COO-3077-151

Courant Mathematics and

Computing Laboratory

U.S. Department of Energy

The PL/I Programming Language

Paul Abrahams

Research and Development Report
Prepared under Contract EY-76-C-02-3077
with the Office of Energy Research

Mathematics and Computing
March 1978

New York University

UNCLASSIFIED

Courant Mathematics and Computing Laboratory

New York University

Mathematics and Computing COO-3077-151

THE PL/I PROGRAMMING LANGUAGE

Paul Abrahams

March 1978

U. S. Department of Energy

Contract EY-76-C-02-3077

This report is to appear as an article on the PL/I programming language in the Encyclopedia of Computer Science and Technology published by Marcel Dekker, Inc.

TABLE OF CONTENTS

INTRODUCTION

PL/I is a large and powerful multipurpose programming language. The intent of the designers of PL/I was to create a language that could be used in business and in scientific applications, as well as in systems programming applications such as writing operating systems. The original design was developed in 1963 by a committee of people drawn from IBM and from SHARE, an IBM user group. For a long time the only important implementations of PL/I were those developed by IBM on the 360 and 370 computers, and the implementation on the GE 645 at the MULTICS Project at MIT. However, during the early 1970's a number of other implementations arose. The implementation of PL/I by other organizations was given impetus by the development of a national and international standard for PL/I by a subcommittee of the American National Standards Institute, in conjunction with a similar subcommittee of the European Computer Manufacturers' Association.

The definition of Standard PL/I was formally released late in 1976, but the content of the standard was publicly known well before then. The standard itself was written in a novel manner as a set of algorithms, expressed in highly stylized English, for the operation of a hypothetical PL/I machine. The version of PL/I described in this article is Standard PL/I.

The design of PL/I drew heavily on the major languages that existed in 1963: Fortran, Cobol, and Algol 60. The syntax of PL/I most resembles that of Fortran, but without

Fortran's rigid rules for program formatting. The notion of block structure was taken from Algol 60, while PL/I structures were taken from the record descriptions of Cobol. However, a great many features were added to PL/I that have no counterpart in its ancestor languages.

An example of a PL/I program is given in Figure 1. A program is written as a sequence of *external procedures*, which are defined in such a way that they can be compiled separately and then linked together when the program is executed. Within an external procedure, there can be *internal procedures*. In this example, there is one internal procedure, named GET_DIGRAM. Each procedure in the program constitutes a block. In addition, a block can be delimited by the PL/I statements BEGIN and END (as in Algol 60).

The internal procedure GET_DIGRAM communicates with the outer procedure DIGRAMS via arguments passed to GET_DIGRAM by the CALL statement. From the viewpoint of GET_DIGRAM, these arguments appear as parameters and are listed in the PROCEDURE-statement.

The variables used in this program are given in the DECLARE-statements. In general, a variable (or other use of an identifier) is described by a set of *attributes*. Not all of these need be given in the DECLARE-statement; those that are not given are deduced through the application of a set of defaulting rules. In fact, defaulting is applied in a great many contexts within PL/I.

On account of its comprehensive nature, PL/I is a difficult language to learn in its entirety. For that reason it was designed so that a user could learn just those parts of the language that he needed, and ignore the rest of it until the occasion arose to use some previously untried feature. The extensive defaulting conventions were included, for a large part, to make it possible to write programs without having to learn about obscure and irrelevant attributes. For instance, one can write business programs in PL/I without ever realizing that the language includes complex numbers and an extensive repertoire of mathematical builtin functions.

Since Standard PL/I is intended to be implemented on a variety of machines, the standard provides that a number of characteristics of the language are implementation-defined. For example, machines differ in their word lengths; therefore the maximum number of digits that need be carried in a floating point computation is left implemented-defined. In the description of PL/I given in this article, implementation-defined features of the language are referred to frequently.

As of this writing, a standard subset version of PL/I is under development by the American National Standards Institute. The definition of this standard subset will probably be released by the time that this article appears. Moreover, an extension of the subset to include facilities for real time and concurrent programming is also under development by the same group.

Figure 1. A Sample PL/I Program

```
/*  THIS PROGRAM READS IN A TEXT AND COUNTS THE NUMBER OF TIMES
    THAT EACH ALPHABETIC DIGRAM OCCURS.  A DIGRAM IS A SEQUENCE
    OF TWO ADJACENT CHARACTERS.  FOR INSTANCE, THE DIGRAMS IN
    'GRUNGE" ARE GR, RU, UN, NG, AND GE.
*/
DIGRAMS: PROCEDURE OPTIONS (MAIN);
         DECLARE COUNT(26,26) FIXED(4);
         /* COUNT(I,J) GIVES THE CURRENT COUNT OF OCCURRENCES
            OF THE DIGRAM  FORMED FROM THE I-TH LETTER AND
            THE J-TH LETTER.
         /*
         DECLARE ALPH CHARACTER(26) INITIAL
           ( 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' );
         DECLARE (P1,P2) FIXED; /* ALPHABETIC POSITION OF LETTER */
         DECLARE (L1,L2) CHARACTER(1);
         /* FIRST AND SECOND LETTERS OF DIGRAM */
         DECLARE DONE BIT(1) INITIAL('0'B);
         /* COMPLETION FLAG */

/* SET ALL ELEMENTS OF THE COUNT ARRAY TO ZERO */

         COUNT = 0;

/* READ AND PROCESS DIGRAMS */

RDLOOP:  DO WHILE('1'B);  /* DO FOREVER */
         CALL GET_DIGRAM(L1,L2,DONE);
         IF DONE THEN
              GO TO PRINT;
         P1 = INDEX(ALPH,L1);
         P2 = INDEX(ALPH,L2);

/* N.B.--INDEX RETURNS ZERO IF LETTER NOT IN ALPHABET */

         IF P1 * P2 > 0 THEN   /* DIGRAM IS ALPHABETIC */
              COUNT(P1,P2) =COUNT(P1,P2)+1;
         END RDLOOP;

/* PRINT THE RESULTS */

PRINT:   DO P1 = 1 TO 26;
              DO P2 = 1 TO 26;
                 IF COUNT(P1,P2)>0 THEN  /* DIGRAM APPEARED */
                    PUT EDIT(SUBSTR(ALPH,P1,1),
                     SUBSTR(ALPH,P2,1), COUNT(P1,P2))
                      (SKIP,2 A(1), X(2), F(4));
                 END;
              END;

         STOP;  /* END EXECUTION OF PROGRAM */
```

Figure 1. Continued

```
/*  INTERNAL PROCEDURE TO EXTRACT THE NEXT PAIR
    FROM THE INPUT TEXT
/*

GET_DIGRAM: PROCEDURE(L1,L2,FLAG);
            DECLARE(L1,L2) CHARACTER(1);  /* LETTER PAIR */
            DECLARE FLAG BIT(1);  /* END-OF-DATA INDICATOR */
            DECLARE CARD CHARACTER(80);   /* INPUT LINE IMAGE */
            DECLARE POSN FIXED STATIC INITIAL(80);
                                  /* CHARACTER POSITION IN INPUT
                                                        CARD */
            DECLARE SYSIN RECORD INPUT FILE;
                  /* INPUT READ FROM FILE SYSIN */
            ON ENDFILE(SYSIN)     /* WHEN INPUT EXHAUSTED */
               GO TO INPUT_FINISHED;
            IF POSN > 79 THEN DO;
               READ FILE(SYSIN)  INTO(CARD);  /* READ A CARD */
               POSN = 1;  /* PROCESS FROM START OF CARD */
               END;
            L1 = SUBSTR(CARD, POSN, 1);
            L2 = SUBSTR(CARD, POSN+1,1);
            POSN = POSN+1;  /* MOVE TO NEXT PAIR */
            RETURN;         /* EXIT FROM THIS PROCEDURE */

/*  COME HERE IF THE READ STATEMENT ENCOUNTERED AN END OF FILE */

INPUT_FINISHED:
            FLAG = '1'B;     /* SIGNAL COMPLETION TO CALLER */
            RETURN;
            END GET_DIGRAM;

            END DIGRAMS;
```

Syntactic Conventions

A PL/I external procedure consists of a sequence of *statements*. With the exceptions of the IF-statement and the ON-statement, every statement is followed by a semicolon. The program is presented in free field format, i.e., statements do not occupy a fixed position on the line. In fact, line boundaries are ignored altogether, so a statement can be split over several lines, or several statements can occupy a single line. With the exception of the assignment statement and the null statement, the type of a statement is indicated by the *keyword* with which it begins.

