERED	attribute
BUILTIN	attribute
BY	clause of DO statement
CALL	statement
CEIL(x)	built-in function
CHAR(value[,size])	built-in function
CHARACTER(length)	attribute
CLOSE	statement
COLUMN(w)	format item
CONVERSION	condition
COS(x)	built-in function
COSD(x)	built-in function
COSH(x)	built-in function
DATE	built-in function
DECIMAL	attribute
DECIMAL(x[,p[,q]])	built-in function
DECLARE	statement
DEFINED	attribute
DIRECT	attribute
DISPLAY	statement
DO	statement
EDIT	STREAM I/O transmission mode
ELSE	clause of IF statement
END	statement
ENDFILE	condition
ENDPAGE	condition
ENTRY	attribute or statement
ENVIRONMENT	attribute
ERF(x)	built-in function
ERFC(x)	built-in function
ERROR	condition
EXP(x)	built-in function
EXTERNAL	attribute
FILE	attribute
FILE(file-name)	option of GET and PUT, specification of RECORD I/O statement
FIXED	attribute
FIXED(x[,p[,q]])	built-in function
FIXEDOVERFLOW	condition
FLOAT	attribute
FLOAT(x[,p])	built-in function
FLOOR(x)	built-in function
FORMAT(format-list)	statement
FROM	option of REWRITE or WRITE statement
GET	statement
GO TO,GOTO	statement
HIGH(i)	built-function

Keyword	Use of Keyword
IF	statement
INDEX(string,config)	built-in function
INPUT	attribute, option of the OPEN statement
INTERNAL	attribute
INTO(variable)	option of READ statement
KEY(file-name)	condition
KEY(x)	option of READ and REWRITE statement
KEYED	attribute
KEYFROM(x)	option of WRITE and LOCATE statement
LABEL	attribute
LINE(w)	format item, option of PUT statement
LOCATE	statement
LOG(x)	built-in function
LOG2(x)	built-in function
LOG10(x)	built-in function
LOW(i)	built-in function
MAIN	option of PROCEDURE statement
MAX(arguments)	built-in function
MIN(arguments)	built-in function
MOD(x_1,x_2)	built-in function
NOCONVERSION	condition prefix identifier, disables CONVERSION
NOFIXEDOVERFLOW	condition prefix identifier, disables FIXEDOVERFLOW
NOOVERFLOW	condition prefix identifier, disables OVERFLOW
NOSIZE	condition prefix identifier, disables SIZE
NOUNDERFLOW	condition prefix identifier, disables UNDERFLOW
NOZERODIVIDE	condition prefix identifier, disables ZERODIVIDE
NULL	built-in function
ON	statement
ONSYSLOG	option of PROCEDURE statement
OPEN	statement
OPTIONS(list)	option of PROCEDURE statement
OUTPUT	attribute, option of the OPEN statement
OVERFLOW	condition
PACKED	attribute
PAGE	format item, option of PUT statement
PAGESIZE(w)	option of the OPEN statement
PICTURE	attribute
POINTER	attribute
PRECISION(x,p[,g])	built-in function
PRINT	attribute
PROCEDURE	statement
PROD(x)	built-in function
PUT	statement
READ	statement
RECORD	attribute
RECORD(file-name)	condition
REPEAT(string,i)	built-in function
REPLY(c)	option of DISPLAY statement
RETURN	statement
RETURNS	attribute
REVERT	statement
REWRITE	statement
ROUND(x,n)	built-in function
SEQUENTIAL	attribute
SET	option of READ and LOCATE statements
SIGN(x)	built-in function
SIGNAL	statement
SIN(x)	built-in function
SIND(x)	built-in function
SINH(x)	built-in function
SIZE	condition
SKIP[(x)]	format item, option of PUT statement
SQRT(x)	built-in function
STATIC	attribute
STOP	statement
STREAM	attribute

Keyword	Use of Keyword
STRING(x)	built-in function
STRING(string-name)	option of GET and PUT statements
SUBSTR(string,i,j)	built-in function, pseudo-variable
SUM(x)	built-in function
SYSTEM	action specification of the ON statement
TAN(x)	built-in function
TAND(x)	built-in function
TANH(x)	built-in function
THEN	clause of IF statement
TIME	built-in function
TO	clause of DO statement
TRANSMIT	condition
TRUNC(x)	built-in function
UNBUFFERED	attribute
UNDERFLOW	condition
UNSPEC(x)	built-in function, pseudo-variable
UPDATE	attribute
WHILE	clause of DO statement
WRITE	statement
ZERODIVIDE	condition

SECTION D: PICTURE SPECIFICATION CHARACTERS

Picture specification characters appear in a PICTURE attribute. They are used to describe the attributes of the associated data item. A discussion of the concepts of picture specifications appears in Part I, Chapter 9, "Editing and String Handling."

A picture specification always describes a character representation that is either a character-string data item or a numeric character data item. A character-string pictured item is one that can consist of alphabetic characters, decimal digits, and other special characters. A numeric character pictured item is one in which the data itself can consist only of decimal digits, a decimal point and, optionally, a plus or minus sign. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified. However, these characters are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are considered to be part of the character-string value of the variable.

Arithmetic data assigned to a numeric character variable is converted to numeric character representation. Editing, such as zero suppression and the insertion of other characters, can be specified for a numeric character data item. Editing cannot be specified for pictured character-string data.

Data assigned to a variable declared with a numeric picture specification must be internal coded arithmetic data (bit strings and numeric character data are converted to internal coded arithmetic before they are assigned to a numeric character variable).

Figures in this section illustrate how different picture specifications affect the representation of values when assigned to a pictured variable. Each figure shows the original value of the data, the attributes of the variable from which it is assigned, the picture specification, and the character-string value of the numeric character or pictured character-string variable.

PICTURE CHARACTERS FOR CHARACTER-STRING DATA

Only the X picture character can be used to specify character-string items. It specifies that the associated position within the item can contain any character whose internal bit configuration can be recognized by the computer in use. No characters can be specified for insertion into a picture character-string item.

Figure D-1 gives examples of character-string picture specifications. In the figure, the letter b indicates a blank character. Note that assignments are left-adjusted, and any necessary padding with blanks is on the right.

PICTURE CHARACTERS FOR NUMERIC CHARACTER SPECIFICATIONS

Numeric character data must represent numeric values; therefore, the associated picture specification cannot contain the character X. The picture characters for numeric character data can specify detailed editing of the data.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
CHARACTER(5)	'9B/2L'	XXXXX	9B/2L
CHARACTER(5)	'9B/2L'	XXX	9B/
CHARACTER(5)	'9B/2L'	XXXXXXX	9B/2Lbb

[1]A variable declared with a character-string picture specification has a character-string value only.

Figure D-1. Pictured Character-String Examples

A numeric character variable can be considered to have two different kinds of value, depending upon its use. They are (1) its arithmetic value and (2) its character-string value.

The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point, and possibly a sign. The arithmetic value of a numeric character variable is used whenever the variable appears in an arithmetic or bit-string expression operation or in an assignment to a variable with either the FIXED, FLOAT, or BIT attribute. In such cases, the arithmetic value of the numeric character variable is converted to internal coded arithmetic representation. The arithmetic value is also used in an assignment to another numeric character variable.

The character-string 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-string value does not, however, include the assumed location of a decimal point, as specified by the picture character V. The character-string value of a numeric character variable is used whenever the variable appears in a character-string expression operation or in an assignment to a character-string variable, or whenever a reference is made to a character-string variable that is defined on the numeric character variable.

The picture characters for numeric character specifications may be grouped into the following categories:

- Digit and Decimal Point Specifiers
- Zero Suppression Characters
- Insertion Characters
- Numeric Signs and Currency Symbol
- Credit, Debit, and Overpunched Signs
- Exponent Specifiers
- Sterling Pictures

The picture characters in these groups can be used in various combinations. These combinations depend on the type of data being described by the specification. A discussion of these types and how they can be described follows.

A numeric character picture specification can describe either decimal or sterling data. Decimal numeric character values can be in fixed-point or floating-point. The numeric character picture specification for a fixed-point value contains only one field and this field can consist of two subfields: an integer subfield describing the digits to the left of the decimal point in the fixed-point value, and a fractional subfield describing the digits to the right of the decimal point.

The numeric character picture specification for a floating-point value consists of two fields: a mantissa field and an exponent field. The mantissa field describes a fixed-point value, which when multiplied by 10 raised to the value described by the exponent field gives the actual value represented by the floating-point notation; the mantissa field is specified in the same way that a fixed-point field is specified. The exponent field describes a signed or unsigned integer power of ten.

The sterling picture specification can contain up to three fields: a pounds field, a shillings field, and a pence field; the pence field can have two subfields. Sterling pictures are discussed separately at the end of this section.

A major 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. At least one of these characters must be used to define a numeric character specification. It cannot contain the picture character X.

DIGIT AND DECIMAL POINT SPECIFIERS

The picture characters 9 and V are used in the simplest form of numeric character specifications that represent fixed-point decimal values.

9 specifies that the associated field position is to contain a decimal digit.

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 is to be inserted. The integer and fractional parts of the assigned value are aligned on the V character; therefore, an assigned value may be truncated or extended with zero digits at either end. Note that if significant digits are truncated on the left, the result is undefined and a SIZE interrupt will

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5)	12345	99999	12345
FIXED(5)	12345	99999V	12345
FIXED(5)	12345	999V99	34500[2]
FIXED(5)	12345	V99999	00000[2]
FIXED(7)	1234567	99999	34567[2]
FIXED(3)	123	99999	00123
FIXED(5,2)	123.45	999V99	12345
FIXED(7,2)	12345.67	9V9	56[2]
FIXED(5,2)	123.45	99999	00123

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.
[2]In this case, PL/I does not define the result since significant digits have been truncated on the left; the result shown, however, is that given for System/360 implementations.

Figure D-2. Pictured Numeric Character Examples

occur, if SIZE is enabled. If no V character appears in the picture specification of a fixed-point decimal value (or in the mantissa 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. The V is considered to be a subfield delimiter in the picture specification; that is, the portion preceding the V and the portion following it (if any) are each a subfield of the specification.

Figure D-2 gives examples of numeric character specifications.

ZERO SUPPRESSION CHARACTERS

The zero suppression picture characters specify conditional digit positions in the character-string value and may cause leading zeros to be replaced by asterisks or blanks. Leading zeros are those (1) 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, (2) that are to the left of the assumed position of a decimal point, and (3) 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. Note that a floating-point number can also have a leading zero in the exponent field.

Z specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank character. When the associated data position does not contain a leading zero, the digit in the position is not replaced by a blank character. The picture character Z cannot appear in the same subfield as the picture character *, nor can it appear to the right of a drifting picture character or any of the picture characters 9, T, I, or R in a field.

* specifies a conditional digit position and is used the way the picture character Z is used, except that leading zeros are replaced by asterisks. The picture character * cannot appear with the picture character Z in the same subfield, nor can it appear to the right of a drifting picture character or any of the picture characters 9, T, I, or R in a field.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5)	12345	ZZZ99	12345
FIXED(5)	00100	ZZZ99	bb100
FIXED(5)	00000	ZZZ99	bbb00
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	34500[2]
FIXED(5)	00000	ZZZVZZ	bbbbb
FIXED(5)	00100	*****	**100
FIXED(5)	00000	*****	*****
FIXED(5,2)	000.01	***V**	***01

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.
[2]In this case, PL/I does not define the result since significant digits have been truncated on the left; the result shown, however, is that given for System/360 implementations.

Figure D-3. Examples of Zero Suppression

Note: If one of the picture characters Z or * appears to the right of the picture character V, then all fractional digit positions in the specification, as well as all integer digit positions must employ the Z or * 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 will be suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The entire character-string value of the data item will then consist of blanks or asterisks. No digits in the fractional part will be replaced by blanks or asterisks if the fractional part contains any significant digit.

Figure D-3 gives examples of the use of zero suppression characters. In the figure, the letter b indicates a blank character.

INSERTION CHARACTERS

The picture characters comma (,), point (.), and blank (B) are insertion characters; they cause the specified character to be inserted into the associated position of the numeric character data. They do not indicate digit positions, but are inserted between digits. Each does, however, actually represent a character position in the character-string value, whether or not the character is suppressed. The comma and point are conditional insertion characters; within a string of zero suppression characters, they, too, may be suppressed. The blank (B) is an unconditional insertion character; it specifies that a blank is to appear in the associated position.

Note: Insertion characters are applicable only to the character-string value. They specify nothing about the arithmetic value of the data item.

causes a comma to be inserted into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur,

the comma is inserted only when an unsuppressed digit appears to the left of the comma position, or when a V appears immediately to the left of it and the fractional part contains any significant digits. In all other cases where zero suppression occurs, one of three possible characters is inserted in place of the comma. The choice of character to replace the comma depends upon the first picture character that both precedes the comma position and specifies a digit position:

- If this character position is an asterisk, the comma position is assigned an asterisk.
- If this character position is a drifting sign or a drifting currency symbol (discussed later), the drifting string is assumed to include the comma position, and the action taken is the same as that for drifting characters.
- If this character position is not an asterisk or a drifting character, the comma position is assigned a blank character.

is used the same way the comma picture character is used, except that a point (.) is assigned to the associated position. This character never causes point alignment in the picture specifications of a fixed-point decimal number and is not a part of the arithmetic value of the data item. That function is served solely by the picture character V. Unless the V actually appears, it is assumed to be to the right of the rightmost digit position in the field, and point alignment is handled accordingly, even if the point insertion character appears elsewhere.

B specifies that a blank character be inserted into the associated position of the character-string value of the numeric character field.

The point (or the comma) can be used in conjunction with the V to cause insertion of the point (or comma) in the position that delimits the end of the integer portion 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. In this case, the point must immediately precede or immediately follow the V. If the point precedes the V, it will be inserted only if a significant digit appears to the left of the V, even if all fractional digits are significant. If the point immediately follows the V, it will be suppressed if all digits to the right of the V are suppressed, but it will appear if there are any fractional digits (along with any intervening zeros).

The insertion characters B, comma, and point must be preceded by a digit position in the same field.

Figure D-4 gives examples of the use of insertion characters. In the figure, the letter b indicates a blank character.

SIGNS AND CURRENCY SYMBOL

The picture characters S, +, and - specify signs in numeric character data. The picture character $ specifies a currency symbol in the character-string value of numeric character data.

These picture characters may be used in either a static or a drifting manner. A drifting character is similar to a zero suppression character in that it can cause zero suppression. However, a single drifting character is always inserted (unless the entire field is suppressed) in the position specified by the end of the drifting string or in the position immediately to the left of the first significant digit.

The static use of these characters specifies that a sign, a currency symbol, or a blank always appears in the associated position. The 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.

A drifting character is specified by multiple use of that character in a picture field. Thus, if a field contains one currency symbol, it is interpreted as static; if it contains more than one, it is interpreted as drifting. The drifting character must be specified in each digit position through which it may drift.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing a V and one of the insertion characters comma, point, or B. Any of the insertion characters comma, point, or B following the last drifting symbol of the string is considered part of the drifting string. However, a following V terminates the drifting string and is not part of it. A field of a picture specification can contain only one drifting string. A drifting string cannot be preceded by a digit position, insertion characters, or a V. If

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
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(4,2)	67.89	9,999,999.V99	0,000,067.89
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	99.999.9	12.345.6
FIXED(6)	001234	ZZ,ZZ,ZZ	bbb12,34
FIXED(6)	000000	ZZ,ZZ,ZZ	bbbbbbbb
FIXED(6)	000000	**,**,**	********
FIXED(6)	123456	99B99B99	12b34b56
FIXED(3)	123	9BB9BB9	1bb2bb3

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-4. Examples of Insertion Characters

a drifting string exists in a field, zero suppression character (Z or *) must not appear in the same field.

The position in the data associated with the characters comma, point, and B appearing in a string of drifting characters will contain one of the following:

- Comma, point, or blank if a significant digit has appeared 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, then 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. If a drifting string is specified for a field, the other potentially drifting characters can appear only once in the field, i.e., the other character represents a static sign or currency symbol.

Only one type of sign character can appear in each field. An S, +, or - used as a static character can appear to the left of all digits in the mantissa and exponent fields of a floating-point specification and either to the right or left of all digits positions of a fixed-point specification.

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, suppresion in the subfield will occur only if all of the integer and fractional digits are zero. The resulting edited data item will then be all blanks. If there are any significant fractional digits, the entire fractional portion will appear unsuppressed.

$ specifies the currency symbol. If this character appears more than once, it is a drifting character; otherwise it is a static character. The static character specifies that the character is to be placed in the associated position. The static character must appear either to the left of all digit positions in a field of a specification or to the right of all digit positions in a specification. See details above for the drifting use of the character.

S specifies the plus sign character (+) if the data value is ≥0, otherwise it specifies the minus sign character (-). The character may be drifting or static. The rules are identical to those for the currency symbol.

+ specifies the plus sign character (+) if the data value is ≥ 0, otherwise it specifies a blank. The character may be drifting or static. 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 character may be drifting or static. The rules are identical to those for the currency symbol.

Figure D-5 gives examples of the use of drifting picture characters. In the figure, the letter b indicates a blank character.

CREDIT, DEBIT, AND OVERPUNCHED SIGNS

The character pairs CR (credit) and DB (debit) specify the signs of fixed-point numeric character data items and usually appear in business report forms.

Any of the picture characters T, I, or R specifies an overpunched sign in the associated digit position of a fixed-point numeric character data item. An overpunched sign is a 12-punch (for plus) or an 11-punch (for minus) punched into the same column as a digit. It indicates the sign of the arithmetic data item. Only one overpunched sign can appear in a specification for a fixed-point number. The overpunch character can appear only in the last digit position within a field.

CR specifies that the associated positions will contain the letters CR if the value of the data is less than zero. Otherwise, the positions will contain two blanks. The characters CR can appear only to the right of all digit positions of a field.

DB is used the same way that CR is used except that the letters DB appear in the associated positions.

T specifies that the associated position, on input, will contain a digit overpunched with the sign of the data. It also specifies that, an overpunch is to be indicated in the character-string value.

I specifies that the associated position, on input, will contain a digit overpunched with + if the value is ≥0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is ≥0.

R specifies that the associated position, on input, will contain a digit overpunched with - if the value is <0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is <0.

Note: The picture characters CR, DB, T, I, and R cannot be used with any other sign characters in the same field.

Figure D-6 gives examples of the CR, DB, and overpunch characters. In the figure, the letter b indicates a blank character.

EXPONENT SPECIFIERS

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 always the last field of a numeric character floating-point picture specification. The picture characters K and E cannot appear in the same specification.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5,2)	123.45	$999V.99	$123.45
FIXED(5,2)	001.23	$ZZZV.99	$bb1.23
FIXED(5,2)	000.00	$ZZZV.ZZ	bbbbbbb
FIXED(5,2)	123.45	$$$9V.99	$123.45
FIXED(5,2)	001.23	$$$9V.99	bb$1.23
FIXED(5,2)	012.00	99$	12$
FIXED(2)	12	$$$,999	bbb$012
FIXED(4)	1234	$$$,999	b$1,234
FIXED(5,2)	123.45	S999V.99	+123.45
FIXED(5,2)	-123.45	S999V.99	-123.45
FIXED(5,2)	-123.45	+999V.99	b123.45
FIXED(5,2)	123.45	-999V.99	b123.45
FIXED(5,2)	123.45	999V.99S	123.45+
FIXED(5,2)	001.23	+++B+V.99	bbb+1.23
FIXED(5,2)	001.23	---9V.99	bbb1.23
FIXED(5,2)	-001.23	SSS9V.99	bb-1.23

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-5. Examples of Drifting Picture Characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
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	999T	102A

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-6. Examples of CR, DB, T, I, and R Picture Characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
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

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-7. Examples of Floating-Point Picture Specifications

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-string 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).

Figure D-7 gives examples of the use of exponent delimiters. In the figure, the letter b indicates a blank character.

STERLING PICTURES

The following picture characters are used in picture specifications for sterling data:

8 specifies the position of a shilling digit in BSI single-character representation. Ten shillings is represented by a 12-punch (&) and eleven through nineteen shillings are represented by the characters A through I, respectively.

7 specifies the position of a pence digit in BSI single-character representation. Tenpence is represented by a 12-punch (&) and elevenpence is represented by an 11-punch (-).

6 specifies the position of a pence digit in IBM single-character representation. Tenpence is represented by an 11-punch (-) and elevenpence is represented by a 12-punch (&).

P specifies that the associated position contains the pence character D.

G specifies the start of a sterling picture. It does not specify a character in the numeric character data item.

H specifies that the associated position contains the shilling character S.

M specifies the start of a field. It does not specify a character in the numeric character data item.

Sterling data items are considered to be fixed-point decimal data items. When involved in arithmetic operations, they are converted to a value representing fixed-point pence. Sterling pictures have the general form:

```
PICTURE
    'G [editing-character-1] ...
    M pounds-field
    M [separator-1] ...
          shillings-field
    M [separator-2] ...
          pence-field
      [editing-character-2] ...'
```

"Editing character 1" can be one or more of the following static picture characters:

$ + - S

The "pounds field" can contain the following picture characters:

Z * 9 , $ + - S

The last four characters ($ + - S) must be drifting characters. The comma can be used as an insertion character.

"Separator 1" can be one or more of the following picture characters:

/ . B

The "shillings field" can be:

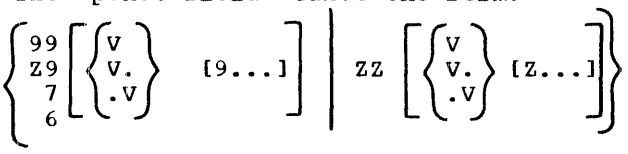
{99 | ZZ | Z9|8}

The picture character Z can occur only if the entire field to the left of this character (including the pounds field) has no digit position other than Z.

"Separator 2" can be one or more of the picture characters:

/ . B H

The "pence field" takes the form:



The final 9 can be replaced by one of the following:

T I R

"Editing character 2" can be

1. a $ and/or a P, or
2. a $ and/or a P, mixed with one or more B characters, or
3. one of CR DB + - S in combination with either of the above configurations.

The pounds, shillings, and pence fields must each contain at least one digit position.

Zero suppression in sterling pictures is performed on the total specification, not separately on each of the fields. Separator characters slash(/), point(.), B, and H are never suppressed. For a single sterling specification, there can be a maximum of one sign. This sign can be specified by "editing character 1," by T, I, or R in the pence field, by "editing character 2", or by a drifting string in the pounds field.

Figure D-8 gives examples of the use of sterling picture specifications.

Source Attributes	Source Data (stated in pence)	Picture Specification	Character-String Value[1]
FIXED(4)	0534	GMZ9M.8M.99V.9CR	b2.4.06.0bb
FIXED(4)	0019	GMZZM.ZZM.ZZP	bb.b1.07D

[1]The arithmetic value of a numeric character variable declared with a sterling picture specification is its value expressed as a valid sterling fixed-point constant, which for arithmetic operations is always converted to its value expressed in pence.

Figure D-8. Examples of Sterling Picture Specifications

SECTION E: EDIT-DIRECTED FORMAT ITEMS

This section describes each of the edit-directed format items that can appear in the format list of a GET or PUT statement.

There are four categories of format items: data format items, printing format items, the spacing format item, and the remote format item.

In this section, the four categories are discussed separately and the format items are listed under each category. The remainder of the section contains detailed discussions of each of the format items, with the discussions appearing in alphabetic order.

DATA FORMAT ITEMS

A data format item describes the external format of a single data item.

For input, the data in the stream is considered to be a continuous string of characters; all blanks are treated as characters in the stream, as are quotation marks. Each data format item in a GET statement specifies the number of characters to be obtained from the stream and describes the way those characters are to be interpreted. Strings should not be enclosed in quotation marks, nor should the letter B be used to identify bit strings.

For output, the data in the stream takes the form specified by the format list. Each data format item in a PUT statement specifies the width of a field into which the associated data item in character form is to be placed and describes the format that the value is to take. Enclosing quotation marks are not inserted, nor is the letter B to identify bit strings.

Leading blanks are not inserted automatically to separate data items in the output stream. String data is left-adjusted in the field whose width is specified. Arithmetic data is right-adjusted. Leading blanks will not appear in the stream unless the specified field width allows for them. Truncation, due to inadequate field-width specification is on the left for arithmetic items, on the right for string items.

Note that the value of binary data both on input and output is always represented in decimal form for edit-directed transmission.