A statement, in turn, is written as a sequence of *tokens* each of which may be either a *delimiter* or a *nondelimiter*. The types of tokens are:

Delimiters	Nondelimiters
operator	identifier
period	arithmetic constant
comma	string constant
left or right parenthesis	isub [discussed below]
colon	
semicolon	
text inclusion	

Two adjacent nondelimiters must have at least one blank between them, and other than that adjacent tokens may have any number of blanks between them. For example the statement

```
DO IVAL = Q TO (A + 3);
```

could be written more compactly as

```
DO IVAL=Q TO(A+3);
```

but it could not be further condensed to

```
DOIVAL=QTO(A+3);
```

The last example is in fact a valid PL/I statement with an entirely different meaning. A *comment* may be used in any place where a blank can appear. A comment is written as the characters "/*" followed by a sequence of characters not containing "*/" , followed by "*/" , e.g.,

```
/* THIS IS A COMMENT */
```

An identifier consists of a letter followed by any number of letters, digits, and break characters "_" , e.g., POPE_LEO_THE_15TH. The syntax of most of the other kinds of tokens is discussed below.

A *text inclusion* has the form

%INCLUDE *textname* ;

The *textname* refers to an externally stored piece of text, which replaces the text inclusion when the program is translated. On account of the variation in operating environments on different machines, the interpretation of the *textname* is implementation defined. With this facility, it is possible to use the same version of a chunk of program in many different external procedures, even ones written by different programmers.

Keywords can be used as identifiers; in this respect PL/I differs from COBOL, which treats keywrods as reserved words,

and from ALGOL, which uses a distinct typeface to represent them. For example,

```
IF IF THEN THEN = ELSE ;
```

is a valid sequence of statements in PL/I. The first statement is an IF statement that tests the variable IF; the second is an assignment statement whose target is the variable THEN. Most of the long key words can be abbreviated, e.g., CTL for CONTROLLED or NOFOFL for NOFIXEDOVERFLOW. The names of some builtin functions can also be abbreviated.

There are three different kinds of PL/I statements that head groups of statements: the DO statement, the PROCEDURE statement, and the BEGIN statement. For all three, the group is ended by an END statement. If a statement name is attached to the statement that heads a group, then the same statement name can also be attached to the END statement that terminates the group, thus indicating which statement is closed out by the END. A single END statement can close out more than one group, however, as the following example illustrates:

```
ALEPH:  PROCEDURE;
        ...
        DO;
        ...
BEIT:   BEGIN;
        ...
        END BEIT;
        ...
        END ALEPH;
```

The final END statement closes out both the leading PROCEDURE statement and the DO statement that follows it.

The different kinds of data in PL/I can be classified into groups called *data types*, or simply *types*. The available types are either *aggregate types* or *scalar types*. An aggregate is composed from simpler types, and can be either an array or a structure. Arrays and structures are discussed below. The scalar types can be grouped into *printable* and *nonprintable* types, sometimes known as *computational* and *noncomputational*. For each type, there can be variables and values of that type; for some types there can also be constants.

A constant associates a name with a single unchanging value, while a variable associates a name with a location where a value can be stored. The value of a variable is in general time-dependent. Variables are introduced into the program by DECLARE-statements, e.g.,

```
DECLARE LETTER_SEQUENCE CHARACTER(15) VARYING;
```

which declares the variable LETTER_SEQUENCE to have character strings of length from 0 to 15 as its values. However, a variable can be declared even though no DECLARE-statement is written for it (see "Declarations" below), and certain kinds of constants are also introduced through DECLARE-statements.

Arithmetic Types

The printable types consist of the *arithmetic types* and the *string types*. The arithmetic types are characterized by four kinds of attributes: the *base* (binary or decimal),

the *scale* (fixed or float), the *mode* (real or complex) and the *precision*. Since all combinations of base, scale, and mode are permitted, there are eight arithmetic types, precision aside. The precision of a fixed type consists of a *number-of-digits* and a *scale-factor*; that of a float type consists just of the number-of-digits. For example, the type REAL FIXED DECIMAL(6,2) (the "(6,2)" indicates the precision) contains values of the form ± DDDD.DD, where the D's are decimal digits. If the scale factor is omitted, it is taken as zero. The binary types are similar, except that binary rather than decimal digits are used. The fixed and float types correspond to the fixed point and floating point arithmetic data available on most computers. FIXED is a generalization of the *integer* type found in a number of other programming languages, since the *integer* type does not provide for scaling. FLOAT corresponds to the *real* type of other languages. A notable feature of the float types is that they can be used to express the desired accuracy of a numerical computation independently of the word length of the computer carrying out the computation.

The type of an arithmetic constant is indicated by its form. A real fixed decimal constant consists of a sequence of decimal digits with an optional decimal point and sign, e.g., - 17.76 . Fixed constants can be scaled; for instance, 78F-4 has the value 0.0078, obtained by multiplying 78 by 10^{-4}. A real float decimal constant consists of a real fixed decimal constant followed by an exponent part indicated by the letter E,

e.g., 4.832E+12. Binary constants are formed similarly, except that only binary digits are used, and the number is followed by the letter B, e.g., 10.1B or 11E16B (designating 3×10^{16}). Complex constants do not exist as such; a complex constant is formed as the sum of a real constant and an imaginary constant, e.g., 4+3I.

String Types

The string types are *character* and *bit*, each of which in turn may be *varying* or *nonvarying*. However, string values are sequences of characters or bits, and the varying and nonvarying attributes are not applied to them. Each string type has a maximum length associated with it; for the nonvarying types, the actual length is always equal to the maximum length. For instance, the type CHARACTER(14) VARYING describes character strings whose length varies from 0 to 14 characters, while the type BIT(8) NONVARYING (NONVARYING is the default) describes bit strings that are always exactly 8 bits long. A character-string constant is written as a sequence of characters enclosed in single quotes, with internal quotes doubled, e.g.,

'THE FARMER''S DAUGHTER'

The null character string, which contains no characters, is written as ''. A bit-string constant is written as a sequence of binary digits enclosed in single quotes and followed by B, e.g., '101001'B; the null bit string is written as ''B. Bit strings can also be written in base-4, base-8, or base-16

notation. For instance, '7400'B3 indicates the base-8 (octal) constant 7400 (equivalent to '111100000000'B), while 'A81'B4 indicates a base-16 constant. (The digit after the B indicates a power of 2.) This extended notation is not available for binary arithmetic constants. The one-bit values '1'B and '0'B are particularly useful, as they are the results returned by the PL/I comparison operators; '1'B represents *true* and '0'B represents *false*.

When a string, either bit or character, is declared, the maximum length need not be given by a constant, so that the declaration

```
DECLARE NEWSTR CHARACTER(K1+2);
```

is permissible. The expression K1+2 must be well-defined at the time that NEWSTR is created. Strings that appear as parameters of procedures may have their maximum lengths given by *, e.g.,

```
DECLARE PARAM_3 CHARACTER(*);
```

In this case, the maximum length of PARAM_3 is determined by the argument corresponding to PARAM_3. Strings used as parameters must have their maximum lengths given either by * or by an expression composed purely of constants; more general expressions are not permitted (but are not particularly useful in this context in any case).

Pictured Types

The *pictured* types are derived from similar types in COBOL, but are more general. A pictured type has an associated picture, e.g., 999V.99, that describes the appearance of the values of that type. The values are represented as character strings, and the semantics of PL/I are such that an implementation is actually obliged to store them that way. There are no constants of pictured type. Pictures can be used in input-output formats as well as in declarations; an example of the declaration of a pictured type is

```
DECLARE SALARY PICTURE '$$$,$$$V.$$';
```

A pictured type has a picture associated with it. The picture is given by a character string. Within the picture, parenthesized counts can be used to indicate repeated characters, so that the picture '$$$$$' can also be written as '(5)$'. A picture can be either a character picture or a numeric picture. A numeric pictured type contains, in addition to the picture itself, a mode specification (either REAL or COMPLEX).

Character pictures are rather simple. They consist of just the characters A, 9, and X. The character A stands for a letter or a blank; the character 9 stands for a digit or a blank; and the character X stands for anything. Character pictures are used to validate strings, i.e., to insure that they are in the proper form. Thus, if we have the declaration

```
DECLARE STRING_TEST PICTURE '(3)AXX9';
```

we can assign to STRING_TEST values consisting of three letters

(or blanks) followed by any two characters followed by a single digit (or blank). If the assigned value does not have these characteristics, an error will be signalled.

A numeric picture is one that contains a character other than A, 9, or X. By this definition, a picture consisting of all 9's is a character picture rather than a numeric picture. A numeric pictured type has an associated arithmetic type, which is determined by the form of the picture together with the mode. The picture itself is independent of the mode. Thus if we have the declarations

```
DECLARE RPIC REAL PICTURE 'ZZZ';
DECLARE CPIC COMPLEX PICTURE 'ZZZ';
```

RPIC has an associated arithmetic type of REAL FIXED DECIMAL(3,0), while CPIC has an associated arithmetic type of COMPLEX FIXED DECIMAL(3,0). (The associated arithmetic type is necessarily decimal.) If the picture contains either of the characters E or K, it is a float picture (i.e., its associated arithmetic type is float); otherwise, assuming it is numeric, it is a fixed picture. The associated arithmetic type can be thought of as specifying the meaning of the values, as distinct from the representation of the values. The meaning becomes important when pictured values are used in arithmetic operations.