Following is a list of data format items:

Fixed-point format item	F(specification)
Floating-point format item	E(specification)
Bit-string format item	B(specification)
Character-string format item	A(specification)

PRINTING FORMAT ITEMS

The printing format items apply only to output and only to files with the PRINT attribute. They specify formatting of the printed page.

Following is a list of printing format items:

Paging format item	PAGE
Line skipping format item	SKIP[(specification)]
Line position format item	LINE(specification)
Column position format item	COLUMN(specification)

A printing format item has no effect unless it is encountered before the data list is exhausted.

The PAGE, SKIP, and LINE format items have the same effect as the corresponding options of the PUT statement, except that the format items are executed only when they are encountered in the format list, while the options of the PUT statement are executed before any data is transmitted.

SPACING FORMAT ITEM

The spacing format item specifies relative horizontal spacing. On input, it specifies a number of characters in the stream to be skipped over and ignored.

On output, it specifies a number of blanks to be inserted into the stream.

The spacing format item is:

X(specification)

The spacing format item has no effect unless it is encountered before the data list is exhausted.

REMOTE FORMAT ITEM

The remote format item specifies the label of a FORMAT statement that contains a format list which is to be taken to replace the remote format item.

The remote format item is:

R(statement-label-designator)

The "statement label designator" is a label constant or an unsubscripted element label variable.

USE OF FORMAT ITEMS

The "specification" that is listed above for all but the PAGE and remote format items can contain one or more expressions. Such expressions must be decimal integer constants.

ALPHABETIC LIST OF FORMAT ITEMS

The A Format Item

The A format item is:

A [(field-width)]

The character-string format item describes the external representation of a string of characters. It must be used only for character strings. Character strings cannot be transmitted by any other format item. No conversion is performed.

General rules:

1. The "field width" (sometimes expressed as w) must be a decimal integer constant, unsigned and greater than zero, but less than 256. It specifies the number of character positions in the data stream that contain (or will contain) the string.
2. On input, the specified number of characters is obtained from the data stream and assigned to the associated element in the data list. The field width is always required on input. If quotation marks appear in the stream, they are treated as characters in the string.
3. On output, the field width need not be specified; in this case, the length of the associated string is used, and the data item completely fills the field. Enclosing quotation marks are never inserted.

The B Format Item

The B format item is:

B [(field-width)]

The bit-string format item describes the external representation of a bit string. Each bit is represented by the character 0 or 1. This format item can be used only for bit strings; bit strings cannot be transmitted by any other format item.

General rules:

1. The "field width" (sometimes expressed as w) must be an unsigned decimal integer constant greater than zero and less than 65. It specifies the number of data-stream character positions that contain (or will contain) the bit string.
2. On input, the character representation of the bit string may occur anywhere within the specified field. Blanks, which may appear before and after the bit string in the field are ignored. The field width is always required on input. Any character other than 0 or 1 (including embedded blanks, quotation marks, or the letter B) will raise the CONVERSION condition.
3. 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. No quotation marks are inserted, nor is the identifying letter B. If the field width is not specified, the declared length of the associated string is used, and the data item completely fills the field.

The COLUMN Format Item

The COLUMN format item is:

COLUMN (character-position)

The column position format item controls the spacing of a data item to a specified character position within the line. It can be used only with a PRINT file and, consequently, it can appear only in a PUT statement.

General rules:

1. The "character position" (sometimes expressed as w) must be a decimal integer constant greater than zero and less than 256.

2. Blank characters are placed into the data stream so that the next field will begin at the specified character position of the current line. If data has already been placed into the specified character position or beyond, the current line is completed, and a new line is started. Blank characters are then inserted into the data stream so that the next field will begin at the specified character position of the new line.

3. If the specified character position lies beyond the rightmost character position of the current line (i.e., if w is greater than the line size), then the character position is assumed to be one.

4. The COLUMN format item has no effect unless it is encountered before the data list is exhausted.

The E Format Item

The E format item is:

E(field-width,number-of-fractional-digits
[,number-of-significant-digits])

The floating-point format item describes the external representation of decimal arithmetic data in floating-point format.

General rules:

1. The "field width," "number of fractional digits," and "number of significant digits" (sometimes referred to as w, d, and s, respectively) must be unsigned decimal integer constants. The field width must be less than 33.

   "Field width" specifies the total number of characters in the field.

   "Number of fractional digits" specifies the number of digits to appear following the decimal point in the mantissa.

   "Number of significant digits" specifies the number of digits that must appear in the mantissa.

2. On input, the data item in the data stream is the character representation of an optionally signed decimal floating-point or fixed-point constant located anywhere within the specified field. If the data item is a fixed-point number, an exponent of zero is assumed.

   The external form of the number is:

   $$[\pm]\text{mantissa}\left[\left\{\begin{matrix}[E]\{+|-\}\\ E[+|-]\end{matrix}\right\}\text{exponent}\right]$$

   The mantissa must be a decimal fixed-point constant.

   a. The number can appear anywhere within the specified field; blanks may appear before and after the number in the field. If the entire field is blank, the CONVERSION condition is raised. When no decimal point appears, the number of fractional digits (d) specifies the number of character positions in that part of the mantissa to the right of the assumed decimal point. If a decimal point actually does appear in the data, it overrides the number of the fractional digits specification.

      The value expressed by "field width" includes trailing blanks, the exponent position, the position for the optional plus or minus sign, and the position for the optional letter E and the position for the optional decimal point in the mantissa.

   b. The exponent is a decimal integer constant that cannot exceed three digits. Whenever the exponent and preceding sign or letter E are omitted, a zero exponent is assumed.

   c. The sign of the mantissa must always be accounted for in the field width, even if it is positive and is represented by a blank.

3. On output, the internal data is converted to floating-point, and the external data item in the specified field has the following general form:

```
[-]{s-d digits}.{d digits}
    E{+|-}exponent
```

   a. The exponent is a two-digit decimal integer constant, which may be two zeros. The exponent is automatically adjusted so that the leading digit of the mantissa is nonzero (provided that the mantissa is not zero, of course).

   b. If the above form of the number does not fill the specified field on output, the number is right-adjusted and extended on the left with blanks. If the number of significant digits is not specified, it is taken to be 1 plus the number of fractional digits. For the D-compiler, the field width for negative or non-negative values of the data item must be greater than or equal to 6 plus the number of significant digits (although the sign of a positive value is not written, it must be accounted for). However, if the number of fractional digits is zero, the decimal point is not written, and the above figure for the field width is reduced by 1.

The F Format Item

The F format item is:

```
F(field-width[,number-of-fractional-digits
                [,scaling-factor]])
```

The fixed-point format item describes the external representation of a decimal arithmetic data item in fixed-point format.

General rules:

1. The "field width," "number of fractional digits," and "scaling factor" (sometimes expressed as w, d, and p, respectively) must be decimal integer constants. Only p can be signed; the others must be unsigned; w must be less than 33 and must account for the sign, even if it is blank.)

2. On input, the data item in the data stream is the character representation of an optionally signed decimal fixed-point constant located anywhere within the specified field. Blanks may appear before and after the number in the field. If the entire field is blank, it is interpreted as zero.

   The number of fractional digits, if not specified, is assumed to be zero.

   If no scaling factor is specified and no decimal point appears in the field, the number of fractional digits specifies the number of digits in the field to the right of the assumed decimal point. If a decimal point actually does appear in the data, it overrides the specification for the number of fractional digits.

   If a scaling factor is specified, it effectively multiplies the value of the data item in the data stream by 10 raised to the value of the scaling factor (i.e., p). Thus, if p is positive, the number is treated as though the decimal point appeared p places to the right of its given position. If p is negative, the number 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 specification for the number of fractional digits, in the absence of an actual point.

3. On output, the internal data is converted, if necessary, to fixed-point, and the external data is the character representation of a decimal fixed-point number, right-adjusted in the specified field.

   If only the field width is specified in the format item, only the integer portion of the number is written; no decimal point appears.

   If both the field width and number of fractional digits are specified, but the scale factor is not, both the integer and fractional portions of the number are written and a decimal point is inserted before the rightmost d digits. Trailing zeros are supplied when the actual number of fractional digits is less than d (the value d must be less than the field width). Suppression of leading zeros is applied to all digit positions to the left of the decimal point.

   The value of the scaling factor effectively multiplies the value of the associated element in the data list by 10 raised to the power of p, before it is edited into its external character representation. When the number of fractional digits is zero, only the integer portion of the number is used.

For all options on output, if the value of the fixed-point number is less than zero, a minus sign is prefixed to the external character representation; if it is greater than zero, a blank appears. Therefore, for all values of the fixed-point number, the field width specification must include a count of both the sign and possibly the decimal point (since the decimal point will not appear if there are no fractional digits).

If the field width is such that significant digits or the sign is lost, the SIZE condition is raised.

The LINE Format Item

The LINE format item is:

LINE (line-number)

The line position format item specifies the particular line on a page of a PRINT file upon which the next data item is to be printed.

General rules:

1. The "line number" (sometimes expressed as w) must be an unsigned decimal integer constant less than 256.
2. The LINE format item specifies that blank lines are to be inserted so that the next line will be the specified line of the current page.
3. If the specified line has already been passed on the current page, or if the specified line is beyond the limit set by default or by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised.
4. If "line number" is equal to zero, it is assumed to be one.
5. The LINE format item has no effect unless it is encountered before the data list is exhausted.

The PAGE Format Item

The PAGE format item is:

PAGE

The paging format item specifies that a new page is to be established.

General rules:

1. The establishment of a new page implies that the next printing is to be on line one.
2. The PAGE format item has no effect unless it is encountered before the data list is exhausted.

The R Format Item

The R format item is:

R (statement-label-designator)

The remote format item allows format items in a FORMAT statement to replace the remote format item.

General rules:

1. The "statement label designator" is a label constant or an element label variable that has as its value the statement label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the format item. The "statement label designator" cannot be subscripted.
2. The R format item and the specified FORMAT statement must be internal to the same block.
3. A FORMAT statement cannot contain an R format item.

The SKIP Format Item

The SKIP format item is:

SKIP[(relative-position-of-next-line)]

The line skipping format item specifies that a new line is to be defined as the current line.

General rules:

1. The "relative position of next line" (sometimes expressed as w) must be an unsigned decimal integer constant between 0 and 3 inclusive. If it is omitted, 1 is assumed.
2. The new line is the specified number of lines beyond the present line.
3. If w is greater than or equal to one, w-1 blank lines will be inserted.

4. If the value of the relative position is zero, the effect is that of a carriage return without line spacing. Characters previously written will be overprinted by the new characters. For example, underscoring can be done.

5. If the SKIP format item is not specified at the end of a line, then SKIP (1) is assumed, that is, single spacing.

6. If the specified line lies beyond the limit set by default or by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised.

7. The SKIP format item has no effect unless it is encountered before the data list is exhausted.

The X Format Item

The X format item is:

X (field-width)

The spacing format item controls the relative spacing of data items in the data stream. It is not limited to PRINT files.

General rules:

1. The "field width" (sometimes expressed as w) must be an unsigned decimal integer constant less than 256. It specifies the number of blanks before the next field of the data stream, relative to the current position in the stream.

2. On input, the specified number of characters is spaced over in the data stream and not transmitted to the program.

3. On output, the specified number of blank characters are inserted into the stream.

4. The spacing format item has no effect unless it is encountered before the data list is exhausted.

This section lists the rules for arithmetic conversion and for conversion of data types. Each type conversion is listed under a separate heading. In addition to the text, seven tables appear:

- Table F-1 states the rules for computing the precision of the result of an arithmetic conversion.
- Table F-2 is a table that can be used to find the length of the result of an arithmetic to bit-string conversion.
- Table F-3 can be used to find the ceiling (CEIL) of any value between 1 and 15 when that value is multiplied by 3.32 or it can be used to find the ceiling (CEIL) of any value between 1 and 56 when that value is divided by 3.32.
- Tables F-4 through F-7 illustrate conversion in arithmetic expression operations, and they give attributes of the results based upon the operator specified and the attributes of the two operands.

ARITHMETIC CONVERSION

The rules for arithmetic conversion specify the way in which a value is transformed from one arithmetic representation to another. It can be that as a result of the transformation the value will change. For example, the number .2, which can be exactly represented as a decimal fixed-point number, cannot be exactly represented in binary. The magnitude of such changes in value depends upon the precisions of the target and source. In expression evaluation, the precision of the target is derived from the precision of the source. In order to estimate and to understand the errors that can occur, the precision rules must be understood; and since the rules also leave some latitude for the implementation, it is helpful to have some knowledge of the way in which conversions are implemented.

Floating-Point Conversion

In System/360 implementations, both decimal and binary floating-point numbers are maintained in the internal hexadecimal form used in System/360. If the specified precision is more than 6 decimal digits, or 21 binary digits, the number is maintained in long floating-point form (14 hexadecimal digits with a hexadecimal exponent). If the precision is 6 decimal digits or less, or 21 binary digits or less, the number is maintained in short floating-point form (6 hexadecimal digits and a hexadecimal exponent).

No actual conversions between binary and decimal are performed on floating-point data. The only precision changes are from long to short, which is done by truncation, and from short to long, which is done by extending with zeros. The declared precision of floating-point data and the base, however, do affect the calculation of target attributes, as well as the attributes of intermediate forms that are determined from the source.

Precision Conversion

Precision conversion occurs if the specified target precision is different from the source precision. In particular, there always is a precision change when the source and target are of different bases. It is also possible that there is an actual change in precision when converting from floating-point to fixed-point, because of the way in which floating-point numbers are represented. Precision changes are performed by truncation or by padding with zeros. Floating-point numbers are converted from short precision to long precision by extending with zeros on the right, and from long precision to short precision by truncation on the right.

Fixed-point numbers maintain decimal or binary point alignment and may be truncated on the left or right, or extended with zeros on the left or right. Since the binary point of a fixed-point binary variable is always assumed to be after the rightmost binary digit, fixed-point binary values assigned to such variables will never result in extension on the right; of course, extension can occur on the left, but only truncation can occur on the right.

No indication is given of loss of significant digits on the right. Loss of digits on the left can be checked for if the SIZE condition is enabled. In System/360 implementations, binary fixed-point numbers are stored in words of 31 bits, whatever the declared width. Decimal numbers are always stored as an odd number of digits, since they are maintained in System/360 packed decimal format, with the rightmost four bits of the rightmost byte expressing the sign.

Base Conversion

Changes in base will usually affect only the value of noninteger fixed-point numbers. Some decimal fractions cannot be expressed exactly in binary, and some errors will then occur due to truncation. Some binary fractions will also require more decimal digits for exact representation than are automatically generated by the conversion rules, and this may also cause errors resulting from truncation.

Since the range of binary fixed-point numbers is smaller than the range of decimal fixed-point numbers, it is possible for significant digits to be lost on the left in conversion from decimal to binary. This will raise the SIZE condition, but an interrupt will not occur unless the condition is explicitly enabled by a SIZE prefix.

The natural notation for constants is decimal and, therefore, most constants are written in decimal. The precision of a constant is derived from the way in which it is written. Care should therefore be taken when writing noninteger constants that will be converted to fixed-point binary.

DATA TYPE CONVERSION

Coded arithmetic data cannot be converted to character string and vice versa. Character string data cannot be converted to numeric character.

Coded Arithmetic to Numeric Character

Coded arithmetic data being converted to numeric character is converted, if necessary, to a decimal value whose scale and precision are determined by the PICTURE attribute of the numeric character item.

Numeric Character to Coded Arithmetic

Numeric character data being converted to coded arithmetic is first interpreted as a decimal item of scale and precision as specified by the corresponding PICTURE attribute. This item is then converted to the base, scale, and precision of the coded arithmetic target.

Numeric Character to Character-String

Numeric character data items are interpreted as character strings. The length of the character string is the same as the length of the numeric character data item as described by its corresponding PICTURE attribute (i.e., the same as the length of the character-string value of the numeric character data).

Character-String to Bit-String

The character 1 in the source string becomes the bit 1 in the target string. The character 0 in the source string becomes the bit 0 in the target string. Any character other than 0 and 1 in the source string will raise the CONVERSION condition.

If the source string is longer than the target, excess characters on the right are ignored (so that excess characters other than 0 or 1 will not raise the CONVERSION condition). If the target is longer than the source, the target is padded on the right with zeros.

Bit-String to Character-String

The bit 0 becomes the character 0, and the bit 1 becomes the character 1. The generated character string, which has the same length as the source bit string, is assigned to the target.

If the source bit string is shorter than the target character string, the remainder of the target is padded with blanks. Examples are shown below.

Source	Value	Result
'1011'B	CHARACTER(4)	'1011'
'10101'B	CHARACTER(10)	'10101bbbbb'
'0001'B	CHARACTER(1)	'0'

Table F-1. Precision for Arithmetic Conversions

Source Attributes	Target Attributes	Target Precision
DECIMAL FIXED (p,q)	DECIMAL FLOAT	p
DECIMAL FIXED (p,q)	BINARY FIXED	1 +p *3.32, q * 3.32 (see note 3)
DECIMAL FIXED (p,q)	BINARY FLOAT	p * 3.32
DECIMAL FLOAT (p)	BINARY FLOAT	p * 3.32
BINARY FIXED (p,q)	BINARY FLOAT	p
BINARY FIXED (p,q)	DECIMAL FIXED	1 + p/3.32, q/3.32 (see note 4)
BINARY FIXED (p,q)	DECIMAL FLOAT	p/3.32
BINARY FLOAT (p)	DECIMAL FLOAT	p/3.32

Notes:
1. In the cases of p*3.32 and p/3.32, the CEIL of the result is taken; the value taken is an integer that is equal to or greater than the result.
2. Target precision never can exceed the implementation-defined maximums, which are 15 for FIXED DECIMAL, 31 for FIXED BINARY, 16 for FLOAT DECIMAL, and 53 for FLOAT BINARY.
3. When q is negative, the following formula applies:
   (MIN(CEIL(p*3.32)+1,31),CEIL(ABS(q)*3.32)*SIGN(q))
4. When q is negative, the following formula applies:
   (CEIL(p/3.32)+1,CEIL(ABS(q)/3.32*SIGN(q))

The CONVERSION condition cannot be raised on conversion from bit to character; however, a character string created by conversion from a bit string can cause a conversion error when reconverted if blanks are inserted.

Coded Arithmetic to Bit-String

The absolute arithmetic value is first converted to a binary integer, whose precision is the same as the length of the bit-string target as given in Table F-2. This integer, without a sign, is then treated as a bit string. This intermediate string is then assigned to the target. Some examples are shown in Figure F-1.

Source Attributes	Source Value	Intermediate String	Target Attributes	Result
FIXED BINARY(10)	15	0000001111	BIT(10)	0000001111
FIXED BINARY(1)	1	1	BIT(1)	1
FIXED DECIMAL(1)	1	0001	BIT(1)	0
FIXED BINARY(3)	-3	011	BIT(3)	011
FIXED DECIMAL(2,1)	1.1	0001	BIT(4)	0001
FLOAT BINARY(4)	1.25	0001	BIT(5)	00010

Figure F-1. Examples of Conversion From Arithmetic to Bit-String

Bit-String to Coded Arithmetic

The bit string is interpreted as an unsigned binary integer with an implementation-defined maximum precision. For the D-Compiler, this is 31 bits. If the string is shorter than 31 bits, zeros are inserted on the left. The result of a bit-string to arithmetic conversion is always positive. Note that padding is on the left, not on the right.

Numeric Character to Bit-String

The numeric character field is first converted to coded arithmetic and then to bit string, according to the above rules.

Bit-String to Numeric Character

The bit string is first converted to coded arithmetic and then to numeric character, according to the above rules.

Table F-2. Lengths of Converted Bit Strings (Coded Arithmetic to Bit-String)

Source Attributes	Target Length
DECIMAL FIXED (p,q)	(p - q) * 3.32
DECIMAL FLOAT (p)	p * 3.32
BINARY FIXED (p,q)	p - q
BINARY FLOAT (p)	p
Note: In the cases of p*3.32 and (p-q)*3.32, the CEIL of the result is taken. Also, for the D-Compiler, the target length must lie within 1 and 31, inclusive.	

TABLE OF CEILING VALUES

Table F-3 is intended to aid the programmer in computing the ceiling values used to determine precisions and lengths in conversions. It gives the ceiling for the result of a multiplication by 3.32 of any value between 1 and 15 as well as the ceiling for the result of a division by 3.32 of any value between 1 and 56.

Table F-3. Ceilings for Values Multiplied and Divided by 3.32

x	CEIL(x*3.32)	y	CEIL(y/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
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
		50-53	16
		54-56	17

TABLES FOR RESULTS OF ARITHMETIC OPERATIONS

Tables F-4 through F-7 give the attributes of the results of arithmetic operations, based on the operator specified and the attributes of the two operands. In these tables the target precisions can never exceed the implementation-defined maximums, which are 15 for FIXED DECIMAL, 31 for FIXED BINARY, 16 for FLOAT DECIMAL, and 53 for FLOAT BINARY.

Table F-4. Attributes of Result in Addition and Subtraction Operations

		First Operand			
		DECIMAL FIXED(p_1,q_1)	DECIMAL FLOAT(p_1)	BINARY FIXED(p_1,q_1)	BINARY FLOAT(p_1)
Second Operand	DECIMAL FIXED (p_2,q_2)	DECIMAL FIXED(p,q) $p=1+MAX(p_1-q_1,p_2-q_2)+MAX(q_1,q_2)$ $q=MAX(q_1,q_2)$	DECIMAL FLOAT(p) $p=MAX(p_1,p_2)$	BINARY FIXED(p,q) $p=1+MAX(p_1-q_1,r-s)+MAX(q_1,s)$ $q=MAX(q_1,s)$ where $r=1+p_2*3.32$ $s=q_2*3.32$	BINARY FLOAT(p) $p=MAX(p_1,r)$ where $r=p_2*3.32$
	DECIMAL FLOAT (p_2)	DECIMAL FLOAT(p) $p=MAX(p_1,p_2)$	DECIMAL FLOAT(p) $p=MAX(p_1,p_2)$	BINARY FLOAT(p) $p=MAX(p_1,r)$ where $r=p_2*3.32$	BINARY FLOAT(p) $p=MAX(p_1,r)$ where $r=p_2*3.32$
	BINARY FIXED (p_2,q_2)	BINARY FIXED(p,q) $p=1+MAX(r-s,p_2-q_2)+MAX(s,q_2)$ $q=MAX(s,q_2)$ where $r=1+p_1*3.32$ $s=q_1*3.32$	BINARY FLOAT(p) $p=MAX(r,p_2)$ where $r=p_1*3.32$	BINARY FIXED(p,q) $p=1+MAX(p_1-q_1,p_2-q_2)+MAX(q_1,q_2)$ $q=MAX(q_1,q_2)$	BINARY FLOAT(p) $p=MAX(p_1,p_2)$
	BINARY FLOAT (p_2)	BINARY FLOAT(p) $p=MAX(r,p_2)$ where $r=p_1*3.32$	BINARY FLOAT(p) $p=MAX(r,p_2)$ where $r=p_1*3.32$	BINARY FLOAT(p) $p=MAX(p_1,p_2)$	BINARY FLOAT(p) $p=MAX(p_1,p_2)$

Table F-5. Attributes of Result in Multiplication Operations

		First Operand			
		DECIMAL FIXED(p_1,q_1)	DECIMAL FLOAT(p_1)	BINARY FIXED(p_1,q_1)	BINARY FLOAT(p_1)
Second Operand	DECIMAL FIXED (p_2,q_2)	DECIMAL FIXED(p,q) $p=p_1+p_2+1$ $q=q_1+q_2$	DECIMAL FLOAT(p) $p=MAX(p_1,p_2)$	BINARY FIXED(p,q) $p=p_1+r+1$ $q=q_1+s$ where $r=1+p_2*3.32$ $s=q_2*3.32$	BINARY FLOAT(p) $p=MAX(p_1,r)$ where $r=p_2*3.32$
	DECIMAL FLOAT (p_2)	DECIMAL FLOAT(p) $p=MAX(p_1,p_2)$	DECIMAL FLOAT(p) $p=MAX(p_1,p_2)$	BINARY FLOAT(p) $p=MAX(p_1,r)$ where $r=p_2*3.32$	BINARY FLOAT(p) $p=MAX(p_1,r)$ where $r=p_2*3.32$
	BINARY FIXED (p_2,q_2)	BINARY FIXED(p,q) $p=r+p_2+1$ $q=s+q_2$ where $r=1+p_1*3.32$ $s=q_1*3.32$	BINARY FLOAT(p) $p=MAX(r,p_2)$ where $r=p_1*3.32$	BINARY FIXED(p,q) $p=p_1+p_2+1$ $q=q_1+q_2$	BINARY FLOAT(p) $p=MAX(p_1,p_2)$
	BINARY FLOAT (p_2)	BINARY FLOAT(p) $p=MAX(r,p_2)$ where $r=p_1*3.32$	BINARY FLOAT(p) $p=MAX(r,p_2)$ where $r=p_1*3.32$	BINARY FLOAT(p) $p=MAX(p_1,p_2)$	BINARY FLOAT(p) $p=MAX(p_1,p_2)$

Table F-6. Attributes of Result in Division Operations

		First Operand			
		DECIMAL FIXED(p_1,q_1)	DECIMAL FLOAT(p_1)	BINARY FIXED(p_1,q_1)	BINARY FLOAT(p_1)
Second Operand	DECIMAL FIXED (p_2,q_2)	DECIMAL FIXED(p,q) p=15 q=15-((p_1-q_1)+q_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FIXED(p,q) p=31 q=31-((p_1-q_1)+s) where s=q_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where r=p_2*3.32
	DECIMAL FLOAT (p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,r) where r=p_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where r=p_2*3.32
	BINARY FIXED (p_2,q_2)	BINARY FIXED(p) p=31 q=31-((r-s)+q_2) where r=1+p_1*3.32 s=q_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where r=p_1*3.32	BINARY FIXED(p,q) p=31 q=31-((p_1-q_1)+q_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)
	BINARY FLOAT (p_2)	BINARY FLOAT(p) p=MAX(r,p_2) where r=p_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where r=p_1*3.32	BINARY FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)

Table F-7. Attributes of Result in Exponentiation Operations

	First Operand	Second Operand (Exponent)	Target Attributes of Result
Case (1)	FIXED DECIMAL(p_1,q_1)	Unsigned integer constant with value n	FIXED DECIMAL(p,q) [provided p≤15] p=(p_1+1)*n-1 q=q_1*n
Case (2)	FIXED BINARY(p_1,q_1)	Unsigned integer constant with value n	FIXED BINARY(p,q) [provided p≤31] p=(p_1+1)*n-1 q=q_1*n
Case (3)	FIXED DECIMAL(p_1,q_1) or FLOAT DECIMAL(p_1)	FIXED DECIMAL(p_2,q_2) or FLOAT DECIMAL(p_2)	FLOAT DECIMAL(p) [unless case (1) above is applicable] p=MAX(p_1,p_2)
Case (4)	FIXED BINARY(p_1,q_1) or FLOAT BINARY(p_1)	FIXED DECIMAL(p_2,q_2) or FLOAT DECIMAL(p_2)	FLOAT BINARY(p) [unless case (2) above is applicable] p=MAX(p_1,CEIL(3.32*p_2))
Case (5)	FIXED DECIMAL(p_1,q_1) or FLOAT DECIMAL(p_1)	FIXED BINARY(p_2,q_2) or FLOAT BINARY(p_2)	FLOAT BINARY(p) [unless case (1) above is applicable] p=MAX(CEIL(3.32*p_1),p_2)
Case (6)	FIXED BINARY(p_1,q_1) or FLOAT BINARY(p_1)	FIXED BINARY(p_2,q_2) or FLOAT BINARY(p_2)	FLOAT BINARY(p) [unless case (2) above is applicable] p=MAX(p_1,p_2)

SECTION G: BUILT-IN FUNCTIONS AND PSEUDO-VARIABLES

All of the built-in functions and pseudo-variables that are available to the PL/I programmer are given in this section. The general organization of this section is as follows:

1. Computational Built-in Functions
   a. String-handling built-in functions
   b. Arithmetic built-in functions
   c. Mathematical built-in functions
   d. Array manipulation built-in functions
2. Miscellaneous Built-in Functions
3. Pseudo-Variables

The computational built-in functions, as shown above, provide string handling, arithmetic operations (absolute value, truncation, etc.), mathematical operations (trigonometric functions, square root, etc.), and array manipulation functions. The computational built-in functions are:

String Handling:

BIT	LOW
BOOL	REPEAT
CHAR	SUBSTR
HIGH	UNSPEC
INDEX	

Arithmetic:

ABS	MAX
BINARY	MIN
CEIL	MOD
DECIMAL	PRECISION
FIXED	ROUND
FLOAT	SIGN
FLOOR	TRUNC

Mathematical:

ATAN	LOG10
ATAND	LOG2
ATANH	SIN
COS	SIND
COSD	SINH
COSH	SQRT
ERF	TAN
ERFC	TAND
EXP	TANH
LOG	

Array Manipulation:

ALL
ANY
PROD
SUM

The miscellaneous built-in functions perform various duties; for example, one function provides the current date, another provides the time. The miscellaneous built-in functions are:

ADDR
DATE
NULL
STRING
TIME

The section on pseudo-variables gives a short discussion for each of the two PL/I pseudo-variables SUBSTR and UNSPEC. A more complete description can be had by consulting the discussion of the corresponding built-in function.

All of the built-in functions and pseudo-variables are presented in alphabetical order under their proper headings.

COMPUTATIONAL BUILT-IN FUNCTIONS

STRING HANDLING BUILT-IN FUNCTIONS

The functions described in this section may be used for manipulating strings. Unless otherwise specified, element expressions or array names can be used as arguments. When an argument is an array name, the value returned by the built-in function is an array of the same dimension and bounds as the argument (the function having been performed for each element of the array argument).

BIT String Built-in Function

Definition: BIT converts a given value to a bit string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a bit-string conversion.

Reference: BIT (expression[,size])

Arguments: The argument "expression" represents the quantity to be converted to a bit string; this argument can be a bit-string, character-string, or arithmetic element expression or array name. The argument "size," when specified, must be a decimal integer constant giving the length of the result. If "size" is not specified, it is determined according to the type conversion rules given in Section F, "Data Conversion." If "expression" is an array name, "size" applies to each element.

Result: The value returned by this function is "expression" converted to a bit string. The length of this bit string is determined by "size," as described above.

BOOL String Built-in Function

Definition: BOOL produces a bit string whose bit representation is a result of a given boolean operation on two given bit strings.

Reference: BOOL (x,y,w)

Arguments: Arguments "x" and "y" represent the two bit strings upon which the boolean operation specified by "w" is to be performed; these arguments can be bit-string, character-string, or arithmetic element expressions or array names. If "x" and "y" are not bit strings, they are converted to bit strings. If "x" and "y" differ in length, the shorter string is extended with zeros on the right to match the length of the longer string.

Argument "w" represents the boolean operation; this argument can be a bit-string, character-string, or arithmetic element expression or array name. It is converted to a bit string of length 4 and is defined as n_1 n_2 n_3 n_4, where each n is either 0 or 1. There are 16 possible bit combinations and thus 16 possible boolean operations. As for "x" and "y," "w" is converted to a bit string (of length 4) before the function is invoked, if necessary.

If more than one argument is an array, the arrays must have identical bounds.

Result: The value returned by this function is a bit string, z, whose length is equal to the longer of "x" and "y." Each bit of z is determined by the boolean operation on the corresponding bits of "x" and "y" as follows: the ith bit of z is set to the value of n_1, n_2, n_3, or n_4, depending on the combination of the ith bits of "x" and "y" as shown in the boolean table below:

xi	yi	zi
0	0	n_1
0	1	n_2
1	0	n_3
1	1	n_4

Example: In the following assignment statement, assume that U and ID have been declared as bit strings, XXX is the string '011'B, YYY is the string '110'B, and the boolean operator is '0110'B:

```
U=ID||BOOL (XXX, YYY, '0110'B);
```

Further, assume that Z represents the value returned to the point at which BOOL is invoked (that is, Z is a bit string of length 3 that is to be concatenated with ID), then the boolean table for this invocation of BOOL can be defined as:

XXXi	YYYi	Zi
0	0	0
0	1	1
1	0	1
1	1	0

which is interpreted as follows:

> Whenever the ith bits of XXX and YYY are 0 and 0, respectively, the ith bit of Z is 0; whenever the ith bits of XXX and YYY are 0 and 1, respectively, the ith bit of Z is 1, and so on.

Thus, since the first bits of XXX and YYY are 0 and 1, respectively, the first bit of Z is 1; since the second bits of XXX and YYY are 1 and 1, respectively, the second bit of Z is 0; and since the third bits of XXX and YYY are 1 and 0, respectively, the third bit of Z is 1. Therefore, the value returned to the point of invocation is the bit string '101'B.

CHAR String Built-in Function

Definition: CHAR converts a given value to a character string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a character-string conversion.

Reference: CHAR (expression[, size])

Arguments: The argument "expression" represents the quantity to be converted to a character string; this argument can be a bit-string, character-string, or numeric character element expression or array name. The argument "size," when specified, must be a decimal integer constant giving the length of the result. If "size" is not specified, it is determined according to the type conversion rules given in Section F, "Data Conversion." If "expression" is an array name, "size" refers to each element of the array.

Result: The value returned by this function is "expression" converted to a character string. The length of this character string is determined by "size," as described above.

HIGH String Built-in Function

Definition: HIGH forms a character string of a given length from the highest character in the collating sequence; that is, each character in the constructed string is the highest character in the collating sequence (see Section B).

Reference: HIGH (i)

Argument: The argument, "i," must be an unsigned decimal integer constant specifying the length of the string that is to be formed. For System/360 implementations, this character is stored as hexadecimal FF.

Result: The value returned by this function is a character string of length "i"; each character in the string is the highest character in the collating sequence.

INDEX String Built-in Function

Definition: INDEX searches a specified string for a specified bit or character string configuration. If the configuration is found, the starting location of that configuration within the string is returned to the point of invocation.

Reference: INDEX (string, config)

Arguments: Two arguments must be specified. The first argument, "string," represents the string to be searched; the second argument, "config," represents the bit or character string configuration for which "string" is to be searched. These arguments must be bit-string, character-string, binary coded arithmetic, or numeric character element expressions or array names. If neither argument is a bit string, or if only one argument is a bit string, both arguments are converted to character strings, if possible. If both arguments are bit strings, no conversion is performed. Note that binary coded arithmetic arguments are converted to bit-string and numeric character arguments are converted to character-string before the above conversions are performed.

If both arguments are arrays, the arrays must have identical bounds.

Result: The value returned by this function is a binary integer of default precision (15). This binary integer is either:

1. The location (i.e., the character or bit position) in "string" at which "config" has been found. If more than one "config" exists in "string," the location of the first one found (in a left-to-right sense) will be returned.

2. The value 0, if "config" does not exist within "string."

Example: If ASTRING is a character string containing:

```
'912NAMEA,1,FIRST,2,SECOND'
```

then the statement:

```
I = INDEX(ASTRING,'1,');
```

will return a binary value of ten to the point of invocation. This binary value represents the location of the configuration '1,' within ASTRING. However, if the statement had been:

```
I = INDEX(ASTRING,'1');
```

then a binary value of two would be returned to the point of invocation. This value is the location of the first '1' appearing within ASTRING.

LOW String Built-in Function

Definition: LOW forms a character string of specified length from the lowest charac-

ter in the collating sequence; i.e., each character of the formed string will be the lowest character in the collating sequence (see Section B).

Reference: LOW (i)

Argument: The argument, "i," must be an unsigned decimal integer constant specifying the length of the string being formed.

Result: The value returned by this function is a character string of length "i"; each character in the string is the lowest character in the collating sequence. For System/360 implementations, this character is stored as hexadecimal 00.

REPEAT String Built-in Function

Definition: REPEAT takes a given string value and forms a new string consisting of the given string value concatenated with itself a specified number of times.

Reference: REPEAT (string,i)

Arguments: The argument "string" represents a character or bit string from which the new string will be formed; this argument can be a binary coded arithmetic, bit-string, character-string, or numeric character element expression or array name. If an argument other than a bit or character string is specified, it is converted, before the function is invoked, to a bit or character string.

The argument "i" must be a decimal integer constant. It represents the number of times that "string" is to be concatenated with itself; "i" must be greater than zero.

Result: The value returned by this function is "string" concatenated with itself "i" times. In other words, the returned value will be a string containing i+1 occurrences of the value "string."

Example: If BSTR is a bit string containing '101'B, the statement

```
A = REPEAT(BSTR,6);
```

will cause the following value to be returned to the point of invocation:

```
'101101101101101101'B
```

SUBSTR String Built-in Function

Definition: SUBSTR extracts a substring of user-defined length from a given string and returns the substring to the point of invocation. (SUBSTR can also be used as a pseudo-variable.)

Reference: SUBSTR (string,i,j)

Arguments: The argument "string" represents the string from which a substring will be extracted; this argument can be a binary coded arithmetic, bit-string, character-string, or numeric character element expression or array name. If "string" is not a bit or character string, it is converted, before the function is invoked, to a bit or character string. Argument "i" represents the starting point of the substring and "j" represents the length of the substring. Argument "i" must be an element expression (it can be an array name but only if "string" is an array) that can be converted to an integer; "j" must be a decimal integer constant. If "i" is an array, it must have the same bounds as "string."

Assuming that the length of "string" is k, arguments "i" and "j" must satisfy the following conditions:

1. j must be less than or equal to k and greater than or equal to 1.
2. i must be less than or equal to k and greater than or equal to 1.
3. The value of i + j - 1 must be less than or equal to k.

Thus, the substring, as specified by "i" and "j" must lie within "string." Note that condition 1 is checked by the compiler; conditions 2 and 3 are never checked.

Result: The value returned by this function is that substring beginning at the ith character or bit of the first argument and extending "j" characters or bits.

Example: If AAA is a character string of length 30, the statement:

```
ITEM = SUBSTR(AAA, 7, 14);
```

will cause a 14-character substring to be extracted from AAA, starting at the seventh character of AAA. The extracted string is then returned to the point of invocation, after which it is assigned to ITEM (assuming ITEM is a character-string variable).

UNSPEC String Built-in Function

Definition: UNSPEC returns a bit string that is the internal coded representation of a given value. (UNSPEC can also be used as a pseudo-variable.)

Reference: UNSPEC (x)

Argument: The argument, "x," may be an arithmetic, character-string, or pointer value (element expressions or array names only) whose internal coded representation is to be found; "x" cannot be a bit string.

Result: The value returned by this function is the internal coded representation of "x." This representation is in bit-string form. The length of this string depends upon the attributes of "x," and is defined for System/360 implementations as follows:

1. If "x" is FIXED BINARY of precision (p,q), the length is 32.

2. If "x" if FIXED DECIMAL of precision (p,q), the length is defined as 8*FLOOR ((p+2)/2).

3. If "x" is FLOAT BINARY of precision p, the length is

   a. 32, if p is less than or equal to 21.

   b. 64, if p is greater than 21.

4. If "x" is FLOAT DECIMAL of precision p, the length is

   a. 32, if p is less than or equal to 6.

   b. 64, if p is greater than or equal to 7.

5. If "x" is a character-string of length n, or a numeric character item whose character-string value is of length n, the length is 8*n; for the D-Compiler, n must not be greater than 8.

6. If "x" is a pointer, the length is 32, however, the value of pointer is represented by the rightmost 24 bits.

ARITHMETIC BUILT-IN FUNCTIONS

All values returned by the arithmetic built-in functions are in coded arithmetic form. The arguments of these functions should also be in that form. If an argument is not coded arithmetic, then, before the function is invoked, it is converted to coded arithmetic according to the rules stated in Section F, "Data Conversion." Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In some function descriptions, the phrase "converted to the highest characteristics" is used; this means that the rules for mixed characteristics, as stated in the section "Data Conversion in Arithmetic Operations" in Part I, Chapter 4, "Expressions," are followed.

In general, an argument of an arithmetic built-in function may be an element expression or an array name. If an argument is an array name, the value returned by the built-in function is an array of the same dimension and bounds as the argument (the function having been performed once for each element of the array). Thus, for example, if an array argument is passed to the absolute value function ABS, the returned value is an array, each element of which is the absolute value of the corresponding element in the argument array.

Unless it is specifically stated otherwise, the base, scale, and precision of the returned value are determined according to the rules for the conversion of expression operands as given in Section F, "Data Conversion."

In many of these built-in functions, the symbol N is used. This symbol represents the maximum precision that a value may have. It is defined, for System/360 implementations, as follows:

N is 15 for FIXED DECIMAL values

16 for FLOAT DECIMAL values

31 for FIXED BINARY values

53 for FLOAT BINARY values

ABS Arithmetic Built-in Function

Definition: ABS finds the absolute value of a given quantity and returns it to the point of invocation.

Reference: ABS (x)

Argument: The quantity whose absolute value is to be found is given by "x."

Result: The value returned by this function is the absolute value of "x." The base, scale, and precision are the same as those of "x."

BINARY Arithmetic Built-in Function

Definition: BINARY converts a given value to binary base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a binary conversion.

Reference: BINARY (x[,p[,q]])

Arguments: The first argument, "x," represents the value to be converted to binary base. Arguments "p" and "q," when specified, must be decimal integer constants giving the precision of the binary result; "q" may be signed. The precision of a fixed-point result is (p,q); the precision of a floating-point result is (p). If both "p" and "q" are omitted, the precision of the result is determined according to the rules given for base conversion in Section F, "Data Conversion." Note that "q" must be omitted for floating-point arguments.

Result: The value returned by this function is the binary equivalent of "x." The scale of this value is the same as that of "x." The precision is given by "p" and "q."

CEIL Arithmetic Built-in Function

Definition: CEIL determines the smallest integer that is greater than or equal to a given value and returns that integer to the point of invocation.

Reference: CEIL (x)

Argument: The argument is "x."

Result: The value returned by this function is the smallest integer that is greater than or equal to "x." The base, scale, and precision are the same as those of "x," with one exception: if "x" is a fixed-point value of precision (p,q), the precision of the result is defined as:

```
(MIN(N,MAX(p-q+1,1)),0)
```

DECIMAL Arithmetic Built-in Function

Definition: DECIMAL converts a given value to decimal base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a decimal conversion.

Reference: DECIMAL (x[,p[,q]])

Arguments: The first argument, "x," represents the value to be converted to decimal base. Arguments "p" and "q," when specified, must be decimal integer constants giving the precision of the decimal result; "q" may be signed. The precision of a fixed-point result is (p,q); the precision of a floating-point result is (p). If both "p" and "q" are omitted, however, the precision of the result is determined according to the rules given for base conversion in Section F, "Data Conversion." Note that "q" must be omitted for floating-point arguments.

Result: The value returned by this function is the decimal equivalent of the argument "x"; its precision is given by "p" and "q."

FIXED Arithmetic Built-in Function

Definition: FIXED converts a given value to fixed-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a fixed-point conversion.

Reference: FIXED (x[,p[,q]])

Argument: The first argument, "x," represents the value to be converted to fixed-point scale. Arguments "p" and "q," when specified, must be decimal integer constants ("q" can be signed) giving the precision of the result, (p,q). For System/360 implementations, if "p" and "q" are omitted, "p" is assumed to be 15 for binary "x" and 5 for decimal "x"; "q" is assumed to be 0.

Result: The value returned by this function is the fixed-point equivalent of the argument "x"; its precision is (p,q).

FLOAT Arithmetic Built-in Function

Definition: FLOAT converts a given value to floating-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a floating-point conversion.

Reference: FLOAT (x[,p])

Arguments: The first argument, "x," represents the value to be converted to floating-point scale. The second argument,

"p_x" when specified, must be decimal integer constant giving the precision of the result. For System/360 implementations, if "p" is omitted, it is assumed to be 21 for binary "x" and 6 for decimal "x."

Result: The value returned by this function is the floating-point equivalent of "x"; its precision is "p."

FLOOR Arithmetic Built-in Function

Definition: FLOOR determines the largest integer that does not exceed a given value and returns that integer to the point of invocation.

Reference: FLOOR (x)

Argument: The argument is "x."

Result: The value returned by this function is the largest integer that does not exceed "x." The base, scale, and precision of this value are the same as those of "x," with one exception: if "x" is a fixed-point value of precision (p,q), the precision of the result is:

(MIN(N,MAX(p-q+1,1)),0)

MAX Arithmetic Built-in Function

Definition: MAX extracts the highest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MAX (x_1, x_2,...,x_n)

Arguments: Two or more arguments must be given.

Result: The value returned by MAX is the value of the maximum-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions:

(p_1,q_1), (p_2,q_2),..., (p_n,q_n)

then the precision of the result is as follows:

(MIN(N,MAX(p_1-q_1,...,p_n-q_n)+
MAX(q_1,...,q_n)),MAX(q_1,...q_n))

MIN Arithmetic Built-in Function

Definition: MIN extracts the lowest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MIN (x_1, x_2,...,x_n)

Arguments: Two or more arguments must be given.

Result: The value returned by MIN is the value of the lowest-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions

(p_1,q_1), (p_2,q_2),...,(p_n,q_n)

then the precision of the result is as follows:

(MIN(N,MAX(p_1-q_1,...,p_n-q_n)+
MAX(q_1,...q_n)),MAX(q_1,...,q_n))

MOD Arithmetic Built-in Function

Definition: MOD extracts the remainder resulting from the division of one quantity by another and returns it to the point of invocation.

Reference: MOD (x_1, x_2)

Arguments: Two arguments must be given. Before the function is invoked, the base and scale of each argument are converted according to the rules for the conversion of expression operands, as given in Section F, "Data Conversion."

Result: The value returned by MOD is the positive remainder resulting from the division of "x_1" by "x_2" to yield an integer quotient. If the result is in floating-point scale, its precision is the higher of the precisions of the arguments (i.e., p=MAX(p_1,p_2)); if the result is in fixed-point scale, its precision is defined as follows:

(MIN(N,p_2-q_2+MAX(q_1,q_2)),MAX(q_1,q_2))

where:

(p_1,q_1) and (p_2,q_2) are the precisions of "x_1" and "x_2," respectively.

The base and scale of the result are those of the converted arguments.

PRECISION Arithmetic Built-in Function

Definition: PRECISION converts a given value to a specified precision and returns the converted value to the point of invocation.

Reference: PRECISION (x,p[,q])

Arguments: The first argument, "x," represents the value to be converted to the specified precision. Arguments "p" and "q" ("q" is optional and may be signed) are decimal integer constants specifying the precision of the result. If "x" is a fixed-point value, "p" and "q" must be specified; if "x" is a floating-point value, only "p" must be specified.

Result: The value returned by this function is the value of "x" converted to the specified precision. The base and scale of the returned value are the same as those of "x."

ROUND Arithmetic Built-in Function

Definition: ROUND rounds a given value at a specified digit and returns the rounded value to the point of invocation.

Reference: ROUND (expression,n)

Arguments: The first argument, "expression," must be coded arithmetic or numeric character. It is an element expression or array name representing the value (or values, in the case of an array) to be rounded; the second argument, "n," is an unsigned decimal integer constant specifying the digit at which the value of "expression" is to be rounded.

Result: If "expression" is fixed-point, ROUND returns the value of "expression" rounded at the nth digit to the right of the decimal (or binary) point. The base and scale of the result are the same as the base and scale of "expression;" the precision of the result is:

(MIN(p+1),N),q)

If "expression" is a floating-point expression, the second argument is ignored, and the rightmost bit in the internal floating-point representation of the expression value is set to 1 if it is 0; if the rightmost bit is 1, the value of the expression is unchanged. The base, scale, and precision of the returned value are those of the value of "expression."

Note that the rounding of a negative quantity results in the rounding of the absolute value of that quantity.

Example: If X is a fixed-point decimal variable of precision (7,5) containing the value 36.24976, and Y and Z are fixed-point decimal variables of precision (6,4), then after the execution of the following statements,

```
Y=ROUND(X,3);
Z=ROUND(X,4);
```

the value of Y is 36.2500 and the value of Z is 36.2498.

SIGN Arithmetic Built-in Function

Definition: SIGN determines whether a value is positive, negative, or zero, and it returns an indication to the point of invocation.

Reference: SIGN (x)

Argument: The argument is "x."

Result: This function returns a fixed-point binary value of default precision (15) according to the following rules:

1. If "x" is greater than 0, the returned value is 1.
2. If "x" is equal to zero, the returned value is 0.
3. If "x" is less than zero, the returned value is -1.

TRUNC Arithmetic Built-in Function

Definition: TRUNC truncates a given value to an integer as follows: first, it determines whether a given value is positive, negative, or equal to zero. If the value is negative, TRUNC returns the smallest integer that is greater than that value; if the value is positive or equal to zero, TRUNC returns the largest integer that does not exceed that value.