When a numeric value is assigned to a pictured variable, the value is edited to conform to the picture. The characters in the picture determine how the editing is to be done, assuming that the value has already been converted to the associated

arithmetic type. Editing by means of a picture is illustrated by the following example: Suppose that the value 34.8 is to be edited using the picture S9999V.99. Then the pictured value will be +0034.80. In this example the S indicates an explicit sign, the 9's indicate explicit digits, the V indicates an implicit decimal point (used to align the numeric value with the characters of the picture) and the period is an insertion character. The meanings of the different picture characters are given in Table 1, and their use is illustrated in Table 2. The I, R, and T characters represent digits with sign-overpunching, i.e., the sign of the entire value is combined with the digit to form a single character. Sign-overpunching is the standard input convention for COBOL; on a keypunch, the characters are formed by punching both a sign (11-row for -, 12-row for +) and a digit in a single column. Although CR and DB are two characters rather than one, the two characters always go together, and are used to indicate a negative quantity. CR stands for "credit" and DB for "debit".

The drifting characters $, Z, +, -, and S are used to edit leading zeros into blanks. When a sequence of drifting characters appears (they must all be the same one), the character drifts to the position to the left of the leading nonzero digit in the value, and the remaining positions to the left are blanked out. Any insertion characters within the blanked-out positions are themselves blanked out.

The period is a true insertion character, in that its presence or absence has no effect on the value represented by

the picture. The period need not appear next to the V, even though the sequence "V." is often used. A single $, S, +, or - can be placed at the beginning or at the end of a picture, in which case it signifies an explicit insertion rather than a drifting position. For instance, the value 92 edited by the picture '(5)$V.$$S' yields the string 'ØØ$92.00+'; in this case the S causes a sign to be inserted, and is not a drifting character.

Although pictures are ordinarily used to edit real values, they can be used to edit complex values also. When a variable is declared to be pictured and complex, the picture is used to edit both the real and imaginary parts of any complex value assigned to the variable, and the two parts are concatenated. However, there is no way to use pictures to insert "I" into the edited representation of a complex number.

Table 1. Meaning of Picture Symbols

(a) Character-picture symbols

A	alphabetic character
9	digit or blank permitted
X	anything permitted

(b) Numeric-picture symbols

9	digit
Y	digit with zero mapped to blank
Z	digit with zero-suppression
$	drifting or inserted dollar sign
*	drifting asterisk (check protection)
+	drifting or inserted sign for positive values
-	drifting or inserted sign for negative values
S	drifting or inserted sign for all values
CR	credit symbol, inserted for negative values
DB	debit symbol, inserted for negative values
I	digit with positive value indicated by overpunch
R	digit with negative value indicated by overpunch
T	digit with sign always indicated by overpunch
.	inserted period
,	inserted comma
B	inserted blank
/	inserted slash
V	implied decimal point
E	start of exponent, E inserted
K	start of exponent, nothing inserted

Table 2. Examples of Numeric Pictures

Picture	Numeric Value	Pictured String
999	7	007
99V9	7	070
YY/YY/YY	760404	76/Ø4/Ø4
ZZZZ	23	ØØ23
Z,ZZZ	123	ØØ123
Z,ZZZ	1234	Ø1,234
Z.VZZS	-1.6	1.60-
Z.VZZ+	.03	ØØ03+
ZV.ZZ+	.03	Ø.03+
-ZZZZ$	0	ØØØØØØ
-ZZZZ$	2	ØØØØ2$
+ZZZZ$	2	+ØØØ2$
ZZ99	0	ØØ00
$$$V.$$	1.23	b$1.23
$$B$$$	2345	$2Ø345
$$B$$$	345	ØØ$345
+++	6	Ø+6
---	6	ØØ6
SSS	6	Ø+6
$***V.**	.07	$***.07
$***.V**	.07	$****07
$$$99CR	8	ØØ$08ØØ
$$$99CR	-123	Ø$123CR
$$$99DB	-123	Ø$123DB
999T	71	07A
999T	-71	07J
I999	1776	A776
I999	-1776	1776
ZZZR	71	71
ZZZR	-71	7J
99.V999BKS99	123456	12.346Ø+04
V.99999E99	123	.12300E03
999V9F3	12345	0123
99999F-2	12.34	01234

Note: A indicates 1 with postiive overpunch.
J indicates 1 with negative overpunch.

Pointers, Areas and Offsets

The nonprintable types of data in PL/I are pointers, areas, offsets, files, labels, entries, and formats. A *pointer* can be thought of as the location of a piece of data; it resembles the *ref* (reference) notion of Algol 68. However, pointer variables in PL/I are untyped; that is, a pointer variable can contain a pointer to data of any type whatsoever. The only pointer constant is the null pointer, which does not point at anything and therefore does not compare equal to any pointer to an existing object. The null pointer is obtained as the value of the builtin function NULL of no arguments. Pointers are used in conjunction with based variables, which act as templates for an area of storage. Based variables are discussed below.

An *area* is a region in which space for based variables can be allocated. Areas can be cleared of their allocations in a single operation, thus allowing for wholesale freeing. Moreover, areas can be moved from one place to another by means of assignment to area variables, or through input-output operations. There is one area constant, the empty area, which is obtained as the value of the builtin function EMPTY of no arguments. Assignment of the empty area to any area variable clears the area of its allocations. More precisely, the old value of the variable is destroyed (though a copy may exist elsewhere), and the new value is an area with nothing allocated in it. The declaration of an area specifies (at

least by default) an area size in implementation-defined units (bytes, words, etc.), e.g.,

```
DECLARE STRUCTURE_AREA AREA(2000);
```

When an area is moved, pointers to objects within the area lose their validity. Therefore, PL/I also provides *offsets*, which are pointers relativized to the origin of a given area. When an area is moved, the offsets of the objects within the area remain unchanged. Conceptually, pointers and offsets are related by the equation

```
pointer = offset + area
```

but in an actual implementation that equation need not hold. There is one offset constant, the null offset, which is obtained by converting the null pointer to an offset.

To make it easier to work with offsets, it is possible to declare an offset with an implicit area association (which can be overridden), e.g.,

```
DECLARE OFF_FROM_A3 OFFSET(A3);
DECLARE A3 AREA(300);
```

When OFF_FROM_A3 is referenced in a context where a pointer is required, the offset value in OFF_FROM_A3 is converted to a pointer relative to the area A3.

Files

A *file* is, conceptually, a port through which communication is established between the program and a *dataset*. A dataset, in turn, is a collection of information residing on an external

medium, accessible to the program only through input-output operations. During the course of execution of a program, a given file may be connected to different datasets, or to no dataset, at different times. Input-output operations reference a file, which must be connected to an appropriate dataset; the operations then take place on the dataset. A file connected to a dataset is said to be *open*; one not connected to a dataset is said to be *closed*.

The file itself is a file value; each file value is uniquely associated with a file constant, declared, for example, by

```
DECLARE FILECON FILE CONSTANT;
```

File variables are declared similarly, e.g., by

```
DECLARE FILEVAR FILE VARIABLE;
```

If neither CONSTANT nor VARIABLE is specified in the declaration, the usual default is CONSTANT. Moreover, declarations of file constants are introduced implicitly in a number of contexts, so that in practice the programmer rarely needs to write these declarations. For instance, the PL/I statement

```
PUT LIST(A,B);
```

causes the values of A and B to be written onto the dataset associated with the file named SYSPRINT (assumed since no other file was specified in the PUT-statement). If no declaration is explicitly given for SYSPRINT, the declaration

```
DECLARE SYSPRINT FILE CONSTANT;
```

is assumed. When the PUT-statement is executed, the file SYSPRINT is opened as a PRINT file (assuming it is not already open).

Labels

A *label* is a name attached to an executable statement so that control can be transferred to that statement by means of a GOTO-statement. A label value has two components: a designator (such as an address) of the statement named by the label, and an *environment*, which records the state of execution of the program at the time when the block containing the label was entered. The environment is necessary because the address by itself does not always provide sufficient information to determine unambiguously the state of execution after a GOTO-statement has been carried out (see the section "Entry Values and Environments" below).