Reference: TRUNC (x)

Argument: The argument is "x."

Result: If "x" is less than zero, the value returned by TRUNC is CEIL(x). If "x" is greater than or equal to zero, the value returned by TRUNC is FLOOR(x). In either case, the base and scale of the result are

the same as those of "x." If "x" is floating-point, the precision remains the same. If "x" is a fixed-point value of precision (p,q), the precision of the result is:

```
(MIN(N,MAX(p-q+1,1)),0)
```

MATHEMATICAL BUILT-IN FUNCTIONS

All arguments to the mathematical built-in functions should be in coded arithmetic form and in floating-point scale. Any argument that does not conform to this rule is converted to coded arithmetic and floating-point before the function is invoked, according to the rules stated in Section F, "Data Conversion." Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In general, an argument to a mathematical built-in function may be an element expression or an array name. If an argument is an array name the value returned by the built-in function is an array of the same dimension and bounds as the argument (the function having been performed once for each element of the array). Thus, for example, an array argument passed to the cosine function COS results in an array, each element of which is the cosine of the corresponding element in the argument array.

All of the mathematical built-in functions return coded arithmetic floating-point values. The base and precision of these values are always the same as those of the arguments.

Figure G-1 at the end of this section provides a quick reference to the mathematical built-in functions.

ATAN Mathematical Built-in Function

Definition: ATAN finds the arctangent of a given value and returns the result expressed in radians, to the point of invocation.

Reference: ATAN (x[,y])

Arguments: The argument "x" must always be specified; the argument "y" is optional. If "y" is omitted, "x" represents the value whose arctangent is to be found.

If "y" is specified, then the value whose arctangent is to be found is taken to be the expression x/y. In this case, both "x" and "y" may not be equal to 0 at the same time.

Result: When "x" alone is specified, the value returned by ATAN is the arctangent of "x," expressed in radians, where:

```
-pi/2<ATAN(x)<pi/2
```

If both "x" and "y" are specified, the possible values returned by this function are defined as follows:

1. For y>0 and any x, the value is arctangent (x/y) in radians.
2. If x>0 and y=0, the value is (pi/2) radians.
3. If x≥0 and y<0, the value is (pi+ arctangent (x/y)) radians.
4. If x<0 and y=0, the value is (-pi/2) radians.
5. If x<0 and y<0, the value is (-pi+ arctangent (x/y)) radians.

ATAND Mathematical Built-in Function

Definition: ATAND finds the arctangent of a given value and returns the result, expressed in degrees, to the point of invocation.

Reference: ATAND (x[,y])

Arguments: If y is omitted, "x" represents the value whose arctangent is to be found. If "y" is specified, the value whose arctangent is to be found is represented by the expression x/y; in this case, both "x" and "y" cannot be equal to 0 at the same time.

Result: If "y" is specified, the value returned by this function is simply the arctangent of "x," expressed in degrees, where:

```
-90<ATAND(x)<90
```

If y is specified, the value returned by this function is ATAN (x,y), except that the value is expressed in degrees and not in radians (see "ATAN Mathematical Built-in Function" in this section); that is, the returned value is defined as:

```
ATAND(x,y) = (180/pi)*ATAN(x,y)
```

ATANH Mathematical Built-in Function

Definition: ATANH finds the inverse hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: ATANH (x)

Argument: The value whose inverse hyperbolic tangent is to be found is represented by "x." The absolute value of "x" must not be greater than or equal to 1; that is, it is an error if ABS(x)≥1.

Result: The value returned by this function is the inverse hyperbolic tangent of "x".

COS Mathematical Built-in Function

Definition: COS finds the cosine of a given value, which is expressed in radians, and returns the result to the point of invocation.

Reference: COS (x)

Argument: The value whose cosine is to be found is given by "x"; this value must be expressed in radians.

Result: The value returned by this function is the cosine of "x."

COSD Mathematical Built-in Function

Definition: COSD finds the cosine of a given value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: COSD (x)

Argument: The value whose cosine is to be found is given by "x"; this value must be expressed in degrees.

Result: The value returned by this function is the cosine of "x."

COSH Mathematical Built-in Function

Definition: COSH finds the hyperbolic cosine of a given value and returns the result to the point of invocation.

Reference: COSH (x)

Argument: The value whose hyperbolic cosine is to be found is given by "x."

Result: The value returned by this function is the hyperbolic cosine of "x."

ERF Mathematical Built-in Function

Definition: ERF finds the error function of a given value and returns it to the point of invocation.

Reference: ERF (x)

Argument: The value for which the error function is to be found is represented by "x."

Result: The value returned by this function is defined as follows:

$$ERF(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

ERFC Mathematical Built-in Function

Definition: ERFC finds the complement of the Error Function (ERF) for a given value and returns the result to the point of invocation.

Reference: ERFC (x)

Argument: The argument, "x," represents the value whose error function complement is to be found.

Result: The value returned by this function is defined as follows:

$$ERFC(x) = 1-ERF(x)$$

EXP Mathematical Built-in Function

Definition: EXP raises e (the base of the natural logarithm system) to a given power and returns the result to the point of invocation.

Reference: EXP (x)

Argument: The argument, "x," specifies the power to which e is to be raised.

Result: The value returned by this function is e raised to the power of "x."

LOG Mathematical Built-in Function

Definition: LOG finds the natural logarithm (i.e., base e) of a given value and returns it to the point of invocation.

Reference: LOG (x)

Argument: The argument, "x," is the value whose natural logarithm is to be found; it must not be less than or equal to 0.

Result: The value returned by this function is the natural logarithm of "x."

LOG10 Mathematical Built-in Function

Definition: LOG10 finds the common logarithm (i.e., base 10) of a given value and returns it to the point of invocation.

Reference: LOG10 (x)

Argument: The argument, "x," represents the value whose common logarithm is to be found; this value must not be less than or equal to 0.

Result: The value returned by this function is the common logarithm of "x."

LOG2 Mathematical Built-in Function

Definition: LOG2 finds the binary (i.e., base 2) logarithm of a given value and returns it to the point of invocation.

Reference: LOG2 (x)

Argument: The argument, "x," is the value whose binary logarithm is to be found; it must not be less than or equal to 0.

Result: The value returned to this function is the binary logarithm of "x."

SIN Mathematical Built-in Function

Definition: SIN finds the sine of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: SIN (x)

Argument: The argument, "x," is the value whose sine is to be found; it must be expressed in radians.

Result: The value returned by this function is the sine of "x."

SIND Mathematical Built-in Function

Definition: SIND finds the sine of a given value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: SIND (x)

Argument: The argument, "x," is the value whose sine is to be found; "x" must be expressed in degrees.

Result: The value returned by this function is the sine of "x."

SINH Mathematical Built-in Function

Definition: SINH finds the hyperbolic sine of a given value and returns the result to the point of invocation.

Reference: SINH (x)

Argument: The argument, "x," is the value whose hyperbolic sine is to be found.

Result: The value returned by this function is the hyperbolic sine of "x."

SQRT Mathematical Built-in Function

Definition: SQRT finds the square root of a given value and returns it to the point of invocation.

Reference: SQRT (x)

Argument: The argument, "x," is the value whose square root is to be found; it must not be less than 0.

Result: The value returned by this function is the positive square root of "x."

TAN Mathematical Built-in Function

Definition: TAN finds the tangent of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: TAN (x)

Argument: The argument, "x," represents the value whose tangent is to be found; "x" must be expressed in radians.

Result: The value returned by this function is the tangent of "x."

TAND Mathematical Built-in Function

Definition: TAND finds the tangent of a given value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: TAND (x)

Argument: The argument, "x," represents the value whose tangent is to be found; "x" must be expressed in degrees.

Result: The value returned by this function is the tangent of "x."

TANH Mathematical Built-in Function

Definition: TANH finds the hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: TANH (x)

Argument: The argument, "x," represents the value whose hyperbolic tangent is to be found.

Result: The value returned by this function is the hyperbolic tangent of "x."

Summary of Mathematical Functions

Figure G-1 summarizes the mathematical built-in functions. In using it, the reader should be aware of the following:

1. All arguments must be coded arithmetic and floating-point scale, or such that they can be converted to coded arithmetic and floating-point.
2. The value returned by each function is always in floating-point.
3. The error conditions are those defined by the PL/I Language. Additional error conditions detected by the D-Compiler can be found in the publication IBM System/360 Disk and Tape Operating Systems, PL/I Programmer's Guide, Form C24-9005.

ARRAY MANIPULATION BUILT-IN FUNCTIONS

The built-in functions described here may be used for the manipulation of arrays. All of these functions require array name arguments and return single element values. Note that since these functions return element values, a function reference to any of them is considered an element expression.

ALL Array Manipulation Function

Definition: ALL tests all bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not the corresponding bits of given array elements are all ones.

Reference: ALL (x)

Argument: The argument, "x," is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of an element in "x" (all elements in "x" must have the same length, of course), and whose bit values are determined by the following rule:

> If the ith bits of all of the elements in "x" are 1, then the ith bit of the result is 1; otherwise, the ith bit of the result is 0.

ANY Array Manipulation Function

Definition: ANY tests the bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not at least one of the corresponding bits of the given array elements is set to 1.

Reference: ANY (x)

Argument: The argument, "x," is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of an element in "x" (all elements in "x" must have the same length, of course), and whose bit values are determined by the following rule:

If the ith bit of any element in "x" is 1, then the ith bit of the result is 1; otherwise, the ith bit of the result is 0.

Function Reference	Value Returned	Error Conditions
ATAN(x)	arctan(x) in radians -(pi/2)<ATAN(x)<(pi/2)	-
ATAN(x,y)	see function description	error if x=0 and y=0
ATAND(x)	arctan(x) in degrees -90<ATAND(x)<90	-
ATAND(x,y)	see function description	error if x=0 and y=0
ATANH(x)	$\tanh^{-1}(x)$	error if ABS(x)≥1
COS(x) x in radians	cosine(x)	-
COSD(x) x in degrees	cosine(x)	-
COSH(x)	cosh(x)	-
ERF(x)	$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$	-
ERFC(x)	1 - ERF(x)	-
EXP(x)	e^x	
LOG(x)	$\log_e(x)$	error if x≤0
LOG10(x)	$\log_{10}(x)$	error if x≤0
LOG2(x)	$\log_2(x)$	error if x≤0
SIN(x) x in radians	sine(x)	-
SIND(x) x in degrees	sine(x)	-
SINH(x)	sinh(x)	-
SQRT(x)	$\sqrt{x}$	error if x<0
TAN(x) x in radians	tangent(x)	-
TAND(x) x in degrees	tangent(x)	-
TANH(x)	tanh(x)	-

Figure G-1. Mathematical Built-in Functions

PROD Array Manipulation Function

Definition: PROD finds the product of all of the elements of a given array and returns that product to the point of invocation.

Reference: PROD (x)

Argument: The argument, "x," should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being multiplied with the previous product.

Result: The value returned by this function is the product of all of the elements in "x." The scale of the result is floating-point, while the base and precision are those of the converted elements of "x."

SUM Array Manipulation Function

Definition: SUM finds the sum of all of the elements of a given array and returns that sum to the point of invocation.

Reference: SUM (x)

Argument: The argument, "x," should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being summed with the previous total.

Result: The value returned by this function is the sum of all of the elements in "x." The scale of the result is floating-point, while the base and precision are those of the converted elements of the argument.

MISCELLANEOUS BUILT-IN FUNCTIONS

The functions described in this section have little in common with each other and with the other categories of built-in functions. Some require arguments and others do not. Those that do not require arguments will be so identified.

ADDR Built-in Function

Definition: ADDR finds the location at which a given variable has been allocated and returns a pointer value to the point of invocation. This pointer value identifies the location at which the variable has been allocated.

Reference: ADDR (x)

Argument: The argument, "x," is the variable whose location is to be found. It can be an element variable, an array, a structure, an element of an array, or an element of a structure. It can be of any data type and storage class.

Result: ADDR returns a pointer value identifying the location at which "x" has been allocated. If "x" is a parameter, the returned value identifies the corresponding argument (dummy or otherwise). If "x" is a based variable, the returned value is determined from the pointer variable declared with "x"; if this pointer variable contains no value, the value returned by ADDR is undefined.

DATE Built-in Function

Definition: DATE returns the current date to the point of invocation.

Reference: DATE

Arguments: None

Result: The value returned by this function is a character string of length six, in the form yymmdd, where:

yy is the current year

mm is the current month

dd is the current day

Example: If the current date is February 29, 1968, execution of the statement

```
X=DATE;
```

will cause the character string '680229' to be returned to the point of invocation.

NULL Built-in Function

Definition: NULL returns a null pointer value to the point of invocation.

Reference: NULL

Arguments: None

Result: The value returned by this function is a null pointer value. For the D-Compiler, a null pointer is an invalid address that can be used as a unique indicator.

STRING Built-in Function

Definition: STRING forms a character string from a given structure having the PACKED attribute and returns that string to the point of invocation.

Reference: STRING (strname)

Argument: The argument, "strname," must be the name of a structure having the PACKED attribute. This structure must be composed of character strings and/or numeric character data only.

Result: The value returned by this function is a character string resulting from the concatenation of all of the elements in "strname."

TIME Built-in Function

Definition: TIME returns the current time to the point of invocation.

Reference: TIME

Arguments: None

Result: The value returned by this function is a character string of length nine, in the form hhmmssttt, where:

- hh is the current hour of the day
- mm is the number of minutes
- ss is the number of seconds
- ttt is the number of milliseconds in machine-dependent increments.

Example: If the current time is 4 P.M., 23 minutes, 19 seconds, and 80 milliseconds, a reference to the TIME function will return the character string '162319080' to the point of invocation.

PSEUDO-VARIABLES

In general, pseudo-variables are certain built-in functions that can appear wherever other variables can appear to receive values. In short, they are built-in functions used as receiving fields. A pseudo-variable can appear on the left of the equal sign in an assignment statement or it can appear in the data list of a GET statement. It cannot appear elsewhere.

There are only two pseudo-variables, SUBSTR and UNSPEC. Since they have built-in function counterparts, only a short description of each pseudo-variable is given here; the corresponding built-in function should be consulted for the details.

SUBSTR Pseudo-variable

Reference: SUBSTR (string,i,j)

Description: The value being assigned to SUBSTR is assigned to the substring of "string," as defined for the built-in function SUBSTR (with one exception: for the SUBSTR pseudo-variable "string" must be an element variable). The remainder of "string" remains unchanged.

UNSPEC Pseudo-variable

Reference: UNSPEC (v)

Description: The letter "v" represents an element variable of arithmetic, character string, or pointer type; it cannot be a bit-string variable. The value being assigned to UNSPEC is evaluated, converted to a bit string (the length of which is a function of the attributes of "v" -- see the UNSPEC built-in function), and then assigned to "v," without conversion to the type of "v."

INTRODUCTION

The ON-conditions are those exceptional conditions that can be specified in PL/I by means of an ON statement. If a condition is enabled, the occurrence of the condition will result in an interrupt. The interrupt, in turn, will result in the execution of the current action specification for that condition. If an ON statement for that condition is not in effect, the current action specification is the standard system action for that condition. If an ON statement for that condition is in effect, the current action specification is either SYSTEM, in which case the standard system action for that condition is taken, or an on-unit, in which case the programmer has supplied his own action to be taken for that condition (i.e., either a null statement or a GO TO statement).

If a condition is not enabled (i.e., if it is disabled), and the condition occurs, an interrupt will not take place, and errors may result.

Some conditions are always enabled unless they have been explicitly disabled by condition prefixes; another (i.e., SIZE) is always disabled unless it has been explicitly enabled by a condition prefix; and still others are always enabled and cannot be disabled.

Those conditions that are always enabled unless they have been explicitly disabled by condition prefixes are:

```
CONVERSION
FIXEDOVERFLOW
OVERFLOW
UNDERFLOW
ZERODIVIDE
```

Each of the above conditions can be disabled by a condition prefix specifying the condition name preceded by NO without intervening blanks. Thus, one of the following names in a condition prefix will disable the respective condition:

```
NOCONVERSION
NOFIXEDOVERFLOW
NOOVERFLOW
NOUNDERFLOW
NOZERODIVIDE
```

Such a condition prefix renders the corresponding condition disabled throughout the scope of the prefix; the condition remains enabled outside this scope (see Part I, Chapter 11, "Exceptional Condition Handling and Program Checkout" for a discussion of the scope of condition prefixes).

Conversely, the condition that is always disabled unless it has been enabled by a condition prefix is SIZE. The appearance of this condition in a condition prefix renders the condition enabled throughout the scope of the prefix; the condition remains disabled outside this scope. Further, a condition prefix specifying NOSIZE will disable the SIZE condition throughout the scope of that prefix.

All other conditions are always enabled and remain so for the duration of the program. These conditions are:

```
ENDFILE
ENDPAGE
ERROR
KEY
RECORD
TRANSMIT
```

SECTION ORGANIZATION

This section presents each condition in its logical grouping, and in alphabetical order within that grouping. In general, the following information is given for each condition:

1. General format -- given only when it consists of more than the condition name.
2. Description -- a discussion of the condition, including the circumstances under which the condition can be raised. Note that an enabled condition can always be raised by a SIGNAL statement; this fact is not included in the descriptions.
3. Result -- the result of the operation that caused the condition to occur. This applies when the condition is disabled as well as when it is enabled. In some cases, the result is not defined; that is, it cannot be predicted. This is stated wherever applicable.

4. Standard system action -- the action taken by the system when an interrupt occurs and the programmer has not specified an on-unit to handle that interrupt.

5. Status -- an indication of the enabled/disabled status of the condition at the start of the program, and how the condition may be disabled (if possible) or enabled.

6. Normal return -- the point to which control is returned as a result of a null statement on-unit. A GO TO statement on-unit is an abnormal on-unit termination. Note that if a condition has been raised by the SIGNAL statement, the normal return is always to the statement immediately following SIGNAL. Also note that the conditions ENDFILE, KEY, and CONVERSION cannot have null statement on-units associated with them and, therefore, a normal return can never be made for these conditions.

The conditions are grouped as follows:

1. Computational conditions -- those conditions associated with data handling, expression evaluation, and computation. They are:

   CONVERSION
   FIXEDOVERFLOW
   OVERFLOW
   SIZE
   UNDERFLOW
   ZERODIVIDE

2. Input/output conditions -- those conditions associated with data transmission. They are:

   ENDFILE
   ENDPAGE
   KEY
   RECORD
   TRANSMIT

3. System action condition -- the condition (i.e., ERROR) that provides facilities to extend the standard system action that is taken after the occurrence of a condition.

COMPUTATIONAL CONDITIONS

The CONVERSION Condition

Description: The CONVERSION condition occurs whenever an illegal conversion is attempted on character-string data. This attempt may be made internally or during an input/output operation. For example, the condition occurs when a character other than 0 or 1 exists in a character string being converted to a bit string or when characters that cannot be interpreted as arithmetic are encountered during a STREAM transmission operation for an arithmetic variable.

All conversions of character-string data are carried out character-by-character in a left-to-right sequence and the condition occurs for the first illegal character. When such a character is encountered, an interrupt occurs (provided, of course, that CONVERSION has not been disabled) and the current action specification for the condition is executed.

Result: When CONVERSION occurs, the contents of the entire result field are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: CONVERSION is enabled throughout the program, except within the scope of a condition prefix specifying NOCONVERSION.

Normal Return: A null on-unit cannot be specified for this condition.

The FIXEDOVERFLOW Condition

Description: The FIXEDOVERFLOW condition occurs when the length of the result of a fixed-point arithmetic operation exceeds N. For System/360 implementations, N is 15 for decimal fixed-point values and 31 for binary fixed-point values.

Result: The result of the invalid fixed-point operation is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: FIXEDOVERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOFIXEDOVERFLOW.

Normal Return: If a null on-unit is specified for this condition, control returns to the point immediately following the point of interrupt.

The OVERFLOW Condition

Description: The OVERFLOW condition occurs when the magnitude of a floating-point number exceeds the permitted maximum. (For System/360 implementations, the magnitude of a floating-point number or intermediate result must not be greater than approximately 10^{75} or 2^{252}.)

Result: The value of such an illegal floating-point number is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: OVERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOOVERFLOW.

Normal Return: If a null on-unit is specified for this condition, control returns to the point immediately following the point of interrupt.

The SIZE Condition

Description: The SIZE condition occurs only when high-order (i.e., leftmost) nonzero binary or decimal digits are lost in an assignment operation (i.e., assignment to a variable or an intermediate result) or in an input/output operation. This loss may result from a conversion involving different data types, different bases, different scales, or different precisions.

The SIZE condition differs from the FIXEDOVERFLOW condition in an important sense, i.e., FIXEDOVERFLOW occurs when the size of a calculated fixed-point value exceeds N (the maximum allowed), whereas 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. SIZE can be raised on assignment of a value regardless of whether or not FIXEDOVERFLOW was raised in the calculation of that value.

Result: The contents of the data item receiving the wrong-sized value are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: SIZE is disabled within the scope of a NOSIZE condition prefix and elsewhere throughout the program, except within the scope of a condition prefix specifying SIZE.

Normal Return: If a null on-unit is specified for this condition, control returns to the point immediately following the point of interrupt.

The UNDERFLOW Condition

Description: The UNDERFLOW condition occurs when the magnitude of a floating-point number is smaller than the permitted minimum. (For System/360 implementations, the magnitude of a floating-point value may not be less than approximately 10^{-78} or 2^{-260}.)

UNDERFLOW does not occur when equal numbers are subtracted (often called significance error).

Note that for the D-Compiler, the expression X**(-Y) (where Y>0) is evaluated by taking the reciprocal of X**Y; hence, the OVERFLOW condition may be raised instead of the UNDERFLOW condition.

Result: The invalid floating-point value is set to 0.

Standard System Action: In the absence of an on-unit, the system prints a message and continues execution from the point at which the interrupt occurred.

Status: UNDERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOUNDERFLOW.

Normal Return: If a null on-unit is specified for this condition, control returns to the point immediately following the point of interrupt.

The ZERODIVIDE Condition

Description: The ZERODIVIDE condition occurs when an attempt is made to divide by zero. This condition is raised for fixed-point and floating-point division.

Result: The result of a division by zero is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: ZERODIVIDE is enabled throughout the program, except within the scope of a condition prefix specifying NOZERODIVIDE.

Normal Return: If a null on-unit is specified for this condition, control

returns to the point immediately following the point of interrupt.

INPUT/OUTPUT CONDITIONS

The input/output conditions are always enabled and cannot appear in condition prefixes; they can be specified only in ON, SIGNAL, and REVERT statements.

The ENDFILE Condition

General Format: ENDFILE (file-name)

Description: The ENDFILE condition can be raised during a GET or READ operation; it is caused by an attempt to read past the file delimiter of the file named in the GET or READ statement.

After ENDFILE has been raised, the file should be closed.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: The ENDFILE condition is always enabled; it cannot be disabled.

Normal Return: A null on-unit cannot be specified for this condition.

The ENDPAGE Condition

General Format: ENDPAGE (file-name)

The "file name" must be the name of a file having the PRINT attribute.

Description: The ENDPAGE 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, an installation-defined system limit applies. The attempt to exceed the limit may be made during data transmission (including any format items specified in the PUT statement), by the LINE option, or by the SKIP option. ENDPAGE is raised only once per page.

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option (or the default) so that it is possible to continue writing on the same page.

After ENDPAGE has been raised, a new page can be started in either of the following ways:

1. Execution of a PAGE option or a PAGE format item.
2. Execution of a LINE option or a LINE format item specifying a line number less than or equal to the current line number.

When either of these occurs, a new page is started in the same way that it is when a PAGE option is executed, i.e., ENDPAGE is not raised and the current line is set to 1. If a new page is not started, the current line number may increase indefinitely.

If ENDPAGE is raised during data transmission, then, on return from a null on-unit, the data is written on the current line. If ENDPAGE results from a LINE or SKIP option, then, on return from a null on-unit, the action specified by LINE or SKIP is ignored.

Standard System Action: In the absence of an on-unit, the system starts a new page.

Status: ENDPAGE is always enabled; it cannot be disabled.

Normal Return: Upon the execution of a null on-unit for this condition, execution of the PUT statement continues in the manner described above.

The KEY Condition

General Format: KEY (file-name)

Description: The KEY condition can be raised only during operations on keyed records. It is raised in any of the following cases:

1. The keyed record cannot be found for a READ or REWRITE statement.
2. An attempt is made to add a duplicate key by a WRITE or LOCATE statement.
3. The key has not been specified correctly.
4. No space is available to add the keyed record.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: KEY is always enabled; it cannot be disabled.

Normal Return: A null on-unit cannot be specified for this condition.

The RECORD Condition

General Format: RECORD (file-name)

Description: The RECORD condition can be raised only during a READ, WRITE, REWRITE, or LOCATE operation. It is raised by either of the following:

1. The size of the record is greater than the size of the variable.
2. The size of the record is less than the size of the variable.

If the size of the record is greater than the size of the variable, the excess data in the record is lost on input and is unpredictable on output. If the size of the record is less than the size of the variable, the excess data in the variable is not transmitted on output and is unaltered on input.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: RECORD is always enabled; it cannot be disabled.

Normal Return: Upon execution of a null on-unit, execution continues with the statement immediately following the one for which RECORD occurred.

The TRANSMIT Condition

General Format: TRANSMIT (file-name)

Description: The TRANSMIT condition can be raised during any input/output operation. It is raised by a permanent transmission error and, as a result, any data transmitted is potentially incorrect. During input, the condition is raised after assignment of the potentially incorrect data item or record. During output, the condition is raised after the transmission of the potentially incorrect data item or record has been attempted.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: TRANSMIT is always enabled; it cannot be disabled.

Normal Return: Upon execution of a null on-unit, processing continues with the next data item for STREAM input/output, or the next statement for RECORD input/output.

SYSTEM ACTION CONDITION

The ERROR Condition

Description: The ERROR condition is raised under the following circumstances:

1. As a result of the standard system action for an ON-condition for which that action is to "print an error message and raise the ERROR condition"
2. As a result of an error (for which there is no ON-condition) occurring during program execution

Standard System Action: In the absence of an on-unit, the D-Compiler prints a message and returns control to the operating system control program.

Status: ERROR is always enabled; it cannot be disabled.

Normal Return: Upon execution of a null on-unit, control is returned to the operating system control program.

SECTION I: ATTRIBUTES

A name appearing in a PL/I program may have one of many different meanings. It may, for example, be a variable referring to arithmetic data items; it may be a file name; it may be a variable referring to a character string, or it may be a statement label or a variable referring to a statement label.

Properties, or characteristics, of the values a name represents (for example, arithmetic characteristics of data items represented by an arithmetic variable) and other properties of the name itself (such as scope, storage class, etc.) together make up the set of attributes that can be associated with a name.

The attributes enable the compiler to assign a unique meaning to the identifier specified in a DECLARE statement. For example, if the variable is an arithmetic data variable, the base, scale, and precision attributes must be associated with the name. Associated attributes are those specified in a DECLARE statement or assumed by default.

This section discusses the different attributes. The attributes are grouped by function and then detailed discussions follow, in alphabetic order, showing the rules, defaults, and format for each attribute.

SPECIFICATION OF ATTRIBUTES

Attributes specified in a DECLARE statement are separated by blanks. Except for the dimension, length, FILE, and precision attribute specifications, they may appear in any order. The dimension attribute specification must immediately follow the array name; the length and precision attribute specifications must follow one of their associated attributes; the FILE attribute must appear first in the declaration of a file name. A comma must follow the last attribute specification for a particular name (or the name itself, if no attributes are specified with it), unless it is the last name in the DECLARE statement, in which case the semicolon is used.

Factoring of Attributes

Except for the dimension and file description attributes, any attributes common to several names can be factored in a declaration to eliminate repeated specification of the same attribute for many identifiers. Factoring is achieved by enclosing the names in parentheses, and following this by the set of attributes which apply. All factored attributes must apply to all of the names. No factored attribute can be overridden for any of the names, but any name within the list may be given other attributes so long as there is no conflict with the factored attributes. For the D-Compiler, factoring can be nested to a level of eight. See the fourth example below for an illustration of such nesting.

Names within the parenthesized list are separated by commas.

Note: Structure level numbers can also be factored, but a factored level number must precede the parentnesized list.

Examples:

```
DECLARE (A,B,C,D) BINARY FIXED (31);

DECLARE (E DECIMAL(6,5),
   F CHARACTER(10)) STATIC;

DECLARE  1A,  2(B,C,D)  BINARY  FIXED
   (15), ...;

DECLARE ((A,B)  FIXED  (10),  C  FLOAT
   (5)) EXTERNAL;
```

DATA ATTRIBUTES

PROBLEM DATA

Attributes for problem data are used to describe arithmetic and string variables. Arithmetic variables have attributes that specify the base, scale, and precision of the data items. String variables have attributes that specify whether the variable represents character strings or bit strings and that specify the length to be maintained. The arithmetic data attributes are:

BINARY|DECIMAL

FIXED|FLOAT

(precision)

PICTURE

The string data attributes are:

BIT|CHARACTER

(length)

PICTURE

Other attributes can also be declared for data variables. The DEFINED attribute specifies that the data item is to occupy the same storage area as that assigned to other data. The storage class and scope attributes also apply to data.

Three other attributes apply only to data aggregates. For array variables, the dimension attribute specifies the number of dimensions and the bounds of an array. The ALIGNED and PACKED attributes specify the arrangement in storage of string or numeric character data elements within data aggregates.

PROGRAM CONTROL DATA

Attributes for program control data specify that the associated name is to be used by the programmer to control the execution of his program. The program control attributes are LABEL and POINTER.

ENTRY NAME ATTRIBUTES

The entry name attributes identify the name being declared as an entry name and describe features of that entry point. For example, the attribute BUILTIN specifies that the reference to the associated name within the scope of the declaration is interpreted as a reference to the built-in function or pseudo-variable of the same name. The entry name attributes are:

ENTRY

RETURNS

BUILTIN

FILE DESCRIPTION ATTRIBUTES

The file description attributes establish an identifier as a file name and describe characteristics for that file, e.g., how the data of the file is to be transmitted, whether records of a file are to be buffered. If the same file name is declared in more than one external procedure, the declarations must not conflict. Except for a file name parameter, a file name must always have the EXTERNAL attribute, either explicitly or by default; file name parameters cannot have a scope attribute.

The file description attributes are:

FILE

STREAM|RECORD

INPUT|OUTPUT|UPDATE

PRINT

SEQUENTIAL|DIRECT

BUFFERED|UNBUFFERED

BACKWARDS

ENVIRONMENT(option-list)

KEYED

SCOPE ATTRIBUTES

The scope attributes specify whether or not a name may be known in another external procedure. The scope attributes are EXTERNAL and INTERNAL. For a discussion of the scope of names, see Part I, Chapter 7, "Recognition of Names."

All external declarations for the same identifier in a program are linked as declarations of the same name. The scope of this name is the union of the scopes of all the external declarations for this identifier.

In all of the external declarations for the same identifier, the attributes declared must be consistent, since the declarations all involve a single name. For example, it would be an error if the identifier ID were declared as an EXTERNAL file name in one procedure and as an EXTERNAL entry name in another procedure in the same program.

The INTERNAL attribute specifies that the declared name cannot be known in any other block except those contained in the block in which the declaration is made. It cannot be specified for a file name.

The same identifier may be declared with the INTERNAL attribute in more than one block without regard to whether the attributes given in one block are consistent with the attributes given in another block, since the compiler regards such declarations as referring to different names.

STORAGE CLASS ATTRIBUTES

The storage class attributes are used to specify the type of storage for a data variable. The storage class attributes are:

STATIC

AUTOMATIC

BASED (pointer-variable)

ALPHABETIC LIST OF ATTRIBUTES

Following are detailed descriptions of the attributes, listed in alphabetic order. Alternative attributes are discussed together, with the discussion listed in the alphabetic location of the attribute whose name is the lowest in alphabetic order. A cross-reference to the combined discussion appears wherever an alternative appears in the alphabetic listing.

ALIGNED and PACKED (Array and Structure Attributes)

The ALIGNED and PACKED attributes specify the arrangement in storage of string or numeric character data elements within data aggregates. Either attribute may be specified for the name of a major structure or the name of an array that is not itself part of a structure.

PACKED specifies that each character string or numeric character field element is to be packed in storage contiguous with the character string or numeric character elements that surround it. If all the data elements of the aggregates are character string or numeric character items of the same type, there should be no unused storage between two adjacent elements. In other cases, some unused space may appear, but storage is to be conserved when possible. The PACKED attribute permits overlay defining.

ALIGNED allows the compiler to choose the alignment for each string data element within the aggregate to suit the environment. For System/360 implementations, the alignment is on byte boundaries. Two adjacent string or numeric character elements of an homogeneous aggregate with the ALIGNED attribute may not necessarily occupy contiguous storage, if a more efficient program is possible.

Note: The ALIGNED and PACKED attributes have no effect when the data itself requires full-word or double-word alignment.

General format:

ALIGNED|PACKED

General rules:

1. Arguments to be passed to the STRING built-in function must be PACKED structures.
2. The PACKED attribute cannot be specified for aggregates containing bit strings.
3. PACKED must be specified for data aggregates used in overlay defining.

Assumptions:

1. The default for major structures is PACKED.
2. The default for arrays that are not part of structures is ALIGNED.

AUTOMATIC, STATIC, and BASED (Storage Class Attributes)

The storage class attributes are used to specify the type of storage allocation to be used for data variables.

AUTOMATIC specifies that storage is to be allocated upon each entry to the block to which the storage declaration is internal. The storage is released upon exit from the block.

STATIC specifies that storage is to be allocated when the program is loaded and is not to be released until program execution has been completed.

The BASED (pointer-variable) attribute specifies a variable that is a description of data that can be applied to different locations in storage.

General format:

STATIC|AUTOMATIC|BASED(pointer-variable)

General rules:

1. AUTOMATIC and BASED variables can have INTERNAL scope only. STATIC variables may have either INTERNAL or EXTERNAL scope.

2. Storage class attributes cannot be specified for entry names, file names, members of structures, DEFINED data items, or parameters.

3. For a structure variable, a storage class attribute can be given only for the major structure name. The attribute then applies to all elements of the structure.

4. The following rules govern the use of based variables:

   a. The pointer variable must be explicitly declared with the POINTER attribute. The pointer variable must be an unsubscripted element variable and must not be an element of a structure; it cannot have the BASED attribute.

   b. When reference is made to a based variable, the data attributes assumed are those of the based variable, while the associated pointer variable identifies the location of data.

   c. A based variable may be used to identify and describe existing data or to obtain storage in a buffer by use of the LOCATE statement.

   d. The attribute EXTERNAL cannot appear with a based variable declaration, but a based variable can be used with an EXTERNAL pointer variable.

Assumptions:

1. If no storage class attribute is specified and the scope is INTERNAL, AUTOMATIC is assumed.

2. If no storage class attribute is specified and the scope is EXTERNAL, STATIC is assumed.

3. If neither the storage class nor the scope attribute is specified, AUTOMATIC is assumed.

BACKWARDS (File Description Attribute)

The BACKWARDS attribute specifies that the records of a SEQUENTIAL INPUT file on magnetic tape are to be accessed in reverse order, i.e., from the last record to the first record.

General format:

BACKWARDS

General rules:

1. The BACKWARDS attribute applies to RECORD files only; thus, it conflicts with the STREAM attribute.

2. The BACKWARDS attribute applies to tape files only.

3. The BACKWARDS attribute cannot be specified for variable length records.

BASED (Storage Class Attribute)

See AUTOMATIC.

BINARY and DECIMAL (Arithmetic Data Attributes)

The BINARY and DECIMAL attributes specify the base of the data items represented by the arithmetic variable as either binary or decimal.

General format:

BINARY|DECIMAL

General rule:

The BINARY or DECIMAL attribute cannot be specified with the PICTURE attribute.

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the

dimension, PACKED, ALIGNED, storage class, and scope attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the default attributes are FIXED BINARY (15). For identifiers beginning with any other alphabetic character the default attributes are FLOAT DECIMAL (6). If FIXED or FLOAT is declared, then DECIMAL is assumed. If DECIMAL or BINARY is declared, FLOAT is assumed. The default precisions are those defined for System/360 implementations.

Example:

```
DECLARE A BINARY, B DECIMAL;
```

The defaults for A are FLOAT(21); the defaults for B are FLOAT(6).

BIT and CHARACTER (String Attributes)

The BIT and CHARACTER attributes are used to specify string variables. The BIT attribute specifies a bit string. The CHARACTER attribute specifies a character string. The length attribute for the string must also be specified.

General format:

```
{BIT      }
{CHARACTER} (length)
```

General rules:

1. The length attribute specifies the length of the declared string. It must be a decimal integer constant, unsigned and greater than zero. The maximum length specification is 255 for character strings and 64 for bit strings.

2. The length attribute must immediately follow the CHARACTER or BIT attribute at the same factoring level with or without intervening blanks.

3. The BIT and CHARACTER attributes cannot be specified with the PICTURE attribute.

4. The PICTURE attribute can be used instead of CHARACTER to declare a character-string variable (see the PICTURE attribute).

5. All of the string attributes must be declared explicitly unless the PICTURE attribute is used. There are no defaults for string data.

6. Bit strings cannot appear in aggregates having the PACKED attribute.

BUFFERED and UNBUFFERED (File Description Attributes)

The BUFFERED attribute specifies that during transmission to and from external storage each record of a SEQUENTIAL RECORD file must pass through intermediate storage buffers that can be addressed through the use of based variables.

The UNBUFFERED attribute specifies that such records do not pass through buffers. No hidden buffers are used by the D-Compiler for UNBUFFERED files.

General format:

```
BUFFERED|UNBUFFERED
```

General rules:

1. The BUFFERED and UNBUFFERED attributes can be specified for SEQUENTIAL RECORD files only; thus, a file with the STREAM or DIRECT attribute cannot have one of these attributes.

2. The UNBUFFERED attribute must not be specified for variable length or blocked records.

3. The UNBUFFERED attribute can be specified only for files associated with magnetic tape or direct access devices.

Assumption:

Default is BUFFERED.

BUILTIN (Entry Attribute)

The BUILTIN attribute specifies that any reference to the associated name within the scope of the declaration is to be interpreted as a reference to the built-in function or pseudo-variable of the same name.

General format:

```
BUILTIN
```

General rules:

1. BUILTIN is used to refer to a built-in function or pseudo-variable in a block

that is contained in another block in which the same identifier has been declared to have another meaning.

2. If the BUILTIN attribute is declared for an entry name, the entry name can have no other attributes.

3. The BUILTIN attribute cannot be declared for parameters.

CHARACTER (String Attribute)

See BIT.

DECIMAL (Arithmetic Data Attribute)

See BINARY.

DEFINED (Data Attribute)

The DEFINED attribute specifies that the variable being declared is to represent part or all of the same storage as that assigned to other data. The DEFINED attribute can be declared for element, array, or major structure variables.

General format:

DEFINED base-identifier

The "base identifier" is the variable whose storage is also to be represented by the variable being declared.

Rules for defining:

1. The storage class and scope attributes cannot be specified for the defined item. It should be noted that although the base can have the EXTERNAL attribute, the defined item always has the INTERNAL attribute and cannot be declared with any scope attribute. If the base is external, its name will be known in all blocks in which it is declared external, but the name of the defined item will not. However, the value of the defined item will be changed if the value of the base item is changed in any block.

2. The base identifier must always be known within the block in which the defined item is declared. The base identifier cannot have the DEFINED attribute, it cannot be a based variable, and it cannot be a parameter.

3. The base identifier cannot be a minor structure or an element of a structure.

There are two types of defining, correspondence defining and overlay defining. If both the defined item and base identifier are arrays with the same number of dimensions, correspondence defining is in effect. In all other cases, overlay defining is in effect.

Correspondence Defining

Correspondence defining means that a reference to an element of the defined item is interpreted as a reference to the corresponding element of the base identifier.

Corresponding arrays must have the same number of dimensions and bounds. The elements of the base identifier and the elements of the defined item must have the same description.

Overlay Defining

Overlay defining means that the defined item is to occupy part or all of the storage allocated to the base. In this way, changes to the value of either variable may be reflected in the value of the other. Overlay defining is permitted between the items shown in Figure I-1:

Rule for overlay defining:

The extent of the defined item must not be larger than the extent of the base. Extent is calculated by summing the lengths of the parts of the data, including all individual elements of arrays.

Dimension (Array Attribute)

The dimension attribute specifies the number of dimensions of an array and the bound of each dimension.

General format:

(bound[,bound[,bound]])

Defined Item	Base Identifier
A coded arithmetic element variable	An unsubscripted coded arithmetic element variable of the same base, scale, and precision
An element label variable	An unsubscripted element label variable
An element pointer variable	An unsubscripted element pointer variable
A character class[1] variable	Character class[1] data
A structure	An identical structure whose makeup is such that matching pairs of items from the structure are valid examples for overlay defining of coded arithmetic, label, and pointer element variables. The elements can also be strings or numeric character data items of matching lengths.

[1]The character class consists of:

a. Numeric character data
b. Character strings
c. Packed structures consisting of items a, b, and d
d. Packed arrays consisting of items a and b

Figure I-1. Permissible Items for Overlay Defining

General rules:

1. The number of bounds specifies the number of dimensions in the array. As shown by the general format, the maximum number of dimensions allowed by the D-Compiler is three.

2. Each bound must be an unsigned decimal integer constant greater than zero. This number specifies the upper bound of the corresponding dimension. The lower bound is always assumed to be 1. Therefore, this number also specifies the extent of the corresponding dimension. For example, if a bound is 8, the extent of that dimension is 1, 2,...,8.

3. The dimension attribute must immediately follow the array name. Intervening blanks are optional. It cannot be factored.

DIRECT and SEQUENTIAL (File Description Attributes)

The DIRECT and SEQUENTIAL attributes specify the manner in which the records of a RECORD file are to be accessed. SEQUENTIAL specifies that the records are to be accessed according to their logical sequence in the data set. DIRECT specifies that the records of the file are to be accessed by use of a key. Each record of a direct file must, therefore, have a key associated with it.

Note that SEQUENTIAL and DIRECT specify only the current usage of the file; they do not specify physical properties of the data set associated with the file.

General format:

```
SEQUENTIAL|DIRECT
```

General rules:

1. DIRECT files must also have the KEYED attribute which is implied by DIRECT. SEQUENTIAL files must not have the KEYED attribute.

2. The DIRECT and SEQUENTIAL attributes cannot be specified with the STREAM attribute.

Assumption:

Default is SEQUENTIAL for RECORD files.

ENTRY Attribute

The ENTRY attribute specifies that the identifier being declared is an entry name.

General format:

ENTRY

General rules:

1. The ENTRY attribute must be specified for any entry name that is declared elsewhere and not known within the block if any reference is made to that entry name (such as in an argument list) unless, within the block:

   a. The entry name appears in a CALL statement or a function reference with an argument list, either of which constitutes a contextual declaration of the ENTRY attribute, or

   b. The entry name is declared to have one of the attributes BUILTIN or RETURNS, the latter of which implies ENTRY. The ENTRY attribute cannot be specified for a name that is given the BUILTIN attribute.

2. The ENTRY attribute must be explicitly declared or implied for an entry name that is a parameter.

3. The ENTRY attribute can be declared for an INTERNAL entry name only within the block to which the name is internal. Internal procedures declared with an ENTRY attribute must also be given the INTERNAL attribute in the same declaration.

Assumptions:

The ENTRY attribute can be assumed either contextually or by implication, as described in Rule 1. The appearance of a name as a label of either a PROCEDURE statement or an ENTRY statement constitutes an explicit declaration of that identifier as an entry name.

ENVIRONMENT (File Description Attribute)

The ENVIRONMENT attribute is an implementation-defined attribute that specifies various file characteristics that are not part of the PL/I language.

General format:

ENVIRONMENT (option-list)

The option-list is defined individually for each implementation of PL/I. For the D-level compiler, it is as follows:

```
[CONSECUTIVE                ]
[REGIONAL ({1|3})           ]

{F (blocksize [,recordsize] )}
{V (maxblocksize)            }
{U (maxblocksize)            }

[BUFFERS (n)]
MEDIUM(logical-device-name,
     physical-device-type)
[LEAVE] [NOLABEL] [VERIFY]
[KEYLENGTH (decimal-integer-constant)]
```

General rules:

1. Each file declaration must have an associated ENVIRONMENT attribute.

2. The options must be separated by one or more blanks.

3. The CONSECUTIVE option implies that the (n+1)th record of the file is located after the nth record of that file. An example of an I/O device for which the CONSECUTIVE option is mandatory is a card reader or a printer. If neither the CONSECUTIVE or the REGIONAL option is specified, the CONSECUTIVE option is assigned by default.

4. The REGIONAL option implies that the physical location of a record on a storage medium is specified by a key. The key is specified by the programmer and constitutes the only way to access the record. The REGIONAL option is permitted only for direct access files.

   REGIONAL(1) is used for files where records are referred to by their relative location with respect to the first record in the file. The relative record number is specified in the KEY or KEYFROM options.

   REGIONAL (3) is used for files where the records are referred to by the location of the track containing this record relative to the first track in the file and a key associated with the record. Both the key and the relative track number (which is a part of the key) are specified in the options KEY or KEYFROM.

5. The F, V, and U options are used to describe physical records. F specifies fixed length records, V specifies variable length records and U specifies records of undefined length.

   Fixed-length records require a block size specification. The record size specification is optional. Both block size and record size are specified by

means of unsigned decimal integer constants. The quotient of block size divided by record size must be an integer. Fixed-length blocked records are constructed if both block size and record size are specified. The blocking factor is the block size divided by the record size. If only the block size is specified, the record size is assumed to be equal to the block size, and the file is considered to be unblocked.

If fixed-length blocked records are to be transferred by a READ SET or LOCATE statement, the record size must be divisible by 8.

When using the V option, the record size for records to be transferred by means of READ SET or LOCATE statements must yield a remainder of 4 after division by 8.

6. The BUFFERS(n) option, where n must be 1 or 2, is used to specify the number of buffers to be used. The BUFFERS option may be used for STREAM files even though neither the BUFFERED nor the UNBUFFERED attributes are permitted since STREAM files have hidden buffers. The BUFFERS option may also be used for BUFFERED RECORD files. The UNBUFFERED attribute precludes the use of the BUFFERS option. The default is BUFFERS(1).

7. The MEDIUM option is used to specify the logical unit name and the device type for the file being declared.

   The logical device name has the form SYSxxx, where xxx may be:

   a. IPT - System input device

   b. LST - System output device used for listing

   c. PCH - System output device (card punch)

   d. 000 through 222 - Logical units SYS000 through SYS244

   The device-type specification contains the number of the device to be used. For instance, if the IBM 1442N1 Card Read/Punch is to be used, the option would be written as 1442. Figure I-2 shows how the individual device types are specified.

   The device types listed in Figure I-2 may be assigned to the logical unit names SYSIPT, SYSLST, and SYSPCH as shown in Figure I-3.

8. The LEAVE option is used to specify that no rewind operation is to be performed at file open or close time. It should be given for files that have the BACKWARDS attribute to ensure proper positioning of the file.

9. The NOLABEL option is used to specify that no file labels are to be processed for a magnetic tape file.

   If the NOLABEL option is specified for output files, a tape mark is automatically written as the first record on the tape. Non-standard labels and additional user labels are not processed.

10. The VERIFY option is used to specify that a read-check is to be performed after every write operation. This option is permitted only with direct-access devices.

11. The KEYLENGTH option is used to specify the length of the key for input and output operations. This option is permitted only with the option REGIONAL (3). The minimum key-length is 9.

    Note: The key length must not be included in the record length.

Device Type	Number	Device-Type Specification
Card Readers and Punches	IBM 2540 (reader)	2540
	IBM 2540 (punch)	2540
	IBM 1442N1	1442
	IBM 1442N2	1442
	IBM 2520B1	2520
	IBM 2520B2	2520
	IBM 2520B3	2520
	IBM 2501	2501
Printers	IBM 1403	1403
	IBM 1404	1404
	IBM 1443	1443
	IBM 1445	1445
Magnetic Tape Drives	IBM 2400 (9-track)	2400
	IBM 2400 (7-track)	2400
Disk Drives	IBM 2311	2311

Figure I-2. Device Types and Corresponding Specifications

Logical Unit Name	Device Type
SYSIPT	IBM 2540 (reader) IBM 1442N1 IBM 2501 IBM 2520B1 IBM 2400 (7- or 9-track)
SYSLST	IBM 1403 IBM 1404 IBM 1443 IBM 2400 (7- or 9-track)
SYSPCH	IBM 2540 (punch) IBM 1442N1 IBM 1442N2 IBM 2520B1 IBM 2520B2 IBM 2520B3 IBM 2400 (7- or 9-track)

Figure I-3. Device Types Associated to SYSIPT, SYSLST, and SYSPCH

Assumptions:

CONSECUTIVE data set organization is assumed unless stated otherwise. Tape reels are rewound unless the LEAVE option is specified. If the BUFFERS(n) option is not specified, one buffer is allocated.