Label constants are declared by the appearance of a label as a *statement-name*, as in

```
BOOK_FOUND: VOLUME = FOLIO(J);
```

which declares BOOK_FOUND as a label constant. In fact, label constants cannot be declared in any way other than by their appearance as statement-names. Not all statement-names declare label constants, however; some of them declare entry constants and format constants. The type of constant declared by a statement-name is determined by the type of statement to which it is attached. Label variables are declared using the attribute LABEL, e.g.,

```
DECLARE LABVAR LABEL VARIABLE;
```

A statement-name can be written with one or more subscripts, and by this convention constant arrays of labels can be created. For instance, if a block contains executable statements with the statement-names CASE(-1), CASE(0), CASE(1), and CASE(2), the appearance of these statement-names constitutes a declaration of CASE as a constant array of labels, whose single subscript has a lower bound of -1 and an upper bound of 2. It is quite permissible to attach several of these subscripted statement-names to a single statement, to give them in nonincreasing order, to attach them to statements also bearing nonsubscripted statement-names, or even to omit some index values from the set. For instance, if CASE(0) were omitted from the above set, the set would still be valid, but a transfer to CASE(0) would be in error. Typically, an element of a constant array of labels is selected by a statement such as

```
GO TO CASE(CASE_NUMBER);
```

Entries

An *entry* is an entry point to a procedure (see "Procedures, Scopes, and Environments" below) treated as a datum. An entry constant is declared by the appearance of a statement-name attached to a PROCEDURE-statement or an ENTRY-statement. For instance,

```
PROCESS_NAME:  PROCEDURE(NAME,SPECS);
```

declares PROCESS_NAME as an entry constant, whose associated

value is the (single) entry point to the procedure headed by the PROCEDURE-statement. Entry values, like label values, carry environments with them. Entry variables and arrays of entry constants are available. A typical application of an entry variable arises in writing a procedure to integrate an arbitrary function; within the procedure, the function is declared as an entry variable (and a parameter), and the actual function to be integrated is passed as an argument.

When an external procedure references an entry in another external procedure, then the first procedure must declare the second explicitly as an entry constant, as in

```
DECLARE SEARCHVAL EXTERNAL ENTRY(CHARACTER(*))
     RETURNS(FIXED);
```

The CONSTANT attribute is assumed by default in this case. The procedure containing this declaration expects SEARCHVAL to be an entry to an external procedure, whose expected argument is a character string of unspecified length, and which returns a fixed value.

Formats

A *format* is used to specify the form of data on a dataset accessed through a stream file (see "Edit-Directed Input-Output" below). A format constant is declared by the appearance of a statement-name on a FORMAT-statement, e.g.,

```
FMT3: FORMAT(SKIP,3A,X(M),A);
```

As with labels and entries, format variables and arrays of format constants are included in PL/I. Format variables

are declared using the attribute FORMAT, e.g.,

```
DECLARE FMTVAR FORMAT VARIABLE;
```

Since formats can contain references to variables, e.g., the M in the example FMT3 above, format values carry environments.

Arrays

Two types of aggregates are provided in PL/I: *arrays* and *structures*. An array is a collection of elements all having the same type; a particular member of the collection is selected using an appropriate sequence of subscripts. A structure, on the other hand, is a collection of elements having possibly different types; a particular member of the collection is selected by using an appropriate name as the selector. A powerful feature of PL/I is that it allows aggregates to be treated as data objects in most contexts, so that it is possible, for instance, to add two arrays in a single operation, or to write a procedure that returns a structure as its value.

An array is characterized by a sequence of *dimensions*; the *dimensionality* of the array is the number of its dimensions. Each dimension has a lower bound and an upper bound, and these can be arbitrary integers. For example, the declaration

```
DECLARE MESH(-100:100,200) FLOAT BINARY(40);
```

declares MESH to be a two-dimensional array. The first subscript has lower bound -100 and upper bound 100, while the second has lower bound 1 (assumed since none is given) and

upper bound 200. When reference is made to an element of an array, the subscripts can be arbitrary expressions, as long as the values of those expressions can be converted to integers.

As with character strings, bounds can be given by expressions as well as constants. For an array parameter, a pair of bounds (not a single one) can be given by *. The * can also be used in a quite different sense to indicate a cross-section of an array. For instance, MESH(3,*) designates a one-dimensional array, with lower bound 1 and upper bound 200, consisting of those elements of MESH whose first subscript is 3. Any number of the subscripts in an array reference can be replaced by *'s; the dimensionality of the resulting array is the number of *'s.

Array variables take on array values. Array values can arise during the evaluation of an expression, e.g., when two arrays are added together. Aside from labels, entries, and formats, there are no array constants in PL/I.

Structures

A structure is a collection of named elements, each of which can itself be a structure. An example of a two-level structure declaration is

```
DECLARE
   1  EMPLOYEE_RECORD,
      2  NAME,
         3  FIRST CHARACTER(10) VARYING,
         3  MIDDLE_INITIAL CHARACTER(1),
         3  LAST CHARACTER(15) VARYING,
      2  ID_NUMBER FIXED DECIMAL(9),
      2  SALARY FIXED DECIMAL(7,2);
```

The number in front of each component is a level number. The structure as a whole is at level one. The members of the level-one structure are the level-two components; those of the level-two components are the level-three components, etc. It is possible to write structure declarations using nonconsecutive level numbers, but there is always an equivalent structure using consecutive ones. In any event, the logical levels are always consecutively numbered.

The organization of a structured value is just like that of a structured variable. A structured value contains a number of components, and can be treated as a single object. For example, two structures can be added just as two arrays can be added. There are no structure constants.

The elements of a structure are referred to using *qualified names*, although abbreviated versions are permissible. The fully-qualified names of the elements of the structure given above are:

```
EMPLOYEE_RECORD  (itself a structure)
EMPLOYEE_RECORD.NAME   (itself a structure)
EMPLOYEE_RECORD.NAME.FIRST
EMPLOYEE_RECORD.NAME.MIDDLE_INITIAL
EMPLOYEE_RECORD.NAME.LAST
EMPLOYEE_RECORD.ID_NUMBER
EMPLOYEE_RECORD.SALARY
```

Abbreviated versions of qualified names are obtained by leaving out any of the component identifiers other than the last one.

These abbreviated forms can be used as long as the result is not ambiguous, i.e., as long as it cannot refer to more than one object.

Arrays of structures and structures of arrays are possible, and can be nested to any depth. An example of such a nested structure is given by:

```
DECLARE
   1  CAR(30),
      2  COUNTRY_OF_ORIGIN FIXED DECIMAL(3),
      2  DEALERS(40),
         3  CITY CHARACTER(20) VARYING,
         3  STATE CHARACTER(2),
         3  COMPANY CHARACTER(30) VARYING;
```

In such a nested entity, dimensionality is inherited. Thus CITY is a two-dimensional array since one dimension is inherited from CAR and the other from DEALERS. CITY(*,17) designates the array composed of the elements

```
CAR(1). DEALERS. CITY(17)
CAR(2). DEALERS. CITY(17)
            ...
CAR(30).DEALERS. CITY(17)
```

In writing a reference to an element of CITY or a similar object, the subscripts can be written in any position as long as they are in the right order. Thus CITY(*,17) could also have been written as CAR(*).CITY(17) or as DEALERS(*,17).CITY.

DECLARATIONS

A *declaration* associates a set of *attributes* with an identifier. The attributes specify the characteristics of the object denoted by the identifier, such as its data type. Within an external procedure, a given identifier can be declared more than once, since the identifier can be declared in different blocks, or as a member of different structures, or both. Nevertheless, each declaration of an identifier designates a distinct object having its own attributes. The rules for name resolution determine how an occurrence of an identifier is resolved to the appropriate declaration (see "Blocks and Scopes" below).

Every occurrence of an identifier within an external procedure must have a corresponding declaration within that external procedure. Moreover, every declaration must have a complete and consistent set of attributes. To satisfy these principles, PL/I includes extensive conventions for defaulting declarations, i.e., for creating declarations that were not written in the program and for deducing attributes of incomplete declarations.

Manifest, Explicit, Contextual, and Implicit Declarations

An identifier that is declared in a DECLARE-statement written by the programmer is said to be *manifestly* declared.*

* The term "manifest", though convenient, is not standard usage.

An identifier that is manifestly declared, or that appears in a parameter list, or appears as a statement name, is said to be *explicitly* declared. An identifier appearing in a parameter list may, but need not, be manifestly declared; an identifier appearing in a statement-name must not be manifestly declared.

If an identifier appears in a procedure and no explicit declaration exists for that identifier, then a default declaration is created if possible. The default declaration is placed in the outermost block of the external procedure. If the identifier is used in such a way as to suggest what its attributes should be, it is said to be *contextually* declared; otherwise it is *implicitly* declared. For instance, if the identifier OUT has not been manifestly declared and the statement

```
PUT FILE(OUT) LIST(VALX,VALY);
```

appears, then OUT will be contextually declared with the attributes FILE and CONSTANT. Similarly, if the identifier EXP has not been manifestly declared and the statement

```
Y = EXP(A**2);
```

appears, EXP will be contextually declared with the attribute BUILTIN. A program is in error if it induces conflicting contextual declarations.

An implicit declaration is created for an identifier if it is declared neither explicitly nor contextually. In this case, the created declaration initially has no attributes, and all the attributes are added by default later on.

Declarations of Statement-Names

The appearance of an identifier as a statement-name completely determines its attributes. The data type of the identifier is determined by the kind of statement named, as well as whether or not the identifier is subscripted. If the named statement is an ENTRY-statement or a PROCEDURE-statement, the identifier acquires the attributes ENTRY and CONSTANT; if the named statement is a FORMAT-statement the identifier acquires the attributes FORMAT and CONSTANT; and if it is any other statement the identifier acquires the attributes LABEL and CONSTANT. Moreover, if the statement-name is subscripted, then the identifier acquires an appropriate DIMENSION-attribute.

The implied declaration of an entry constant cannot be fully constructed until the types of its parameters are known. When the parameter types are known, the entry constant can be characterized completely. At that point the parameter specifications and the RETURNS-attribute, if any, are added to the declaration. For instance, the statements

```
FN:  PROCEDURE(A,B) RETURNS(CHARACTER(12)VARYING);
     DECLARE A CHARACTER(*);
     DECLARE B POINTER;
       ...
     END FN;
```

lead to the derived declaration

```
DECLARE FN ENTRY(CHARACTER(*),POINTER)
     RETURNS(CHARACTER(12) VARYING) CONSTANT;
```

Attribute Consistency and Completeness

The diagram of Figure 2 can be used to determine whether a set of attributes is consistent and complete, except for a few peculiar cases. In this diagram, the following conventions are followed:

1. Double lines indicate the definition of a term.
2. Rectangles indicate specific attributes. If a rectangle is dashed, it indicates an optional attribute.
3. Ovals indicate sets of attributes.
4. Horizontal lines indicate sets for which all (nondashed) elements must be present).
5. Vertical lines indicate sets from which one alternative must be chosen.

A set of attributes is complete and consistent if it can be constructed from this diagram; it is consistent if it is a subset of a complete and consistent set.

The interpretation of this diagram is illustrated by the attribute set

INTERNAL VARIABLE AUTOMATIC ALIGNED POINTER INITIAL

Starting from the node "consistent-set", a scope and a declaration-type must both be chosen. The scope can be either INTERNAL or EXTERNAL; INTERNAL is chosen. The declaration-type can be a variable, a named-constant, etc.; it is chosen to be a variable. In that case, the attribute VARIABLE must be present, as well as a storage-type and a data-declaration. The storage-type is chosen to be a storage-class; the storage-class