EXTERNAL and INTERNAL (Scope Attributes)

The EXTERNAL and INTERNAL attributes specify the scope of a name. INTERNAL specifies that the name can be known only in the declaring block and its contained blocks. EXTERNAL specifies that the name may be known in other blocks containing an external declaration of the same name.

General format:

```
EXTERNAL|INTERNAL
```

General rules:

1. All file names must be external. They cannot be declared as internal.
2. All external names are restricted by the D-Compiler to a maximum length of six characters.

Assumptions:

INTERNAL is assumed for entry names of internal procedures and for variables with any storage class. EXTERNAL is assumed for file names and entry names of external procedures.

FILE Attribute

The FILE attribute specifies that the identifier being declared is a file name.

General format:

```
FILE
```

General rule:

The FILE attribute must be explicitly declared for each file name and file name parameter. It must be the first attribute in a file declaration.

FIXED and FLOAT (Arithmetic Data Attributes)

The FIXED and FLOAT attributes specify the scale of the arithmetic variable being declared. FIXED specifies that the variable is to represent fixed-point data items. FLOAT specifies that the variable is to represent floating-point data items.

General format:

```
FIXED|FLOAT
```

General rule:

The FIXED and FLOAT attribures cannot be specified with the PICTURE attribute.

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the dimension, PACKED, ALIGNED, storage class, and scope attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the default attributes are FIXED BINARY (15). For identifiers beginning with any other alphabetic character, the default attributes are FLOAT DECIMAL (6). If BINARY or DECIMAL is specified, FLOAT is assumed. If FIXED or FLOAT is specified, DECIMAL is assumed. The default precisions are those defined for System/360 implementations.

FLOAT (Arithmetic Data Attribute)

See FIXED.

INPUT, OUTPUT, and UPDATE (File Description Attributes)

The INPUT, OUTPUT, and UPDATE attributes indicate the function of the file. INPUT specifies that data is to be transmitted from the data set to the program. OUTPUT specifies that data is to be transmitted from the program to the data set, not an existing data set, but a newly created one. UPDATE specifies that the data can be transmitted in either direction; that is, the file is both an input and an output file.

General format:

INPUT|OUTPUT|UPDATE

General rules:

1. A file with the INPUT attribute cannot have the PRINT attribute.
2. A file with the OUTPUT attribute cannot have the BACKWARDS attribute.
3. A file with the UPDATE attribute cannot have the STREAM, BACKWARDS, or PRINT attributes. A declaration of UPDATE for a SEQUENTIAL file indicates the update-in-place mode. To access such a file, the sequence of statements must be READ, then REWRITE.
4. One of the above attributes must be given for each file unless the file has been declared with the PRINT attribute, in which case, OUTPUT is implied.
5. These attributes must be specified in the DECLARE statement except in the case of an UNBUFFERED file, in which case, INPUT or OUTPUT can be specified in the OPEN statement.

Assumption:

The PRINT attribute implies OUTPUT.

INTERNAL (Scope Attribute)

See EXTERNAL.

KEYED (File Description Attribute)

The KEYED attribute specifies that each record in the file has a key associated with it, and that the statement options KEY and/or KEYFROM may be used to access records in the file.

General format:

KEYED

General rules:

1. A KEYED file cannot be read sequentially.
2. The KEYED attribute can be specified for DIRECT files only.

Assumption:

The DIRECT attribute implies KEYED.

LABEL (Program Control Data Attribute)

The LABEL attribute specifies that the identifier being declared is a label variable and is to have statement labels as values.

General format:

LABEL

General rules:

1. The variable can have as values any of the statement labels known within the scope of the variable.
2. If the variable is a parameter, its value can be any statement label variable or constant passed as an argument.
3. An entry name cannot be a value of a label variable.

Length (String Attribute)

See BIT.

OUTPUT (File Description Attribute)

See INPUT.

PACKED (Array and Structure Attribute)

See ALIGNED.

PICTURE (Data Attribute)

The PICTURE attribute is used to define the internal and external formats of character-string and numeric character data and to specify the editing of data. Numeric character data is data having an arithmetic value but stored internally in character form. Numeric charcter data must be converted to coded arithmetic before arithmetic operations can be performed.

The picture characters are described in Section D, "Picture Specification Characters."

General format:

PICTURE

{'character-picture-specification'
 'numeric-picture-specification'}

A "picture specification," either character or numeric, is composed of a string of picture characters enclosed in single quotation marks (as shown in the format). An individual picture character may be preceded by a repetition factor, which is an unsigned decimal integer constant greater than zero, n, enclosed in parentheses, to indicate repetition of the character n times. Picture characters in a specification are considered to be grouped into fields, some of which contain subfields.

General rules:

1. The "character picture specification" is used to describe a character-string data item. Only the picture character X can be used. It indicates that the associated position in the data item can contain any character. At least one X must be specified. A character picture specification is a single field with no contained subfields.

   Example:

   ```
   DECLARE ORDER# PICTURE '(13)X';
   ```

   This declaration specifies that values of ORDER# are to be character strings of length 13. For example, the character string 'GF342-63-0024' would fit this description.

   Editing and suppression characters are not allowed in character picture specifications. Each picture specification character must represent an actual character in the data item.

2. The "numeric picture specification" is used to describe a character item that represents an arithmetic value or a character-string value, depending on its use. A numeric picture specification can consist of one or more fields, some of which can be divided into subfields. A single field is used to describe a fixed-point number or the mantissa of a floating-point number. Either may be divided into two subfields, one describing the integer portion, the other describing the fractional portion. For floating-point numbers, a second field is required to describe the exponent; it cannot be divided into subfields. Four basic picture characters can be used in a numeric picture specification:

   9 indicating any decimal digit

   V indicating the assumed location of a decimal point. It does not specify an actual character in the character-string value of the data item. It indicates the end of a subfield of a picture specification.

   K indicating, for floating-point data items, that the exponent should be assumed to begin at the position associated with the picture character following the K. It does not specify an actual character in the character-string value of the data item. The K delimits the two fields of the specification.

   E indicating, for floating-point data items, that the associated position will contain the letter E to indicate the beginning of the exponent. The E also delimits the two fields.

   In addition to these characters, zero suppression characters, editing characters, and sign characters may be included in a numeric picture specification to indicate editing. Editing characters are not a part of the arithmetic value of a numeric character data item, but they are a part of its character-string value. Each numeric picture specification must include at least one digit specifier. Repetition factors are allowed in numeric picture specifications.

3. A numeric character data item can have only a decimal base. Its scale and precision are specified by the picture characters. The PICTURE attribute cannot be specified in combination with base, scale, or precision attributes.

4. The following paragraphs indicate the combinations of picture characters for different arithmetic data formats.

   a. Decimal fixed-point items are described in the following general form:

      PICTURE '[9]...[V][9]...'

      Sign, editing, and zero suppression picture characters can be included in a fixed-point specification. The V may not appear more than once in a specification, although it may be used in combination with the decimal point (.) or comma (,) editing characters, which cause insertion of a period or comma. If no V is included, the decimal point is assumed to be to the right of the rightmost digit. Only one sign indication can be included in the field. The specification must include at least one digit position.

      Example:

      DECLARE A PICTURE '999V99';

      This specification describes numeric character items of five digits, two of which are assumed to be fractional digits.

   b. Decimal floating-point items are described by the following general form:

      PICTURE '[9]...[V][9]...[E|K]9[9]'

      Both the first field and the exponent field must each contain at least one digit position. The exponent field can contain no more than two digits, since System/360 implementations allow only two digits in the exponent field of a decimal floating-point number. If arithmetic data items are to be assigned to the described variable, the exponent field must contain both of the allowed digit specification characters, or the second digit of the exponent field will be lost and the SIZE condition will be raised.

      Sign, editing, and zero suppression picture characters can be included in a floating-point specification. One sign indication is allowed for each field. Only one V is allowed, and it can appear in the first field only. As with fixed-point specifications, the V may appear in combination with the decimal point editing character (as .V or V.). X, T, I, R, CR, DB, and sterling picture characters are not allowed.

5. The precision of a numeric character variable is dependent upon the number of digit positions, actual and conditional. Digit positions can be specified by the following characters:

   9 which is an actual digit character

   Z, * } which are conditional digit characters specifying zero suppression

   T, I, R } which are digit characters specifying an overpunch

   $, S, +, - } which are conditional digit drifting characters

   Each but the first conditional digit drifting character in a drifting string specifies a digit position. A conditional digit drifting character used alone does not specify a digit position. Precision of a fixed-point variable is (p,q), where p is the number of digit positions in the picture specification and q is the number of digit positions following V. Precision of a floating-point variable is (p), where p is the number of digit positions preceding the E or K. Indicated static editing characters or insertion characters do not participate in the specification of precision, but they must be counted in the number of characters if the data item is assigned internally to a character string.

6. A variable representing sterling data items can be specified by using a numeric picture specification that consists of three fields, one each for pounds, shillings, and pence. The

pence field can be divided into two subfields. Data so described is stored in character format as three contiguous numbers corresponding to each of the three fields. If any arithmetic operations are specified for the variable, its value is converted to coded fixed-point decimal representing the value in pence. Sterling picture specifications have the following form:

```
PICTURE
   'G [editing-character-1]...
   M pounds-field
   M [separator-1]...
         shillings-field
   M  [separator-2]...
         pence-field
      [editing-character-2]...'
```

Picture specifications characters, editing characters, and separators that can be used in any of these fields are discussed in Section D, "Picture Specification Characters." The precision (p,q) of a sterling data item is defined as follows:

q = number of fractional digits in the pence field.

p = 3 + q + the number of digit positions, actual and conditional, in the pounds field.

POINTER (Program Control Data Attribute)

The POINTER attribute specifies that the identifier being declared is a pointer variable and can be used to identify data existing in any storage class.

General format:

```
POINTER
```

General rules:

1. The POINTER attribute can be given to an identifier only via a DECLARE statement. Thus, a pointer variable must be explicitly declared with the POINTER attribute.

2. The value of a pointer variable can be established in two ways:

   a. by pointer assignment.

   b. by the SET clause in a READ or LOCATE statement.

3. Pointer data cannot appear as an operand in an arithmetic expression, nor can conversions be performed between pointer data and other data types.

4. The only operators that can be applied directly to pointer data are the comparison operators = and ¬=.

5. Pointer data cannot be read or written via STREAM transmission.

6. A pointer variable cannot have the BASED attribute. Therefore, a pointer variable cannot be an element of a structure having the BASED attribute.

Precision (Arithmetic Data Attribute)

The precision attribute is used to specify the minimum number of significant digits to be maintained for the values of variables, and to specify, for fixed-point decimal variables, the scale factor (the assumed position of the decimal point). The precision attribute applies to both binary and decimal data.

General format:

```
(number-of-digits  [,scale-factor])
```

The "number of digits" and "scale factor" are unsigned decimal integer constants. The "number of digits" cannot be zero. The precision attribute specification is often represented, for brevity, as (p,q), where p represents the "number of digits" and q represents the "scale factor."

General rules:

1. The precision attribute must immediately follow, with or without intervening blanks, the scale (FIXED or FLOAT), or base (DECIMAL or BINARY) attribute at the same factoring level.

2. The "number of digits" specifies the number of digits to be maintained for data items assigned to the variable. The scale factor specifies the number of fractional digits. No point is actually present; its location is assumed.

3. The "scale-factor" is a decimal integer constant that states the number of digits to the right of the decimal point. It can be used only with decimal fixed-point variables. A binary fixed-point variable may represent only integer numbers and therefore always has an assumed scale factor of zero.

4. When the scale factor is not specified for decimal fixed-point data, it is assumed to be zero; that is, the variable is to represent integers.

5. The scale factor can be larger than the number of digits. Such a scale factor always specifies a fraction, with the decimal point assumed to be located the specified number of digit places to the left of the rightmost actual digit. Intervening zeros are assumed, but they are not stored; only the specified number of digits are actually stored.

6. The precision attribute cannot be specified in combination with the PICTURE attribute.

7. The maximum number of digits allowed for System/360 implementations is 15 for decimal fixed-point data, 31 for binary fixed-point data, 16 for decimal floating-point data, and 53 for binary floating-point data. For the D-Compiler the scale factor cannot be greater than 15.

Assumptions:

The defaults for the D-Compiler are as follows:

(5,0) for DECIMAL FIXED

(15) for BINARY FIXED

(6) for DECIMAL FLOAT

(21) for BINARY FLOAT

PRINT (File Description Attribute)

The PRINT attribute specifies that the data of the file is ultimately to be printed. The PAGE, LINE, and SKIP options of the PUT statement and the PAGESIZE option of the OPEN statement can be used only with files having the PRINT attribute. These options are described in Section J, "Statements".

General format:

PRINT

General rules:

1. The PRINT attribute implies the OUTPUT and STREAM attributes.

2. The PRINT attribute causes the initial data byte within each record to be reserved for ASA printer control characters. Any length specification of the record must be 1 plus the length of the print line to account for this control character. These control characters are set by the PAGE, SKIP, or LINE format items or options.

Assumption:

If no FILE or STRING specification appears in a PUT statement, the standard output file is assumed.

RECORD and STREAM (File Description Attributes)

The RECORD and STREAM attributes specify the kind of data transmission to be used for the file. STREAM indicates that the data of the file is considered to be a continuous stream of data items, in character form, to be assigned from the stream to variables, or from expressions into the 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 directly to or from a variable or directly to or from a buffer.

General format:

RECORD|STREAM

General rules:

1. A file with the STREAM attribute can be specified only in the OPEN, CLOSE, GET, and PUT statements.

2. A file with the RECORD attribute can be specified only in the OPEN, CLOSE, READ, WRITE, REWRITE, and LOCATE statements.

3. A file with the STREAM attribute cannot have any of the following attributes: UPDATE, DIRECT, SEQUENTIAL, BACKWARDS, BUFFERED, UNBUFFERED, and KEYED.

4. A file with the RECORD attribute cannot have the PRINT attribute.

Assumptions:

Default is STREAM.

RETURNS (Entry Name Attribute)

The RETURNS attribute may be specified in a DECLARE statement for an entry name that is used in a function reference within the scope of the declaration. It is used to describe the attributes of the function value returned when that entry name is invoked as a function.

General format:

```
RETURNS (attribute...)
```

It is used in the following manner:

```
DECLARE entry-name [ENTRY]
    RETURNS (attribute...);
```

General rules:

1. The RETURNS attribute implies the ENTRY attribute, and, hence, ENTRY can be omitted.

2. The attributes in the parenthesized list following the keyword RETURNS are separated by blanks. They must agree with the attributes specified (or assumed by default) in the PROCEDURE or ENTRY statement to which the entry name is prefixed. If the attributes of the actual value returned do not agree with those declared with the RETURNS attribute, no conversion will be performed.

3. Only arithmetic, string, PICTURE, or POINTER attributes can be specified.

4. Internal procedures declared with a RETURNS attribute must also be given the INTERNAL attribute in the same declaration. For an internal function, the RETURNS attribute can be specified only in a DECLARE statement that is internal to the same block as the function procedure.

5. Unless default attributes for the entry name apply, any invocation of a function must appear within the scope of a RETURNS attribute declaration for the entry name.

Assumptions:

If the RETURNS attribute is not specified within the scope of a function reference, the defaults assumed for the returned value are FIXED BINARY (15) if the entry name begins with any of the letters I through N; otherwise, the defaults are FLOAT DECIMAL (6). Default precisions are those defined for System/360 implementations.

SEQUENTIAL (File Description Attribute)

See DIRECT.

STATIC (Storage Class Attribute)

See AUTOMATIC.

STREAM (File Description Attribute)

See RECORD.

UPDATE (File Description Attribute)

See INPUT.

This section presents the PL/I statements in alphabetical order. Most statements are accompanied by the following information:

1. Function -- a short description of the meaning and use of the statement
2. General format -- the syntax of the statement
3. Syntax rules -- rules of syntax that are not reflected in the general format
4. General rules -- rules governing the use of the statement and its meaning in a PL/I program

The Assignment Statement

Function:

The assignment statement evaluates expressions and assigns values to elements, arrays, or structures.

General formats:

The assignment statement has five general format types. They are as shown in Figure J-1.

Syntax rules:

1. In Type 1, the variable in the receiving field (i.e., to the left of the equal sign) must represent a single

Type 1. Element Assignment

```
{element-variable}
{pseudo-variable }  =  element-expression;
```

Type 2. Array Assignment

```
                          {array-expression  }
array-variable        = {element-expression};
```

Type 3. Structure Assignment

```
                          {structure-expression}
structure-variable    = {element-expression  } ;
```

Type 4. Statement Label Assignment (2 forms)

```
a. element-label-variable = {label-constant          }
                            {element-label-variable  } ;

                          {label-constant          }
b. label-array = {element-label-variable   } ;
                 {label-array              }
```

Type 5. Pointer Assignment (2 forms)

```
a. element-pointer-variable = element-pointer-expression;

                          {element-pointer-expression}
b. pointer-array = {pointer-array             } ;
```

Figure J-1. Assignment Statement Types

element whose data type is arithmetic or string.

2. In Type 2, the variable in the receiving field must represent an array of arithmetic or string elements.

   If an element expression appears to the right of the equal sign, the value of the expression is assigned to each element of the array in the receiving field.

   If an array expression appears to the right of the equal sign, all of the arrays in the receiving field and all array operands in the expression must have the same number of dimensions and identical bounds.

3. In Type 3, the variable in the receiving field must represent a structure and each element of the structure must be an arithmetic or string element. (Pointer and label elements are also allowed, but these are special cases; see general rule 3.)

   If an element expression appears to the right of the equal sign, the value of the expression is assigned to every element of the structure in the receiving field.

   If a structure expression appears to the right of the equal sign, then, the relative structuring of all structures on both sides must be the same.

4. In Type 4, item b, if a label constant or an element label variable appears on the right, then the constant or the value of the variable is assigned to every element in the label array in the receiving field.

   If a label array appears on the right, then the number of dimensions and the bound of each dimension of that array must be identical to those of the label array in the receiving field.

5. In Type 5, an "element pointer expression" is either an element pointer variable or a function reference that returns an element pointer value.

   In Type 5, item b, if an element pointer expression appears to the right of the equal sign, the value of the expression is assigned to every element of the pointer array in the receiving field.

   Also item b, if a pointer array appears to the right, the number of dimensions and the bound of each dimension of that array must be identical to those of the pointer array in the receiving field.

General rules:

1. The assignment statement is evaluated as follows:

   a. For Types 1, 4, and 5, any expressions that appear in the receiving field, either in subscripts or in pseudo-variables, are evaluated from left to right. The expression on the right of the equal sign is evaluated and its value is assigned to the variable in the receiving field.

   b. For Types 2 and 3, the assignment statement is treated as a sequence of element assignment statements involving corresponding elements of the arrays or structures concerned. For arrays, the elements are assigned in row-major order; for structures, the elements are assigned in the order in which they were declared.

      Note that the result of the evaluation for a later position in an array or structure may be affected by the evaluation and assignment to an earlier position (see Example 1 below).

   c. When necessary, the value of the expression on the right is converted to the characteristics of the variable in the receiving field according to the rules given in Section F, "Data Conversion."

2. When a variable in the receiving field is a string or the UNSPEC pseudo-variable, the expression on the right is evaluated as in general rule 1, and the assignment is performed from left to right, starting with the leftmost bit or character position. The following may also apply:

   a. If the value of the expression is longer than the string, the value is truncated on the right to match the length of string.

   b. If the value of the expression is shorter than the string, the value is extended on the right with zeros for bit strings and with blanks for character strings.

3. If a pointer or label variable is an element of a structure appearing in a receiving field, it is assigned a value just like any other element in the structure. However, no conversion

is performed and, therefore, the value assigned to such a pointer or label variable must be another pointer or label variable.

4. Label array and pointer array assignment as shown in Types 4 and 5, respectively, follow the rules given for array assignment in general rule 1.

Example 1:

The following example illustrates array assignment:

```
Given the array A    2   4
                     3   6
                     1   7
                     4   8

and the array B      1   5
                     7   8
                     3   4
                     6   3
```

Consider the assignment statement:

```
A = (A+B)**2-A(1,1);
```

After execution, A has the value:

```
     7    74
    93   189
     9   114
    93   114
```

Note that the new value for A(1,1), which is 7, is used in evaluating the expression for all other elements.

Example 2:

The following example illustrates string assignment:

Given:

```
A  is a string whose value is 'XZ/BQ'.
B is a string whose value is 'MAFY'.
C is a string of length 3.
D is a string of length 5.
```

Then in the statement:

```
C = A, the value of C is 'XZ/'.
C = 'X', the value of C is 'Xbb'.
D = B, the value of D is 'MAFYb'.
D = SUBSTR   (A,2,3)||SUBSTR  (A,2,3),
    the value of D is 'Z/BZ/'.
SUBSTR (A,2,4) = B, the value of A is
    'XMAFY'.
SUBSTR  (B,2,2) = 'R',  the value of B
    is 'MRbY'.
```

Example 3:

The following example (where A, B, and C are element variables) illustrates element assignment:

```
A=A+SIN(B) + C**2;
```

Example 4:

The following examples illustrate structure assignment:

a.
```
DECLARE 1 X, 2 Y, 2 Z, 2 R, 3 S, 3 P,
        1 A, 2 B, 2 C, 2 D, 3 E, 3 Q;

X = X * A;
```

The assignment statement is equivalent to the following statements:

```
X.Y = X.Y * A.B;
X.Z = X.Z * A.C;
X.S = X.S * A.E;
X.P = X.P * A.Q;
```

b.
```
DECLARE 1 A, 2 B, 2 C, 3 D, 3 E;

A = A + A.B;
```

The assignment statement is equivalent to the following:

```
A.B = A.B + A.B;
A.C = A.C + A.B;
```

The last statement is equivalent to:

```
A.D = A.D + A.B;
A.E = A.E + A.B;
```

Example 5:

The following example illustrates statement label assignment:

```
    DECLARE P LABEL;
    P = A;
    GO TO P;
      .
      .
      .
A:  X = Y**2;
```

This set of statements causes control to transfer to A when the GO TO P statement is executed.

Example 6:

The following example illustrates conversion of data defined by a picture description assigned to floating-point data, and vice versa:

```
DECLARE A FLOAT, B PICTURE '999V99';

A = B; (B is converted from fixed-
       point to floating-point.)

B = A; (A is converted from floating-
       point to fixed-point.)
```

The BEGIN Statement

Function:

The BEGIN statement heads and identifies a begin block.

General format:

```
BEGIN;
```

General rules:

1. A BEGIN statement is used in conjunction with an END statement to delimit a begin block. A complete discussion of begin blocks can be found in Part 1, Chapter 6, "Blocks, Flow of Control, and Storage Allocation."

2. A RETURN statement cannot appear within a begin block.

The CALL Statement

Function:

The CALL statement invokes a procedure and causes control to be transferred to the specified entry point of that procedure.

General format:

```
CALL entry-name
   [(argument[,argument]...)];
```

Syntax rules:

1. The entry name represents an entry point of the procedure that is being invoked.

2. An argument can be any expression except a based variable, a built-in function name, an operational structure expression, or an operational array expression. Examples of valid arguments include minor structure names, label variables, entry names, pointer expressions, string constants, array names, and file names. Note, however, that if the attributes of an argument are not consistent with those of its corresponding parameter, no conversion is performed and an error will probably result.

General rule:

See Part 1, Chapter 10, "Subroutines and Functions" for detailed descriptions of the interaction of arguments with the parameters that represent these arguments in the invoked procedure.

The CLOSE Statement

Function:

The CLOSE statement dissociates the named file from the data set with which it was associated by a previous opening. It also dissociates from the specified file, either INPUT or OUTPUT and PAGESIZE, if specified in the opening of that file. However, all attributes explicitly specified for that file in a DECLARE statement remain in effect.

General format:

```
CLOSE FILE(file-name)
        [,FILE(file-name)]... ;
```

General rules:

1. The "file name" in the FILE(file-name) specification indicates the file to be closed. Since more than one such specification can be given in a CLOSE statement, more than one file can be closed by one CLOSE statement.

2. A closed file can be reopened.

3. Closing an unopened file, or a previously closed file, has no effect.

4. If a file is not closed by a CLOSE statement, it is automatically closed at the completion of the program in which it was opened.

The DECLARE Statement

Function:

The DECLARE statement is the principal method for explicitly declaring attributes of names.

General format:

```
DECLARE
  [level] identifier [attribute]...
  [,[level] identifier [attribute]...]...;
```

Syntax rules:

1. "Level" is a nonzero unsigned decimal integer constant. It can appear only in structure declarations; the major structure must have the level 1. A blank space must separate a level number from the identifier following it.

2. In general, attributes must immediately follow the identifier to which they apply (as shown in the general format). However, attributes common to several name declarations can be factored to eliminate repeated specification of the same attribute for many identifiers. Factoring is achieved by enclosing the involved declarations (non-common attributes included) in parentheses and following this by the set of common attributes. In the case of a factored level number, the level number must precede the parenthesized list (a blank is not required between the factored level number and the left parenthesis). Dimension and file description attributes cannot be factored. Factoring can be nested up to a level of eight. For examples of factoring, see "Factoring of Attributes" in Section I, "Attributes."

General rules:

1. A major structure identifier or an identifier not contained within a structure can be specified in only one DECLARE statement within a particular block. All attributes given explicitly for that identifier must be declared together in that DECLARE statement. (Note, however, that certain identifiers having the FILE attribute may be given the INPUT or OUTPUT attribute in an OPEN statement as well. See "The OPEN Statement" in this section and in Part I, Chapter 8, "Input and Output," for further information.)

2. Attributes of external names, in separate blocks and compilations, must be consistent.

3. Labels may be prefixed to DECLARE statements (however, such labels are treated as comments and, hence, have no meaning). Condition prefixes cannot be attached to a DECLARE statement.

4. File names must be explicitly declared, and the first attribute in a file declaration must be FILE.

The DISPLAY Statement

Function:

The DISPLAY statement causes a message to be displayed to the machine operator. An option allows the machine operator to reply.

General format:

```
DISPLAY(element-expression)
[REPLY(character-string-element-variable)];
```

Syntax rule:

The "character-string element variable" cannot be a pseudo-variable.

General rules:

1. Execution of the DISPLAY statement causes the element expression to be evaluated, and, where necessary, converted to a character string. This character string is the message to be displayed to the machine operator.

   For the D-compiler, it can be no more than 80 characters long.

2. If the REPLY option is specified, the machine operator will respond with a message that is to be assigned to the character-string element variable specified in the option. The D-Compiler does not restrict the length of the reply; however, the character string variable, like any other character string, cannot exceed 255 characters.

3. If the REPLY option is not specified, execution continues uninterrupted after the execution of the DISPLAY statement.

4. If the REPLY option is specified, execution of the program is suspended until the operator's reply has been completed.

```
Type 1.  DO;

Type 2.  DO WHILE (element-expression);

Type 3.  DO variable =specification [,specification]...;

            where "specification" has the following form:

                                 ┌TO expression2 [BY expression3]┐
            expression1          └BY expression3 [TO expression2]┘ [WHILE(expression4)]
```

Figure J-2. General Format of DO Statement

The DO Statement

Function:

The DO statement heads a DO-Group and can also be used to specify repetitive execution of the statements within the group.

General formats:

The three format types for the DO statement are shown in Figure J-2.

Syntax rules:

1. In all three types, the DO statement is used in conjunction with the END statement to delimit a DO-group. Only Type 1 does not provide for the iterative execution of the statements within the group.

2. In Type 3, the "variable" must represent a single element; it cannot be subscripted. Arithmetic variables are generally used, but label, pointer, and string variables are allowed, provided that the expansions given in the general rules below result in valid PL/I programs. Note, however, that if "variable" is neither arithmetic nor bit string "expression2" and "expression3" must be omitted.

3. Each expression in a specification must be an element expression.

4. If "BY expression3" is omitted from a "specification," and if "TO expression2" is included, "expression3" is assumed to be 1.

5. If "TO expression2" is omitted from a "specification," iterative execution continues until it is terminated by the WHILE clause or by some statement within the group.

6. If both "TO expression2" and "BY expression3" are omitted from a specification it implies a single execution of the group, with the control "variable" having the value of "expression1." This is true even if "WHILE expression4" is included.

General rules:

1. In Type 1, the DO statement only delimits the start of a DO-group; it does not provide for iterative execution.

2. In Type 2, the DO statement delimits the start of a DO-group and provides for iterative execution as defined by the following:

```
LABEL: DO WHILE (expression);
       statement-1
       .
       .
       .
       statement-n
       END;
NEXT:  statement
       /*STATEMENT FOLLOWING THE DO
         GROUP*/
```

The above is exactly equivalent to the following expansion:

```
LABEL: IF (expression) THEN;
         ELSE GO TO NEXT;
       statement-1
       .
       .
       statement-n
       GO TO LABEL;
NEXT:  statement
       /*STATEMENT FOLLOWING
         THE DO GROUP*/
```

3. In Type 3, the DO statement delimits the start of a DO-group and provides for controlled iterative execution as defined by the following:

```
LABEL: DO variable=expression1
         TO expression2 BY expression3
         WHILE (expression4);
                 statement-1
                 .
                 .
                 .
                 statement-m
         LABEL1: END;
         NEXT:   statement
```

The above is exactly equivalent to the following expansion:

```
LABEL: e1=expression1;
       e2=expression2;
       e3=expression3;
       v=e1;
LABEL2:IF (e3>=0)&(v>e2)|(e3<0)&(v<e2)
          THEN GO TO NEXT;
       IF (expression4) THEN;
         ELSE GO TO NEXT;
       statement-1
       .
       .
       .
       statement-m
LABEL1:v=v+e3;
       GO TO LABEL2;
NEXT:  statement
```

In the above expansion e1, e2, and e3 are compiler-created work areas having the attributes of "expression1," "expression2," and "expression3," respectively; v is synonymous with "variable."

a. The above expansion only shows 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. Thus, if a second "specification" appeared in the DO statement the second expansion would look like this:

```
NEXT:     e5=expression5;
          .
          .
          .
          v=e5;
LABEL3:   IF... THEN GO TO NEXT1;
          IF (expression8) THEN;
            ELSE GO TO NEXT1;
          statement-1
          .
          .
          .
          statement-m
LABEL4:   v=v+e7;
          GO TO LABEL3;
NEXT1:    statement
```

Note that statements 1 through m are not actually duplicated in the program.

b. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in the expansion is omitted.

c. If "TO expression2" is omitted, the statement e2=expression2 and the IF statement identified by LABEL2 are omitted.

d. If both "TO expression2" and "BY expression3" are omitted, all statements involving e2 and e3, as well as the statement GO TO LABEL2 are omitted.

4. The WHILE clause in Types 2 and 3 specifies that before each iteration of statement execution, the associated element expression is evaluated, and, if necessary, converted to a bit string. If any bit in the resulting string is 1, the statements of the DO-group are executed. If all bits are 0, then, for Type 2, execution of the DO-group is terminated, while for Type 3, only the execution associated with the "specification" containing the WHILE clause is terminated; iterative execution for the next "specification," if one exists, then begins.

5. In a "specification," "expression1" represents the initial value of the control "variable"; "expression3" represents the increment to be added to the control variable after each execution of the statements in the group; "expression2" represents the terminating value of the control "variable." Execution of the statements in a DO group terminates for a "specification" as soon as the value of the control "variable" is outside the range defined by "expression1" and "expression2." When execution for the last "specification" is terminated, control passes to the statement following the DO-group.

6. Control may transfer into a DO-group from outside the DO-group only if the DO-group is delimited by the DO statement in Type 1; that is, only if iterative execution is not specified. Consequently, iterative DO-groups cannot contain ENTRY statements.

The END Statement

Function:

The END statement terminates blocks and groups.

General format:

```
END [statement-label-constant];
```

General rules:

1. The END statement always terminates that group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no corresponding END statement. Thus, if a statement label constant follows END, it must be the label of such a DO, BEGIN, or PROCEDURE statement. Note that if END corresponds to a DO or BEGIN statement to which more than one label has been attached, the label following END must be the label immediately preceding the keyword DO or BEGIN.

2. If control reaches an END statement for a procedure, it is treated as a RETURN statement.

The ENTRY Statement

Function:

The ENTRY statement specifies a secondary entry point of a procedure.

General format:

```
entry-name: ENTRY[(parameter
   [,parameter]...)] [attribute]...;
```

Syntax rules:

1. The only attributes that may be specified with an ENTRY statement are the arithmetic, string, PICTURE, and POINTER attributes. The attributes specified determine the characteristics of the value returned by the procedure when it is invoked as a function at this entry point.

2. A condition prefix cannot be specified for an ENTRY statement.

3. The ENTRY statement must have one and only one entry name appended to it.

4. No more than 12 parameters can appear in an ENTRY statement (the sum total of all of the parameters in any one procedure cannot exceed 12).

General rules:

1. The relationship established between the parameters of a secondary entry point and the arguments passed to that entry point is similar to that established for primary entry point parameters and arguments. See Part I, Chapter 10, "Subroutines and Functions" for a complete discussion of this subject.

2. As stated in syntax rule 1, the attributes specified with an ENTRY statement determine the characteristics of the value returned by the procedure when it is invoked as a function at this entry point. The value being returned by the procedure (i.e., the value of the expression in a RETURN statement) is converted, if necessary, to correspond to the specified attributes. If the attributes are not specified at the entry point, default attributes are applied, according to the first letter of the entry name used to invoke the entry point. The data attributes of a secondary entry point (default or otherwise) must be exactly the same as those of the primary entry point if the procedure is used as a function procedure.

3. The ENTRY statement must be internal to the procedure for which it defines a secondary entry point. It may not be internal to any block contained in this procedure; nor may it be within a DO group that specifies iterative execution.

4. The parameters of a secondary entry point must be explicitly declared elsewhere in the block (i.e., either in the parameter list of the PROCEDURE statement or in a DECLARE statement, or both).

5. For the D-Compiler, the maximum length of an external name is six. Therefore, the entry name of an ENTRY statement internal to an external procedure cannot be longer than six.

The FORMAT Statement

Function:

The FORMAT statement specifies a format list that can be used by edit-directed

transmission statements to control the format of the data being transmitted.

General format:

```
label: [label:]... FORMAT (format-list);
```

Syntax rules:

1. The "format list" must be specified according to the rules governing format list specifications with edit-directed transmission as described in Part I, Chapter 8, "Input and Output."

2. At least one "label" must be specified for a FORMAT statement. In general, one of the labels (or a label variable having the value of one of the labels) is the statement label designator specified in a remote format item.

General rules:

1. A GET or PUT statement may include a remote format item, R, in the format list of an edit-directed data specification. That portion of the format list represented by R must be supplied by a FORMAT statement identified by the statement label specified with R. An R format item cannot appear in the format list of a FORMAT statement.

2. The remote format item and the FORMAT statement must be internal to the same block.

3. If a condition prefix is associated with a FORMAT statement, it must be identical to the condition prefix associated with the GET or PUT statement referring to that FORMAT statement.

The GET Statement

Function:

The GET statement is a STREAM transmission statement that can be used in either of the following ways:

1. It can cause the assignment of data from an external source (that is, from a data set) to one or more internal receiving fields (that is, to one or more variables).

2. It can cause the assignment of data from an internal source (that is, from a character-string variable) to one or more internal receiving fields (that is, to one or more variables).

General format:

```
GET [FILE(file-name)                          ]
    [STRING(character-string-variable)        ]
        data-specification;
```

Syntax rules:

1. The "data specification" is as described in Part I, Chapter 8, "Input and Output."

2. The "data specification" must follow the FILE or STRING option, if either option is specified.

3. The "character string variable" refers to the character string that is to provide the values to be assigned to the variables in the "data specification."

4. The "file name" is the name of a file that has been associated (by an implicit or explicit opening) with the data set that will provide the values to be assigned to the variables in the "data specification." It must have the STREAM and INPUT attributes.

5. If neither the FILE nor the STRING option appears, the standard system input file is assumed.

General rules:

1. If the FILE option refers to an unopened file, the file is opened implicitly.

2. If the STRING option has been specified, the internal GET operation always begins at the beginning of the specified string. If the number of characters in this string is less than the total number of characters required by the variables in the "data specification," the ERROR condition is raised. Note that the variables in the "data specification" do not have to be character strings; the internal assignment is the same as the transmission from the stream to internal storage, the only difference being that the "character string variable" is considered to be the input stream.

The GO TO Statement

Function:

The GO TO statement causes control to be transferred to the statement identified by the specified label.

General format:

```
{GO TO}{label-constant;                }
{GOTO }{element-label-variable;        }
```

General rules:

1. If an "element label variable" is specified, the value of the label variable determines the statement to which control is transferred. Since the label variable may have different values at each execution of the GO TO statement, control may not always pass to the same statement.

2. A GO TO statement cannot pass control to an inactive block.

3. A GO TO statement cannot transfer control from outside a DO group to a statement inside the DO group if the DO-group specifies iterative execution, unless the GO TO terminates a procedure invoked from within the DO-group or unless the GO TO is an on-unit given control from within the DO-group.

4. If a GO TO statement transfers control from within a block to a point not contained within that block, the block is terminated. Also, 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 Part I, Chapter 6, "Blocks, Flow of Control, and Storage Allocation" for examples and details). When one or more blocks are terminated by a GO TO statement, conditions are reinstated and automatic variables are freed just as if the blocks had terminated in the usual fashion.

5. When a GO TO statement transfers control out of a procedure that has been invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued.

6. If the GO TO statement is an on-unit, the specified label must be unsubscripted.

The IF Statement

Function:

The IF statement tests the value of a specified expression and controls the flow of execution according to the result of that test.

General format:

```
IF element-expression THEN unit-1
                          [ELSE unit-2]
```

Syntax rules:

1. Each unit is either a single statement (except DO, END, PROCEDURE, BEGIN, DECLARE, FORMAT, or ENTRY), a DO-group, or a begin block.

2. The IF statement itself is not terminated by a semicolon; however, each "unit" specified must be terminated by a semicolon.

3. Each "unit" may be labeled and may have a condition prefix.

General rules:

1. The element expression is evaluated and, if necessary, converted to a bit string. When the ELSE clause (that is, ELSE and its following "unit") is specified, the following occurs:

   If any bit in the string is 1, "unit-1" is executed, and control then passes to the statement following the IF statement. If all bits in the string have the value 0, "unit-1" is skipped and "unit-2" is executed, after which control passes to the next statement.

   When the ELSE clause is not specified, the following occurs:

   If any bit in the string is 1, "unit-1" is executed, and control then passes to the statement following the IF statement. If all bits are 0, "unit-1" is not executed and control passes to the next statement.

   Each "unit" may contain statements that specify a transfer of control (e.g., GO TO); hence, the normal sequence of the IF statement may be overridden.

2. IF statements may be nested; that is, either "unit," or both, may itself be an IF statement. Since each ELSE

clause is always associated with the innermost unmatched IF in the same block or DO-group, an ELSE with a null statement may be required to specify a desired sequence of control.

3. A condition prefix to the IF statement itself applies to "element expression" only.

The LOCATE Statement

Function:

The LOCATE statement is a RECORD transmission statement that can be used only for output files having the BUFFERED attribute. It allocates storage for a based variable in an output buffer to allow the creation of a record for that based variable. The record is created by assigning values to the based variable within the buffer. The record is not transmitted to the external medium until immediately before the next WRITE, LOCATE, or CLOSE statement (or implicit close operation) is executed for the specified file.

General format:

```
LOCATE based-variable FILE(file-name)
   SET(pointer-variable);
```

Syntax rules:

1. The FILE and SET specifications must appear in the order shown in the general format.

2. The based variable must be an unsubscripted based variable that is not a minor structure or an element of a structure.

3. The pointer variable must be a subscripted or unsubscripted element pointer variable.

4. The file name is the name of a file that has been associated (by opening) with the data set that will eventually receive the record. The file must have the SEQUENTIAL, OUTPUT, and BUFFERED attributes.

General rules:

1. The based variable is used, for variable-length records, to determine the length of the record. When the LOCATE statement is executed, the pointer variable in the SET specification is set to identify the location in the buffer at which the based variable is to be allocated.

2. The record identified by the based variable is written out of the buffer, and into the output file, immediately before the next WRITE, LOCATE, or CLOSE operation (implicit or explicit) for that file. For blocked records, the record is not written until the whole block is completed. Note that the length of the record identified by the based variable must be evenly divisible by 8, for fixed-length records, and must yield a remainder of 4 after division by 8, for variable-length records.

3. The FILE specification must refer to a previously opened file.

The Null Statement

Function:

The null statement causes no action and does not modify sequential statement execution.

General format:

```
[label:]...;
```

The ON Statement

Function:

The ON statement specifies what action is to be taken (programmer-defined or standard system action) when an interrupt results from the occurrence of the specified exceptional condition.

General format:

```
ON condition{SYSTEM;|on-unit}
```

Syntax rules:

1. The condition may be any of those described in Section H "ON-Conditions."

2. The "on-unit" represents a programmer-defined action to be taken when an interrupt results from the occurrence of the specified "condition." It can be either a single unlabeled GO TO or null statement.

3. Since the "on-unit" itself requires a semicolon, no semicolon is shown for the "on-unit" in the general format.

However, the word SYSTEM must be followed by a semicolon.

General rules:

1. The ON statement determines how to handle an interrupt that has occurred for the specified condition. Whether the interrupt is handled in a standard system fashion or by a programmer-supplied method is determined by the action specification in the ON statement, as follows:

   a. If the action specification is SYSTEM, the standard system action is taken. The standard system action is not the same for every condition, although for most conditions the system simply prints a message and raises the ERROR condition. Section H, "ON-Conditions" gives the standard system action for each condition. (Note that the standard system action is always taken if an interrupt occurs and no ON statement for the condition is in effect.)

   b. If the action specification is an "on-unit," the programmer has supplied his own interrupt-handling action, namely, the action defined by the statement in the on-unit itself. The on-unit is not executed when the ON statement is executed; it is executed only when an interrupt results from the occurrence of the specified condition (or if the interrupt results from the condition being raised by a SIGNAL statement).

2. The action specification (i.e., "on-unit" or SYSTEM) established by executing an ON statement in a given block remains in effect throughout that block and throughout all blocks in any activation sequence initiated by that block, unless it is overridden by the execution of another ON statement or a REVERT statement, as follows:

   a. If a later ON statement specifies the same condition as a prior ON statement and this later ON statement is executed in a block that lies within the activation sequence initiated by 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.

   b. 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 completely nullified.

3. The label of a GO TO statement on-unit must be known within the block in which the ON statement for that on-unit is executed. (Remember that an ON statement is executed as it is encountered in statement flow; whereas, the action specification for that ON statement is executed only when the associated interrupt occurs.)

4. The file name of an input/output condition must be known within the procedure or begin block to which the ON statement specifying the condition is internal.

5. A condition raised during execution results in an interrupt if and only if the condition is enabled at the point where it is raised.

   a. The SIZE condition is disabled by default. All other conditions are enabled by default.

   b. The enabling and disabling of OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, and SIZE, can be controlled by condition prefixes.

The OPEN Statement

Function:

The OPEN statement opens a file by associating a filename with a data set. It also can complete the specification of attributes for the file, if a complete set of attributes has not been declared for the file being opened.

General format:

```
OPEN FILE(file-name) options-group
 [,FILE(file-name) options-group]...;
```

where "options-group" is as follows:

```
[INPUT|OUTPUT]
[PAGESIZE(element-expression)]
```

Syntax rules:

1. The INPUT or OUTPUT option can be specified in an OPEN statement only for an UNBUFFERED file. If it is not specified in the OPEN statement, then the corresponding INPUT or OUTPUT attribute must have been specified in the DECLARE statement for the file. INPUT or OUTPUT cannot be specified in both the OPEN and DECLARE statements.

2. The FILE specification must appear first.

3. The "file name" is the name of the file that is to be associated with a data set. Several files can be opened by one OPEN statement.

General rules:

1. The opening of an already open file does not affect the file. In such cases, any expressions in the "options group" are evaluated, but they are not used.

2. The PAGESIZE option can be specified only for a file having the STREAM and PRINT attributes. The element expression is evaluated and converted to an integer, which represents the maximum number of lines to a page. This integer must be greater than zero and less than 256. During subsequent transmission to the PRINT file, a new page may be started by use of the PAGE format item or by an option in the PUT statement. For the D-Compiler, if PAGESIZE is not specified, the default is defined by the installation-specified system limit.

3. When a PRINT file is opened, a new page is started.

The PROCEDURE Statement

Function:

The PROCEDURE statement has the following functions:

- It heads a procedure.
- It defines the primary entry point to the procedure.
- It specifies the parameters, if any, for the primary entry point.
- It may specify certain special characteristics that a procedure can have.
- It specifies the attributes of the value that is returned by the procedure when it is invoked as a function at its primary entry point.

General format:

```
entry-name:  PROCEDURE[(parameter
             [,parameter]...)]

             [OPTIONS(option-list)]

             [data-attributes];
```

where, for the D-Compiler, "option list" is defined as:

```
MAIN[,ONSYSLOG]
```

Syntax rules:

1. The "data attributes" represent the attributes of the value returned by the procedure when it is invoked as a function at its primary entry point. Only arithmetic, string, PICTURE, and POINTER attributes are allowed.

2. OPTIONS is a special procedure specification. It and the "data attributes" may appear in any order and are separated by blanks. OPTIONS can and must be specified for only one external procedure in the program.

3. One and only one entry name must appear on a PROCEDURE statement.

4. The sum total of different parameters that can be specified for one procedure (including any specified in ENTRY statements) cannot exceed 12.

General rules:

1. When the procedure is invoked, a relationship is established between the arguments passed to the procedure and the parameters that represent those arguments in the invoked procedure. This topic is discussed in Part I, Chapter 10, "Subroutines and Functions."

2. The OPTIONS specification can be used only for an external procedure. The MAIN option specifies that this procedure is the initial procedure and will be invoked by the operating system as the first step in the execution of the program. The ONSYSLOG option specifies that all output resulting from actions derived from ON conditions will go on the system log. No other options are permitted. If both are specified, MAIN must appear first. The procedure declared with the OPTIONS attribute remains active for

the duration of the program and hence cannot be called by other procedures. For the D-Compiler, one and only one external procedure must have the OPTIONS (MAIN) designation.

3. The "data attributes" specify the attributes of the value returned by the procedure when it is invoked as a function at its primary entry point. The value specified in the RETURN statement of the invoked procedure is converted to conform with these attributes before it is returned to the invoking procedure.

   If "data attributes" are not specified, default attributes are supplied. In such a case, the name of the entry point (the entry name by which the procedure has been invoked) is used to determine the default base, scale, and precision.

4. The entry name of an external procedure is an external name and as such is restricted by the D-Compiler to a maximum length of six.

The PUT Statement

Function:

The PUT statement is a STREAM transmission statement that can be used in either of the following ways:

1. It can cause the values in one or more internal storage locations to be transmitted to a data set on an external medium. Related to this, it can control the format of a PRINT file.

2. It can cause the values in one or more internal storage locations to be assigned to an internal receiving field (represented by a character-string variable).

General format:

```
PUT [FILE (file-name)                          ]
    [STRING (character-string-variable)        ]

    [PAGE[LINE(element-expression)]            ]
    [SKIP[(element-expression)]                ]
    [LINE(element-expression)                  ]

    [data-specification];
```

Syntax rules:

1. If neither the FILE nor STRING option appears, the standard system output file is assumed.

2. The FILE option specifies transmission to a data set on an external medium. The file name in this option is the name of the file that has been associated (by implicit or explicit opening) with the data set that is to receive the values. This file must have the OUTPUT and STREAM attributes.

3. The STRING option specifies transmission from internal storage locations (represented by variables or expressions in the "data specification") to a character string (represented by the "character string variable"). The "character string variable" cannot be a pseudo-variable.

4. The "data-specification" option is as described in Part I, Chapter 8, "Input and Output."

5. If the FILE or STRING option appears, it must be the first option. If the data-specification appears, it must be the last option. A minimum of either the PAGE, LINE, SKIP, or "data specification" must appear.

General rules:

1. If the FILE option is specified, and the "file name" refers to an unopened file, the file is opened implicitly.

2. If the STRING option is specified, the PUT operation begins assigning values to the beginning of the string (that is, at the left most character position), after appropriate conversions have been performed. Blanks and delimiters are inserted as usual. If the string is not long enough to accomodate the data, the ERROR condition is raised. Note that the variables in the "data specification" do not have to be character strings; the internal assignment is the same as the transmission from internal storage to the stream, the only difference being that the "character-string variable" is considered to be the output stream.