Figure 2. Complete and Consistent Attribute Sets

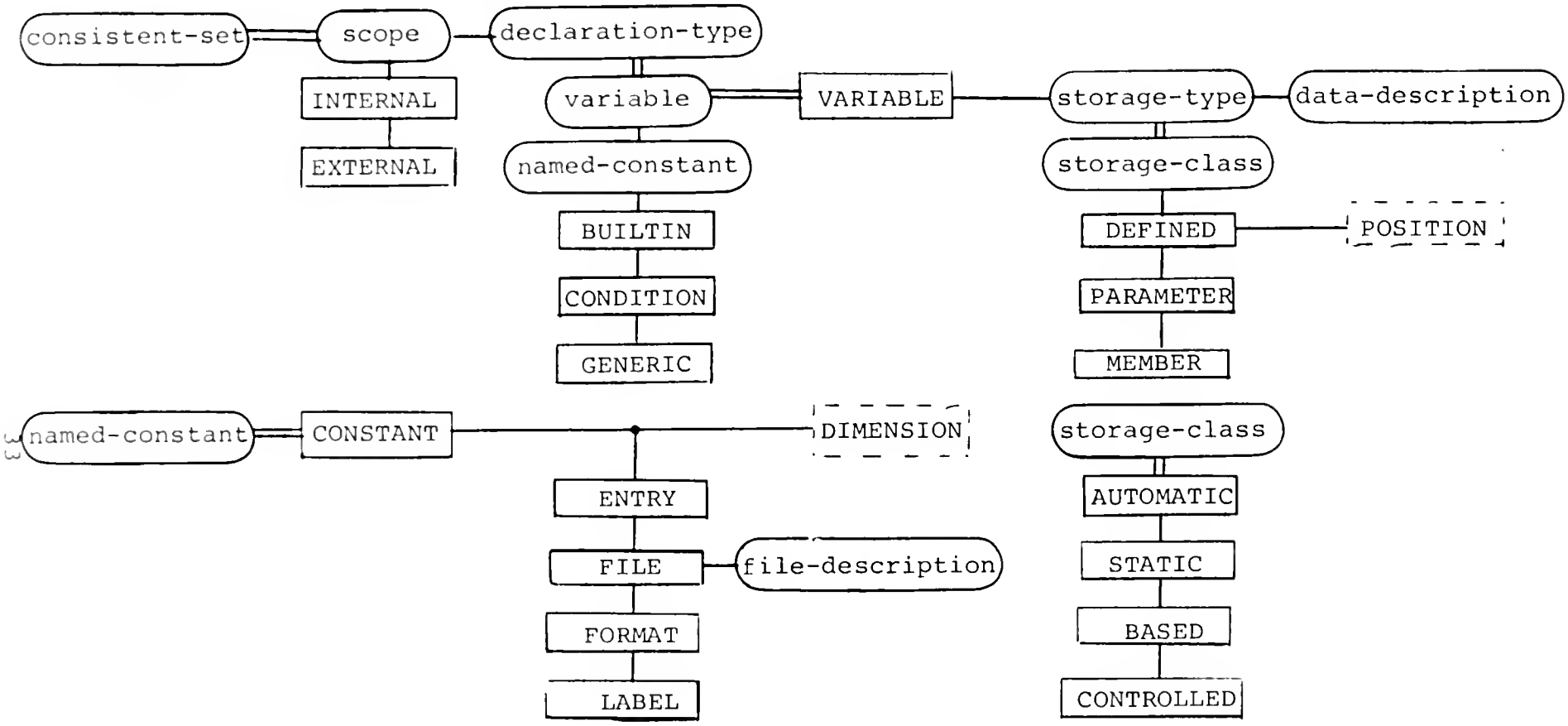


Figure 2. Continued

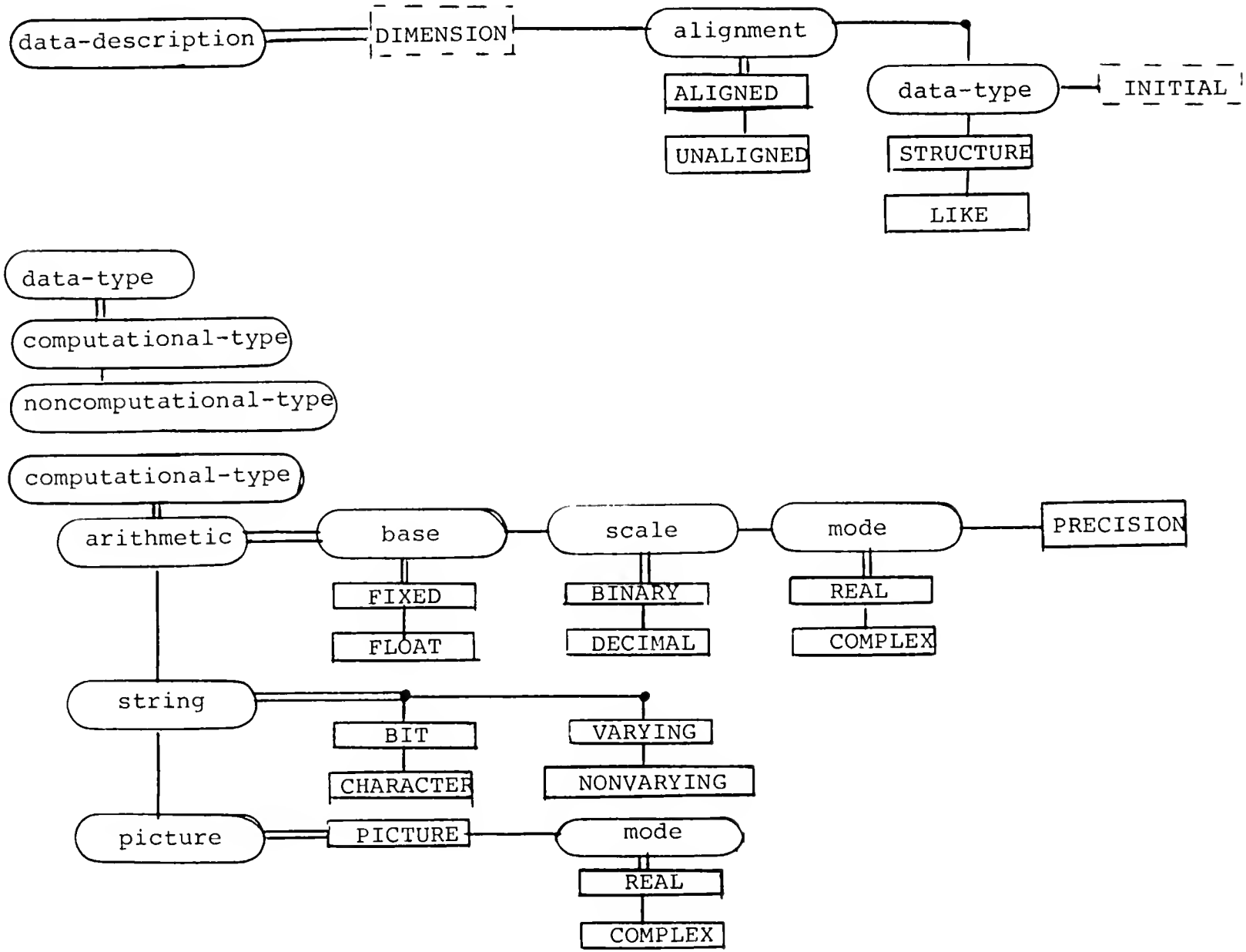
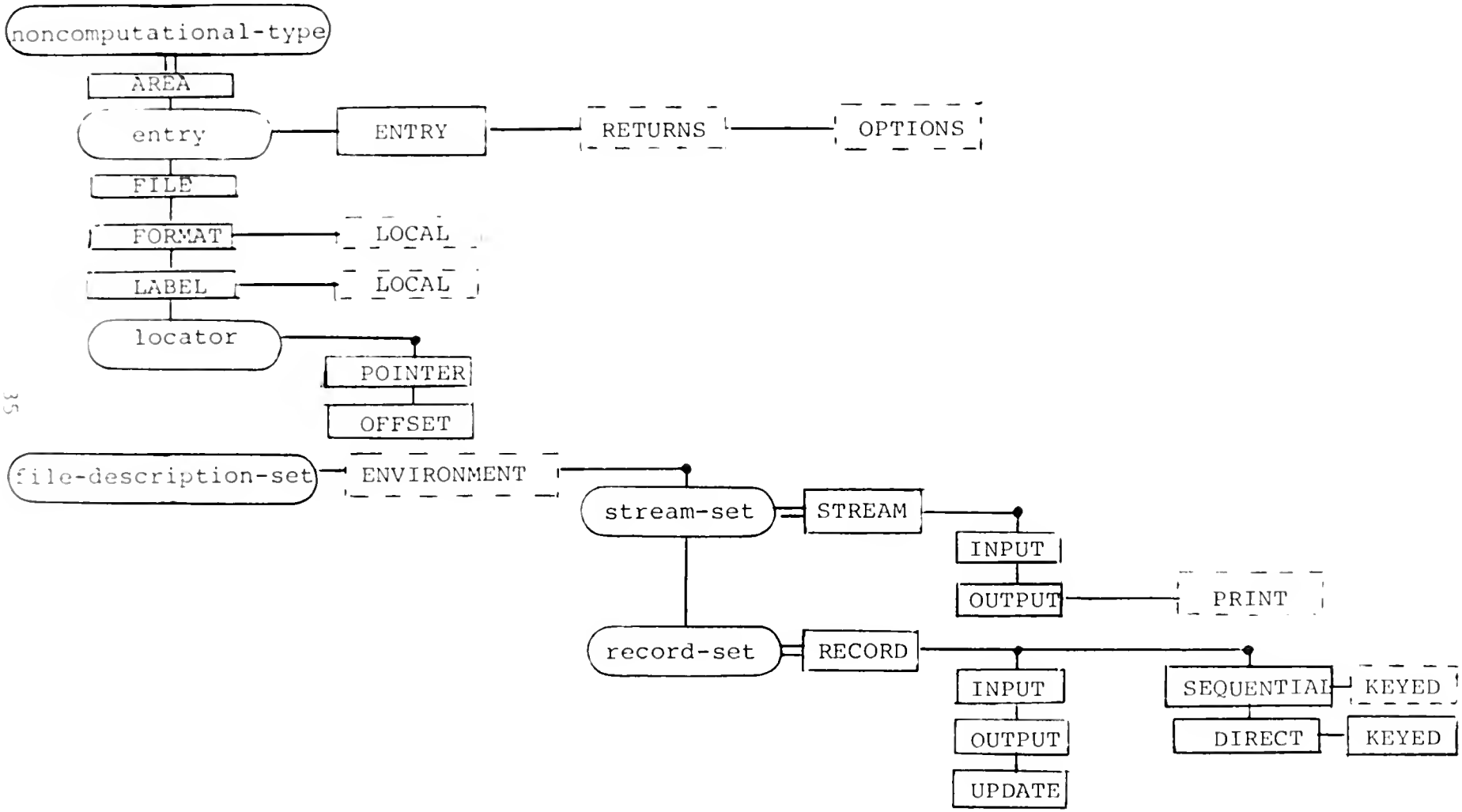


Figure 2. Continued



is chosen to be AUTOMATIC. The data-description may, but need not, contain DIMENSION; in this case it does not contain DIMENSION. The data-description must also contain an alignment, chosen to be ALIGNED, and either a data-type with optional INITIAL, or STRUCTURE. In this case, INITIAL is present (the actual initial values are ignored). The data-type is chosen to be noncomputational. Of the alternatives for non-computational-type, locator is selected; of the alternatives for locator, POINTER is selected. Thus the entire set of attributes is shown to be complete and consistent. (The order in which the attributes are given is immaterial.)

Not all consistency violations are shown up by reference to this diagram. For example, the combination

AUTOMATIC EXTERNAL

and the combination

STATIC LABEL INITIAL

are both invalid, but pass the test of the diagram. Specific auxiliary rules are needed in order to disallow these and similar cases. The requirement for completeness is relaxed for the case of file constants, i.e., declarations with the attributes FILE and CONSTANT; although the file-description-set must be consistent, it need not be complete.

Certain attributes either permit or require subspecification; these are listed in Table 3. For example, the BASED attribute permits, but does not require, the subspecification of an auxiliary pointer (see "Based Storage" below), while the

Table 3. Attributes Requiring or Permitting Subspecification

A. Subspecification Required

Attribute	Required Specification
CHARACTER	length
BIT	length
AREA	size of area
DIMENSION	dimensionality
RETURNS	type of returned value
PRECISION	number-of-digits, possible scale-factor
PICTURE	picture specification
GENERIC	generic specification
ENTRY with CONSTANT	parameter types
POSITION	position count
DEFINED	base variable
LIKE	likened variable

B. Subspecification Permitted

Attribute	Permitted Subspecification
BASED	basing pointer
OFFSET	area
ENTRY with VARIABLE	parameter types

CHARACTER attribute requires the subspecification of a string length.

The ENTRY-attribute and the RETURNS-attribute themselves contain attribute sets as subspecifications. These attribute sets are known as *descriptors*, and must also be complete and consistent. The descriptors in the ENTRY-attribute describe the parameters expected by a procedure, while the descriptors in a RETURNS-attribute describe the value returned by a procedure.* To be complete and consistent, a descriptor must be derivable from the "data-type" node in the diagram of Figure 2, but may optionally contain the MEMBER-attribute. The descriptor for a structure as a whole looks like the declaration of a structure variable with the identifiers left off, e.g.,

```
DECLARE GENFUNC ENTRY (
     1  (*),
        2  FIXED BINARY,
        2  CHARACTER(30) VARYING );
```

Standard and User Defined Defaults

The PL/I defaulting rules specify how an incomplete but consistent set of attributes is to be completed. Although there is a standard set of defaulting rules, the DEFAULT-statement can be used to override them. The defaulting rules, whether standard or user-defined, consist of a *predicate* and a *default attribute set*. The predicate indicates a test to be applied to the attributes already present in the declaration

* Since procedures can accept structures as arguments and can return structures as values, the specification of a single parameter or of a returned value can contain more than one descriptor.

(or descriptor), while the default attribute set indicates additional attributes to be supplied provided that they are consistent with the ones already present. Inconsistency, in this case, is not an error; it simply means that the default is not applied.

The principal standard defaulting rules are:

1. Add the attributes FIXED, REAL, and BINARY. For instance, if FIXED alone is present, REAL and BINARY are added.*
2. If the arithmetic attributes are present but no precision has been specified, add an implementation-defined precision whose subspecification depends on the arithmetic attributes already present.
3. If CHARACTER or BIT is present, assume NONVARYING.
4. If CHARACTER or BIT is present but no length has been specified, assume a length of 1. If AREA is present but no area size has been specified, assume an implementation-defined value for the area size. If POSITION is present but no count has been specified, assume a count of 1.
5. If neither VARIABLE nor CONSTANT has been specified, assume VARIABLE unless ENTRY or FILE is present. If ENTRY or FILE is present by itself, assume CONSTANT. If ENTRY or FILE is present along with other attributes, the default depends on what those other attributes are.
6. If FILE and CONSTANT, or ENTRY and CONSTANT, or CONDITION is present, assume that the scope is EXTERNAL; in all other cases assume that it is INTERNAL.

* This rule differs from the one used in the well-known IBM implementation of PL/I.

7. If EXTERNAL is present, assume that the storage class is STATIC; in all other cases assume that it is AUTOMATIC.

8. If CHARACTER or BIT is present, assume UNALIGNED; otherwise assume ALIGNED.*

As a consequence of the defaulting rules, an implicitly declared identifier, which starts with an empty attribute set, will acquire the attribute set

REAL FIXED BINARY PRECISION($\underline{d}_1$,0) VARIABLE INTERNAL AUTOMATIC ALIGNED

Here $\underline{d}_1$ is the implementation-defined default number-of-digits for the attribute combination FIXED BINARY. Similar constants $\underline{d}_2$,$\underline{d}_3$, and $\underline{d}_4$ are specified for FIXED DECIMAL, FLOAT BINARY, and FLOAT DECIMAL.

The DEFAULT-statement contains a predicate and one or more default attribute sets. Declarations are completed by applying the user-supplied DEFAULT-statements in the order that they appear in the program, and then applying the system default rules. Unlike DECLARE-statements, DEFAULT-statements are order-dependent. A default attribute set is added if and only if the appropriate predicate is satisfied and none of the elements in the set leads to a conflict. The predicate can include both attributes and identifier ranges, indicated by the keyword RANGE. The predicate is formed as a boolean combination of attributes and ranges. RANGE(*) is satisfied by any identifier, but not by a descriptor. RANGE(a1:a2) is satisfied by any

* This rule does not apply to structures. The alignment of a structure, if given explicitly, is passed down to all elementary members of the structure unless a conflicting alignment is given at a lower level.

identifier starting with a letter between a1 and a2 inclusive, while RANGE(init) is satisfied by any identifier starting with *init*. For instance,

```
DEFAULT(RANGE(AB)|FLOAT & ¬BINARY) COMPLEX STATIC;
```

applied to the declaration

```
DECLARE AB35 FIXED;
```

yields

```
DECLARE AB35 FIXED COMPLEX STATIC;
```

and applied to the declaration

```
DECLARE XYZ FLOAT DECIMAL;
```

yields

```
DECLARE XYZ FLOAT DECIMAL COMPLEX STATIC;
```

However, it has no effect when applied to

```
DECLARE AB35 REAL FIXED;
```

since COMPLEX conflicts with REAL. It also has no effect when applied to

```
DECLARE XYZ FLOAT BINARY;
```

since the predicate is not satisfied.

The LIKE-Attribute

The LIKE-attribute can be used to copy part of a structure declaration into another declaration. It is useful when a program uses many similarly organized structures. For instance, if a program includes the declarations

```
DECLARE
   1  ASSEMBLY BASED,
      2  NEXT_PART POINTER,
      2  FIRST_COMPONENT POINTER,
      2  DESCRIPTION,
         3  PART_NUMBER PICTURE 'X(5)9',
         3  COST FIXED DECIMAL(6,2);
DECLARE 1 GROUP(20) STATIC LIKE ASSEMBLY;
```

then the second declaration is equivalent to

```
DECLARE
   1  GROUP(20) STATIC,
      2  NEXT_PART POINTER,
      2  FIRST_COMPONENT POINTER,
      2  DESCRIPTION,
         3  PART_NUMBER PICTURE 'X(5)9',
         3  COST FIXED DECIMAL(6,2);
```

The LIKE-attribute causes copying of members only; attributes at the level of the LIKE-attribute are not copied. In this example, a reference to GROUP(20).NEXT_PART requires that the qualifying identifier GROUP be included, since otherwise the reference would be ambiguous. This behavior is a general property of structures declared using the LIKE-attribute.

EXPRESSIONS, TYPE CONVERSION, AND ASSIGNMENT

The kinds of expressions acceptable in PL/I are similar to those found in most higher-level languages. These are:

literal constants

references to variables and named constants

parenthesized expressions

function calls

prefix expressions

infix expressions

references to builtin functions

The only kinds of literal constants recognized are arithmetic constants and string constants. Other constants are obtained either as named constants, e.g., statement-names, or as the results of builtin functions, e.g., NULL. References to variables may have subscripts, pointer qualifications (see below), and name qualifications. An example of a reference with all three is

```
PT -> X(I,J). B
```

Although the pointer-qualifier symbol -> looks like an operator, it is not treated as one (see "Based Storage" below).

Parentheses are used within expressions in three ways: to designate subscripts of arrays, to designate arguments of functions, and to group components of expressions containing operators. When an expression containing operators, e.g., + and *, is enclosed in parentheses, it is treated as a single

entity -- this is the usual convention in mathematical notation. When a reference to a variable is enclosed in parentheses, it is then treated as a general expression rather than as a variable. This rule only makes a difference in the context of a procedure call.

A function call has two parts: the reference to the function and the argument list. The function reference need not have the form of a single identifier, since functions can return entry values, can be subscripted, and can be members of structures. Function calls, subscripted references, and references to builtin functions all have the same syntactic form, so a knowledge of the relevant declarations is necessary in order to distinguish them. An example of a rather complex function call is

```
A.B(3)(I,'NEXT')(X)
```

In this case, A.B(3) is an element of an array of structures (or a structure of arrays) containing an entry value. That entry value designates a procedure that expects two arguments -- in this case, I and the string constant 'NEXT' -- and itself returns an entry value. The entry value obtained from this second procedure is then applied to the argument X. A function expecting no arguments is called by using an empty argument list. Thus

```
NEXT_SUIT( )
```

would call the procedure NEXT_SUIT with no arguments.
If E is a procedure that itself returns an entry value, then

F(E)

indicates that F is to be called with an argument consisting of the entry value associated with E, while

F(E( ))

indicates that F is to be called with an argument obtained by calling E as a function of no arguments. This convention is somewhat different from the one used in Algol 60 and many of its descendants.

Function calls are discussed in more detail under "Procedures, Scopes and Environments" below.

Prefix and Infix Expressions

A *prefix expression* consists of an expression preceded by a prefix operator, while an *infix expression* consists of two expressions with an infix operator between them. When an expression contains a string of operators, the meaning is determined by the precedences of the operators. Those operators with highest precedence are applied first, then those of next highest precedence, etc. The infix operators, grouped by precedence from high to low, are:

```
                **
                *  /
                +  -
                ||
=   ¬=   <   >   <=   >=   ¬<   ¬>
                &
                |
```

Table 4 summarizes the meanings of the operators.

Table 4. PL/I Operators and Their Meanings

Infix Operators

**	exponentiation
*	multiplication
/	division
+	addition
-	subtraction
||	concatenation
=	equal
¬=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
¬<	not less
¬>	not greater

Prefix Operators

-	minus
+	plus
¬	not

When a sequence of adjacent operators, all of the same precedence, appears, the operators are applied from left to right except in the case of **, which is applied from right to left. Thus

```
A * B / C ** D ** E
```

is interpreted as

```
(A * B) / (C ** (D ** E))
```

Prefix operators are always applied first unless they conflict with the ** operator; in that case the ** is applied first, so that

```
- A ** 3
```

is interpreted as

```
- (A ** 3)
```

even though

```
- A * 3
```

is interpreted as

```
(- A) * 3
```

There are five arithmetic operators in PL/I:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

In order to apply any of the first four, the operands must first be converted to a common base, scale, and mode according to the following rules:

Base: binary if either operand binary, otherwise decimal

Scale: float if either operand float, otherwise fixed

Mode: complex if either operand complex, otherwise real

The operands need not have the same precision, however. The rules for the results of these operations assume that there is a maximum value N for the number-of-digits of the result. The "precision rules" then give the number-of-digits (and, for the case of fixed, the scale-factor) of the result. They are arranged so that digits on the right are never thrown away except in the case of division, where it cannot be avoided. If the operands are float, then the number-of-digits of the result is the maximum of the numbers-of-digits of the operands. Otherwise, assume that the two operands, which are necessarily fixed, have precision (p,q) and (r,s) respectively. The result precision (m,n) for the four operations is given by:

```
+,-     m = min(N, max(p-q,r-s) + max(q,s) + 1)
        n = max(q,s)
*       m = min(N,p+r+1)
        n = q+s
/       m = N
        n = N-p+q-s
```

Should the result value exceed the capacity of the result precision, the FIXEDOVERFLOW-condition is raised, indicating an error (see "ON-Units and ON-Statements" below).

The situation with regard to exponentiation is somewhat more complicated. Suppose that the formula x ** y is being

computed. If either x or y is float, the result is float, and the result precision is that of p. If x is real and fixed, and y is a small integer constant, then the result is also real and fixed, and the precision of the result is given by:

$$m = (p + 1) * y - 1$$

$$n = q * y$$

In any other case where both x and y are real, the result is real. If either x or y is complex, the result is complex. In certain cases, such as x real and fixed with a negative value and y not an integer, an error is indicated.

The comparison operators are:

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

The operators <= and ¬> are equivalent, as are the operators >= and ¬<; ¬> and ¬< are included mainly for intellectual compatibility with COBOL, which uses the phrases NOT GREATER and NOT LESS. All of the comparison operators return a one-bit value: '1'B if the comparison is satisfied, and '0'B if it is not. The equality and inequality comparisons can be applied to any type of data, although for most of the non-printable types both operands must have the same type. (The

only exception is that pointers can be compared to offsets.)