3. The options PAGE, SKIP, and LINE can be given only for PRINT files. If specified, they take effect before the transmission of the values defined by the "data specification" takes place. If PAGE and LINE are specified in the same PUT statement, PAGE takes effect before LINE.

4. The PAGE option causes a new current page to be defined within the data set. If a "data specification" is present, the transmission of values occurs after the definition of the new

page. A new current page implies line 1.

5. The SKIP option causes a new current line to be defined for the data set. The "element expression," if present, is converted to an integer, w which must be greater than or equal to 0 and less than or equal to 3. If w is greater than zero, w-1 blank lines are created, and the new current line is w plus the line value for the old current line. If w is equal to zero, the effect is that of a carriage return, with the current line remaining constant; characters previously written will be overprinted. If "element expression" is not present, w is assumed to be 1. If less than w lines remain on the current page (where the number of lines on the current page is determined by the PAGESIZE option of the OPEN statement or by default), the ENDPAGE condition is raised.

6. The LINE option causes a new current line to be defined for the data set. The "element expression" is converted to an integer w. The new current line is set equal to w, and blank lines are inserted between the old current line and the new current line. However, if w is less than or equal to the old current line, or if w exceeds the number of lines on the current page (see the PAGESIZE option description in the OPEN statement), the ENDPAGE condition is raised. If w is less than or equal to zero, it is assumed to be 1.

The READ Statement

Function:

The READ statement is a RECORD transmission statement that transmits a record from an INPUT or UPDATE file to a variable in internal storage.

General format:

```
READ FILE(file-name)
     {INTO (variable)          }
     {SET (pointer-variable)   }
      [KEY (element-expression)];
```

Syntax rules:

1. The FILE specification must appear first. INTO or SET must be specified.

2. The "file name" is the name of the file from which the record is to be read. This file must have the RECORD attribute and must also have either the INPUT or UPDATE attributes.

3. The variable of the INTO option is the variable into which the record is to be read. It must be an unsubscripted variable not contained in a structure. It cannot be a label variable or a parameter and it cannot have the DEFINED attribute.

General rules:

1. The file appearing in the FILE specification must have been opened previously.

2. The KEY option must appear if the file has the DIRECT attribute. The "element expression" is the key that determines which record will be read. (See Part I, Chapter 8, "Input and Output" for a discussion of keys.) The KEY option cannot appear for a SEQUENTIAL file.

3. The SET option cannot be specified for files having the UNBUFFERED or DIRECT attributes. This option specifies that the record is to be read into a buffer and the "pointer variable" is to be set to point to the location of that record within the buffer. The description of the record is determined by a based variable associated with that pointer variable. The value of the pointer variable is valid until the next READ statement is executed or until the file is closed.

The RETURN Statement

Function:

The RETURN statement terminates execution of the procedure to which the RETURN statement is internal, and returns control to the invoking procedure. It may also return a value to the invoking procedure.

General format:

```
RETURN [(element-expression)];
```

General rules:

1. If the "element expression" is not specified, the RETURN statement can only terminate a procedure that has not been invoked as a function. When such a statement is executed, control

is returned to the invoking procedure at the point logically following the point of invocation. If a RETURN statement is executed in the initial procedure, program execution is terminated.

2. If the "element expression" is specified, the procedure terminated by this statement must be a function procedure. When such a statement is executed, control is returned to the invoking procedure at the point of invocation; the value returned to this point is the value of the "element expression." If this value does not conform to the explicit or default attributes specified for the procedure being terminated, the value is converted to these attributes before it is actually returned.

3. The RETURN statement cannot appear within a begin block.

The REVERT Statement

Function:

The REVERT statement nullifies the effect of the current action specification for the specified condition only if the current action specification is the result of an ON statement executed within the same invocation of the block in which the REVERT statement is executed. When this is true, the action specification that was in effect for the specified condition when the block containing the REVERT statement was invoked is re-established and once again takes effect.

General format:

```
REVERT condition;
```

Syntax rule:

The "condition" is any of those described in Section H, "ON-Conditions."

General rule:

The execution of a REVERT statement has the effect described above only if (1) an ON statement, specifying the same condition and internal to the same block, was executed after the block was activated and (2) the execution of no other similar REVERT statement has intervened. If either of these two conditions is not met, the REVERT statement is treated as a null statement.

The REWRITE Statement

Function:

The REWRITE statement can be used only for update files. It replaces an existing record in a data set.

General format:

```
REWRITE FILE (file-name) [FROM(variable)
     [KEY (element-expression)]];
```

Syntax rules:

1. The FILE specification must appear first. KEY cannot be specified without FROM.

2. The "file name" is the name of the file containing the record to be rewritten. The file must have the UPDATE attribute.

3. The "variable" in the FROM option represents the record that will replace the existing record in the specified file. It must be an unsubscripted variable; it cannot be contained in a structure; it cannot be a parameter; and it cannot have the DEFINED attribute.

General rules:

1. The file whose name appears in the FILE specification must have been opened previously.

2. The KEY option must appear if the file has the DIRECT attribute; it cannot appear otherwise. The element-expression is converted to a character string. This character string is the source key that determines which record is to be rewritten.

3. The FROM option must be specified for UPDATE files having either the DIRECT attribute or both the SEQUENTIAL and UNBUFFERED attributes.

4. The FROM option can be omitted only for update files having the SEQUENTIAL and BUFFERED attributes. When this is the case, the record rewritten is the record in the buffer. Hence, this record must be the last record that was read and it should have been read by a READ statement with a SET option. (The record will be updated by whatever assignments were made to it in the buffer.) If it was read by a READ with an INTO option, the record would be rewritten unchanged.

The SIGNAL Statement

Function:

The SIGNAL statement simulates the occurrence of an interrupt. It may be used to test the current action specification for the associated condition.

General format:

```
SIGNAL condition;
```

Syntax rule:

The "condition" is any one of those described in Section H, "ON-Conditions."

General rules:

1. When a SIGNAL statement is executed, it is as if the specified condition has actually occurred. Sequential execution is interrupted and control is transferred to the current on-unit for the specified condition. If the on-unit is a null statement, control normally returns to the statement immediately following the SIGNAL statement.

2. If the specified condition is disabled, no interrupt occurs, and the SIGNAL statement becomes equivalent to a null statement.

3. If there is no current on-unit for the specified condition, then the standard system action for the condition is performed.

The STOP Statement

Function:

The STOP statement causes immediate termination of the program in which it is executed.

General format:

```
STOP;
```

The WRITE Statement

Function:

The WRITE statement is a RECORD transmission statement that transfers a record from a variable in internal storage to an OUTPUT or UPDATE file.

General format:

```
WRITE FILE (file-name) FROM (variable)
   [KEYFROM(element-expression)];
```

Syntax rules:

1. The FILE specification must appear first.

2. The "file name" specifies the file in which the record is to be written. This file must be a RECORD file that has either the OUTPUT attribute or the DIRECT and UPDATE attributes.

3. The "variable" in the FROM specification contains the record to be written. It must be an unsubscripted variable; it cannot be contained in a structure; it cannot be a parameter; and it cannot have the DEFINED attribute.

4. The KEYFROM option must be specified for DIRECT files; it cannot be specified otherwise.

General rules:

1. The file must have been opened previously.

2. If the KEYFROM option is specified, the "element expression" is the source key that specifies the relative location in the data set where the record when it is written. (See Part I, Chapter 8, "Input and Output" for a discussion of source keys.)

This section provides definitions for most of the terms used in this publication.

access: the act that encompasses the reference to and retrieval of data.

action specification: in an ON statement, the on-unit or single keyword SYSTEM, either of which specifies the action to be taken whenever an interrupt results from the raising of the named condition.

activation: institution of execution of a block. A procedure block is activated when it is invoked at any of its entry points; a begin block is activated when it is encountered in normal sequential flow.

active: the state in which a block is said to be after activation and before termination.

additive attributes: file attributes for which there are no defaults and which, if required, must always be stated explicitly.

address: a specific storage location at which a data item can be stored.

allocated variable: a variable with which storage has been associated.

allocation: the association of storage with a variable.

alphabetic character: any of the characters A through Z and the alphabetic extenders #, $, and @.

alphameric character: an alphabetic character or a digit.

alternative attributes: file attributes that may be chosen from groups of two or more alternatives. If none is specified, a default is assumed.

argument: an expression, file name, statement label constant or variable, or entry name passed to an invoked procedure as part of the procedure reference. It cannot be a built-in function name or a based variable.

arithmetic conversion: the transformation of a value from one arithmetic representation to another arithmetic representation.

arithmetic data: data that has the characteristics of base, scale, and precision. It includes coded arithmetic data and numeric character data.

arithmetic operators: any of the prefix operators, + and -, or the infix operators, +, -, *, /, and **.

array: a named, ordered collection of data elements, all of which have identical attributes. An array has dimensions, and elements that are identified by subscripts.

assignment: giving a value to a variable.

attribute: a descriptive property associated with a name or expression to describe a characteristic of a data item or a file that the name may represent.

automatic storage: storage that is allocated at the activation of a block and released at the termination of that block.

base: the number system in terms of which an arithmetic value is represented. In PL/I, the base is binary or decimal.

based variable: a variable declared to have the BASED (pointer-variable) attribute specification. The pointer variable associates the description with an allocation of storage.

begin block: a collection of statements headed by a BEGIN statement and ended by an END statement that delimits the scope of names and is activated by normal sequential statement flow. It controls the allocation and freeing of automatic storage declared in that block.

binary: the number system based on the value 2.

bit: a binary digit, either 0 or 1

bit string: a string of one or more bits.

bit-string operators: any of the operators ¬(not), &(and), and |(or).

block: a begin block or a procedure block.

bound: the upper limit of an array dimension. The lower limit is always assumed to be 1.

buffer: an intermediate area, used in input/output operations, into which a record is read during input and from which a record is written during output.

built-in function: one of the PL/I-defined functions.

call: the invocation of a subroutine by means of the CALL statement.

character string: A string composed of one or more characters from the data character set.

coded arithmetic data: arithmetic data whose characteristics are given by the base, scale, and precision attributes. The types for System/360 are packed decimal, binary full words, and hexadecimal floating-point.

comment: a string of characters, used for documentation, which is preceded by /* and terminated by */ and which is treated as a blank.

comparison operators: the operators ¬< < <= ¬= = >= > ¬>

compile time: the time during which a source program is translated into an object module.

compiler: a translator that converts a source program into an object module.

compound statement: a statement that contains other statements. IF and ON are the only compound statements.

concatenation: the operation that connects two strings in the order indicated thus forming one string whose length is equal to the sum of the lengths of the two strings. It is specified by the operator ||.

condition name: a language keyword that represents an exceptional condition that might arise during execution of a program.

condition prefix: a parenthesized list of one or more condition names prefixed to a statement by a colon. It determines whether or not the program is to be interrupted if one of the specified conditions occurs within the scope of the prefix. Condition names within the list are separated by commas.

constant: an arithmetic or string data item that does not have a name; a statement label.

contained in: all of the text of a block except the entry names of that block. (A label of a BEGIN statement is not contained in the begin block defined by that statement.)

contextual declaration: the association of attributes with an identifier according to the context in which the identifier appears. Only entry names can be contextually declared.

conversion: the transformation of a value from one representation to another.

data: representation of information or of value.

data character set: all of those characters whose bit configuration is recognized by the computer in use.

data item: a single unit of data; it is synonymous with "element."

data list: a list of expressions used in a STREAM input/output specification that represent storage areas to which data items are to be assigned during input, and from which data items are to be written, during output. (On input, the list may contain only variables.)

data set: a collection of data external to the program.

data specification: the portion of an edit-directed data transmission statement that specifies the mode of transmission (EDIT) and includes the data list and the format list.

decimal: the number system based on the value 10.

declaration: the association of attributes with an identifier explicitly, contextually, or implicitly.

default: the alternative assumed when an identifier has not been declared to have one of two or more alternative attributes.

delimiter: any valid special character or combination of special characters used to separate identifiers and constants, or statements from one another.

dimensionality: the number of bound specifications associated with an array. It cannot be greater than three.

disabled: the state in which the occurrence of a particular condition will not result in a program interrupt.

DO-group: a sequence of statements headed by a DO statement and closed by its corresponding END statement.

dummy argument: a compiler-assigned variable for an argument that has no programmer-assigned name.

edit-directed transmission: STREAM transmission; both a data list and a format list are specified.

element: a single data item as opposed to a collection of data items, such as a

structure or an array. (Sometimes called a "scalar item.")

element variable: a variable that can represent only a single value at any one point in time.

enabled: that state in which the occurrence of a particular condition will result in a program interrupt.

entry name: a label of a PROCEDURE or ENTRY statement.

entry point: a point in a procedure at which it may be invoked by reference to the entry name. (See primary entry point and secondary entry point.)

epilogue: those processes which occur at the termination of a block.

exceptional condition: an occurrence, which can cause a program interrupt, of an unexpected situation, such as an overflow error, or an occurrence of an expected situation, such as an end of file, that occurs at an unpredictable time.

explicit declaration: the assignment of attributes to an identifier by means of the DECLARE statement, the appearance of the identifier as a label, or the appearance of the identifier in a parameter list.

exponent (of floating-point constant): a decimal integer constant specifying the power to which the base of the floating-point number is to be raised.

expression: the representation of a value; examples are variables and constants appearing alone or in combination with operators, and function references. The term "expression" refers to an element expression, an array expression, or a structure expression.

external declaration: an explicit or contextual declaration of the EXTERNAL attribute for an identifier. Such an identifier is known in all other blocks for which such a declaration exists.

external name: an identifier which has the EXTERNAL attribute.

external procedure: a procedure that is not contained in any other procedure.

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): a character-string picture specification or a portion (or all) of a numeric character picture specification. If more than one field appears in a single specification, they are divided by the K or E exponent character for floating-point data or the M field-separator for sterling data. Only one field can appear in a fixed-point specification.

file: a symbolic representation, within a program, of a data set.

file name: a symbolic name used within a program to refer to a data set.

format item: a specification used in edit-directed transmission to describe the representation of a data item in the stream or to control the format of a printed page.

format list: a list of format items required for an edit-directed data specification.

function: a procedure that is invoked by the appearance of one of its entry names in a function reference.

function reference: the appearance of an entry name in an expression, usually in conjunction with an argument list.

group: a DO group.

identifier: a string of alphameric and break characters, not contained in a comment or constant, preceded and followed by a delimiter and whose initial character is alphabetic.

implicit declaration: association of attributes with an identifier used as a variable without having been explicitly or contextually declared; default attributes apply, depending upon the initial letter of the identifier.

inactive block: a procedure or begin block that has not been activated or that has been terminated.

infix operator: an operator that defines an operation between two operands.

initial procedure: an external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. Every PL/I program must have an initial procedure. It is invoked automatically as the first step in the execution of a program.

input/output: the transfer of data between an external medium and internal storage.

internal block: a block that is contained within another block.

internal name: an identifier that has the INTERNAL attribute.

internal procedure: a procedure that is contained in another block.

internal to: all of the text contained in a block except that text contained in another block. Thus the text of an internal block (except for its entry names) is not internal to the containing block. Note: An entry name of a block is not contained in that block.

interrupt: the suspension of normal program activities as the result of the occurrence of an enabled condition.

invoke: to activate a procedure at one of its entry points.

invoked procedure: a procedure that has been activated at one of its entry points.

invoking block: a block containing a statement that activates another block.

iteration factor: an expression that specifies the number of times a given format item or list of format items is to be used in succession in a format list.

key: see source key and recorded key.

keyword: an identifier that is part of the language and which, when used in the proper context, has a specific meaning to the compiler.

known: a term that is used to indicate the scope of an identifier. For example, an identifier is always known in the block in which it has been declared.

label constant: synonymous with statement label.

label prefix: an unparenthesized identifier prefixed to a statement by a colon.

leading zeros: zeros that have no significance in the value of an arithmetic number; all zeros to the left of the first significant digit (1 through 9) of a number.

level number: an unsigned decimal integer constant specifying the hierachy of a name in a structure. It appears to the left of the name and is separated from it by a blank.

major structure: a structure whose name is declared with level number 1.

minor structure: a structure whose name is declared with a level number greater than 1.

multiple declaration: two or more declarations of the same identifier internal to the same block without different qualifications, or two or more EXTERNAL declarations of the same identifier as different names within a single program.

name: an identifier that has been declared.

nesting:

1. the occurrence of a block within another block.
2. the occurrence of a group within another group.
3. the occurrence of an IF statement in a THEN clause or an ELSE clause.
4. the occurrence of a function reference as an argument of another function reference.

numeric character data: arithmetic data described by a picture that is stored in character form. It has both an arithmetic value and a character-string value. The picture must not contain an X picture specification character.

on-unit: the action to be executed upon the occurrence of the ON-condition named in the containing ON statement.

operator: a symbol specifying an operation to be performed. See arithmetic operators, bit-string operators, comparison operators, and concatenation.

option: a specification in a statement that may be used by the programmer to influence the execution of the statement.

packed decimal: the System/360 internal representation of a fixed-point decimal data item.

parameter: a name in an invoked procedure that is used to represent an argument passed to that procedure.

picture: a character-by-character specification describing the composition and attributes of numeric character and character-string data. It allows editing.

point of invocation: the point in the invoking block at which the procedure reference to the invoked procedure appears.

pointer variable: a variable that identifies the storage to be used when referring to a based variable.

precision: the value range of an arithmetic variable expressed as the total number of digits allowed and, for fixed-point variables, the assumed location of the decimal (or binary) point.

prefix: a label or a parenthesized list of condition names connected by a colon to the beginning of a statement.

prefix operator: an operator that precedes, and is associated with, a single operand. The prefix operators are ¬ + -.

primary entry point: the entry point named in the PROCEDURE statement.

problem data: string or arithmetic data that is processed by a PL/I program.

procedure: a block of statements, headed by a PROCEDURE statement and ended by an END statement, that defines a program region and delimits the scope of names and that is activated by a reference to its name. It controls allocation and freeing of automatic storage declared in that block.

procedure reference: a function or subroutine reference.

program: a set of one or more external procedures, one of which must have the OPTIONS(MAIN) attribute in its PROCEDURE statement.

program control data: data used in a PL/I program to affect the execution of the program. Label data and pointer data are the types of program control data.

prologue: those processes that occur at the activation of a block.

pseudo-variable: one of the built-in function names that can be used as a receiving field. Only SUBSTR and UNSPEC can be so used.

qualified name: a sequence of names of structure members connected by periods, to uniquely identify a component of a structure.

receiving field: any field to which a value may be assigned.

record: the unit of transmission in a RECORD input or output operation.

recorded key: a character string recorded in a direct-access volume to identify the data record that immediately follows.

repetition factor: a parenthesized unsigned decimal integer constant preceding a string configuration as a shorthand representation of a string constant. The repetition factor specifies the number of occurrences that make up the actual constant. In picture specifications, the repetition factor specifies repetition of a single picture character.

repetitive specification: an element of a data list that specifies controlled iteration to transmit a list of data items, generally used in conjunction with arrays.

returned value: the value returned by a function procedure to the point of invocation.

scale: fixed- or floating-point representation of an arithmetic value.

scope (of a condition prefix): the range of a program throughout which a condition prefix applies.

scope (of a name): the range of a program throughout which a name has a particular interpretation.

secondary entry point: an entry point defined by a label of an ENTRY statement within a procedure.

source key: a character string or a numeric character data item referred to in a RECORD transmission statement that identifies a particular record within a direct-access data set. The source key is a string to be compared with, or written as, a recorded key to positively identify the record.

source program: the program that serves as input to the compiler.

standard file: a file assumed by the compiler in the absence of a FILE or STRING option in a GET or PUT statement.

statement: a basic element of a PL/I program that is used to delimit a portion of a program, to describe data used in the program, or to specify action to be taken. edure

statement label: an identifying name prefixed to any statement other than a PROCEDURE or ENTRY statement.

statement label variable: a variable declared with the LABEL attribute and thus able to assume as its value a statement label.

static storage: storage that is allocated before execution of the program begins and that remains allocated for the duration of the program.

stream: data being transferred from or to an external medium represented as a continuous string of data items in character form.

string: a connected sequence of characters or bits that is treated as a single data item.

structure: a hierarchical set of names that refers to an aggregate of data items that may have different attributes.

subfield: the integer description portion or the fraction description portion of a picture specification field that describes a noninteger fixed-point data item. The subfields are divided by the picture character V.

subroutine: a procedure that is invoked by a CALL statement. A subroutine cannot return a value to the invoking block, but it can alter the value of variables that are known within the invoking block.

subscript: an element expression specifying a location within a dimension of an array.

termination: 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 some other active block by means of a GO TO statement. A return of control to the operating system via a RETURN or END statement in the initial procedure or a STOP statement in any block results in the termination of the program. See epilogue.

variable: a name that represents data. Its attributes remain constant, but it can represent different values at different times. Variables fall into three categories: element, array, and structure variables. Variables may be subscripted and/or qualified.

The DOS/TOS PL/I D-Compiler is upwardly compatible with the PL/I F-Compiler, which operates under the IBM System/360 Operating System. In general, a PL/I source program written for the D-Compiler produces the same results when compiled and executed under the F-Compiler. However, since the compilers are still evolving, some upward incompatibilities exist between the version of the D-Compiler described in this publication and the version of the F-Compiler that is described in the publication IBM System/360 PL/I Reference Manual, Form C28-8201. These upward incompatibilities are discussed in the list below.

1. Pointers, based variables, and the STRING, ADDR, and NULL built-in functions are not implemented by the F-Compiler.

2. Some error conditions defined by the PL/I language are not checked by the D-Compiler but they are checked by the F-Compiler. For example, the D-Compiler does not check for transfers into an iterative DO-group; hence, the programmer will get unpredictable results at object-time. However, the F-Compiler does check for this error condition and will provide a diagnostic should it arise.

3. If a SIZE error occurs during output controlled by an F or E format item, the value that caused the error is transmitted as a field of asterisks by the D-Compiler, whereas the F-Compiler transmits the truncated value. (This is so whether or not SIZE is enabled.)

4. If the magnitude of a value transmitted as output under control of the F format item is less than one, or if the mantissa of a value transmitted under E format is zero, the F-Compiler places a leading zero before the decimal point; the D-Compiler does not. For example, a value transmitted by the D-Compiler as -.500, would be transmitted by the F-Compiler as -0.500.

5. Under the D-Compiler, the first PUT statement referring to a PRINT file results in a new page; under the F-Compiler, it does not. Therefore, for consistent output, it is suggested that the PAGE option be used in the first PUT statement referring to the standard system output file.

6. The F-Compiler gives warning diagnostics for, and effectively ignores, any ENVIRONMENT attribute options valid for the D-Compiler but not valid for the F-Compiler. Such options must be specified in DD statements for the F-Compiler.

7. The keywords SYSIN and SYSPRINT have no meaning under the D-Compiler. However, they do have meaning under the F-Compiler, so care should be taken in using them. "Standard Files" in Part I, Chapter 8 of this publication contains a complete discussion of this subject.

8. When running a D-level program under the F-Compiler, keywords that are not implemented in the D-Compiler, e.g., REAL, COMPLEX, PT, may cause problems. For example, if REAL is an external procedure in a D-level program, the name of that procedure should be changed before the program is run under the F-Compiler. Otherwise, a function reference to REAL will be taken as a reference to the built-in function of that name.

9. Bit-string to arithmetic conversion in the D-Compiler always results in a value whose attributes are FIXED BINARY(31). However, the F-Compiler follows the rules specified in the publication IBM System/360 PL/I Reference Manual, Form C28-8201, and, therefore, will sometimes convert to FIXED BINARY(15).

10. Under the D-Compiler, the order of evaluation of TO and BY expressions in the DO statement proceeds by first evaluating "expression2" and then evaluating "expression3," while the F-Compiler evaluates the expressions in the sequence in which they are specified. Different results can occur only if during evaluation of one of these expressions a function is called and this function changes variables that are used in the other expression.

11. Under the F-Compiler, the character value of a numeric character data item when all digit positions (integer and fractional) have been suppressed, will contain a drifting character in the rightmost digit position, if all digit position in the field have employed that drifting character. Under the

D-Compiler, this drifting character does not appear; the character value consists entirely of blanks.

12. For the F-Compiler, the length of the reply in the REPLY option of the DISPLAY statement cannot exceed 72 characters; for the D-Compiler, this length is not restricted.

(If more than one page number is given, the primary discussion is listed first.)

Printed in U.S.A.

C28-8202-0

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]