The comparison operators that test for inequality cannot be applied to complex arithmetic data or to nonprintable data, but they can be applied to real arithmetic data, to pictured data, and to strings. The meanings of these operators applied to arithmetic data are the usual ones; a numeric pictured datum is treated as the numeric value that it represents. When character strings of unequal length are compared, the shorter one is filled on the right with blanks to bring it to the same length as the longer one. The ordered comparisons are then done left to right on the basis of the implementation-defined collating sequence, which defines an order for the individual characters. For instance, the letters are ordered alphabetically and the digits numerically. Two character strings compare equal if they are identical after the shorter one has been blank-filled on the right. The ability to perform ordered comparisons of character strings is particularly useful in applications that involve sorting names. Similar rules apply to bit strings: If two bit strings are of unequal length, the shorter one is filled on the right with zero-bits prior to the comparison. If the strings differ, the comparison is done bit-by-bit from the left with the rule that a one-bit is greater than a zero-bit.

If the two operands of a comparision have different types, they are converted to a common type. If one has an arithmetic type, the other is converted to an arithmetic type. If neither has an arithmetic type, but one has a character type and the

other has a bit type, the bit-type operand is converted to character.

The concatenation operator, "||", is used to put two strings together. For instance, the value of

```
'AVER' || 'AGE'
```

is the string 'AVERAGE'. If one operand is a bit string and the other is a character string, the bit string is converted to a character string (as it is for comparisons).

There are also three logical operators:

&	and
|	or
¬	not

"&" and "|" are infix operators, while "¬" is a prefix operator. The operands of these operators are expected to be bit strings, so that if they have any other type, they are conerted to bit strings.

Builtin Functions

PL/I includes a large variety of builtin functions. A list of these, together with a brief explanation of what each one does, is given in Table 5. Some of the more important builtin functions will now be described.

The first group of builtin functions deals with strings. The descriptions of the functions will be given for character strings, but the definitions for bit strings are analogous.

The builtin function LENGTH(x) returns as its value the

actual length of the character string x. It is useful in two contexts: determining the length of a string passed as a parameter, and determining the current length of a varying string. In both of these cases, the declaration of the string does not provide enough information to determine the length.

The builtin function SUBSTR(x,y,z) is used to extract a portion of a string. x is the string, y is the position of the first character to be extracted, and z is the number of characters to be extracted. If z is omitted, all of the string starting with the character at position y is extracted. The null string is a possible result. SUBSTR can also be used on the left side of an assignment. For example, given the statements

```
DECLARE CHAR8 CHARACTER(8);
CHAR8 = 'TOM JONES';
SUBSTR(CHAR8,2,5) = 'IM HA';
```

the resulting value of CHAR8 is 'TIM HANES'. A builtin function used on the left side of an assignment in this way is called a *pseudovariable.*

The builtin function INDEX(x,y) finds the first position within the string x where the string y occurs. If y does not occur at all within x, the value of INDEX is 0. For example:

```
value of INDEX('SYNCOPATION', 'COP') = 4
value of INDEX('SYNCOPATION', 'COPE') = 0
```

The builtin function VERIFY(x,y) finds the first character in

Table 5. SUMMARY OF THE PL/I BUILTIN FUNCTIONS

In this table, descriptions of the various PL/I builtin functions are given. These descriptions are intended to indicate the intent of each function, and in some cases the principal restrictions on their arguments. In the descriptions of the functions, square brackets are used to indicate optional arguments.

1. ABS(x) - the absolute value of x. If x is complex, the value is its modulus.
2. ACOS(x) - the arc cosine of x. x must not be complex.
3. ADD(x,y,p,[q]) - the sum of x and y with precision (p,q) or (p,0) if the result is fixed, and with precision (p) if the result is float.
4. ADDR(x) - a pointer to the generation of x.
5. AFTER(sa,ca) - the portion of the string sa that follows the first occurrence of ca within sa. If ca does not occur within sa, the value is the null string.
6. ALLOCATION(x) - the number of generations of the controlled variable x that currently exist.
7. ASIN(x) - the arc sine of x. x must not be complex.
8. ATAN(y[,x])- the arc tangent of y/x if x is given, and of y otherwise.
9. ATAND(y[,x]) - the arc tangent in degrees of y/x if x is given, and of y otherwise. x and y must be real.
10. ATANH(x) - the hyperbolic arc tangent of x.

Table 5. Continued

11. BEFORE(*sa*,*ca*) - the portion of the string *sa* that precedes the first occurrence of *ca* within *sa*. If *ca* does not occur within *sa*, the value is the null string.
12. BINARY(*x*[,*p*[,*q*]]) - the result of converting *x* to binary, with precision determined by *p* and *q* if one or both is given.
13. BIT(*x*,[*le*]) - the result of converting *x* to bit, with length *le* if *le* is given.
14. BOOL(*x*,*y*,*ca*) - the boolean function of *x* and *y* whose truth table is specified by the four-bit value *ca*.
15. CEIL(*x*) - the least integer greater than or equal to *x*. *x* must be complex.
16. CHARACTER(*sa*,[*le*]) - the result of converting *sa* to character, with length *le* if *le* is given.
17. COLLATE( ) - the implementation-defined collating sequence, as a character string.
18. COMPLEX(*x*,*y*) - the complex number $x + i\ y$.
19. CONJG(*x*) - the complex conjugate of *x*.
20. COPY(*sa*,*le*) - the string consisting of *le* copies of *sa* concatenated together.
21. COS(*x*) - the cosine of *x*.
22. COSD(*x*) - the cosine of *x*, with *x* given in degrees. *x* must not be complex.
23. COSH(*x*) - the hyperbolic cosine of *x*.
24. DATE( ) - the current date, in the form *yymmdd*, where *yy* is the year, *mm* is the month, and *dd* the day.

25. DECAT(*sa*,*ca*,*pa*) - a portion of the string *sa*. *sa* is partitioned into three pieces by the first occurrence of *ca*. The three-bit string *pa* specifies which of the three pieces (before *ca*, *ca* itself, after *ca*) are to be concatenated to form the value of DECAT.

26. DECIMAL(*x*[,*p*[,*q*]]) - the result of converting *x* to decimal, with precision determined by *p* and *q* if one or both is given.

27. DIMENSION(*x*,*n*) - the number of elements in the *n*-th dimension of the array *x*, defined as HBOUND(*x*,*n*)-LBOUND(*x*,*n*) + 1.

28. DIVIDE(*x*,*y*,*p* [,*q*]) - the quotient of *x* and *y* with precision (*p*,*q*) or (*p*,0) if the result is fixed, and with precision (*p*) if it is float.

29. DOT(*x*,*y* [,*p*[,*q*]]) - the dot product of *x* and *y*, with precision determined by *p* and *q* if one or both is given.

30. EMPTY( ) - the empty area-value.

31. ERF(*x*) - the statistical error function of *x*.

32. ERFC(*x*) - the complement of the statistical error function of *x*.

33. EVERY(*x*) - the value '1'B if every bit in *x* is a one-bit, and '0'B otherwise. For this purpose, all scalar-elements of *x* are converted to bit.

34. EXP(*x*) - the exponential function of *x*.

35. FIXED(*x*, *p*[,*q*]) - the result of converting *x* to fixed, with precision determined by *p* and, if it is given, *q*.

36. FLOAT(*x*,*r*) - the result of converting *x* to float with precision (*r*).

37. FLOOR(x) - the greatest integer less than or equal to x. x must be real.

38. HBOUND(x,n) - the upper bound of the n-th dimension of the array x.

39. HIGH(le) - a string of le copies of the highest character in the collating sequence.

40. IMAG(x) - the imaginary part of the complex number x.

41. INDEX(sa,ca) - the position within the string sa of the first occurrence of the string ca. The value is 0 if ca does not occur within sa.

42. LBOUND(x,n) - the lower bound of the n-th subscript of the array x.

43. LENGTH(sa) - the length of the string sa.

44. LINENO(fn) - the current line number (within a page) of the print file fn.

45. LOG(x) - the natural logarithm of x.

46. LOG10(x) - the logarithm to the base 10 of x. x must not be complex.

47. LOG2(x) - the logarithm to the base 2 of x. x must not be complex.

48. LOW(le) - a string of le copies of the lowest character in the collating sequence.

49. MAX(x_1,x_2,...,x_n) - the largest of the numerical values of the x_i . The x_i must not be complex.

50. MIN(x_1,x_2,...,x_n) - the smallest of the numerical values of the x_i. The x_i must not be complex.

51. MOD(x,y) - the value of x modulo y. x and y must not be complex.

52. MULTIPLY(*x*,*y*,*p*[,*q*]) - the product of *x* and *y* with precision (*p*,*q*) or (*p*,0) if the result is fixed, and with precision (*p*) if it is float.

53. NULL( ) - the null pointer.

54. OFFSET(*pt*,*ar*) - the result of converting the pointer *pt* to an offset within the area *ar*.

55. ONCHAR( ) - the leftmost erroneous character within the current ONSOURCE-value. When the conversion-condition is raised, the current ONSOURCE-value is set to the string whose conversion was being attempted, and it retains this value during the execution of the associated on-unit.

56. ONCODE( ) - an implementation-defined integer indicating the nature of the on-condition associated with the current on-unit.

57. ONFIELD( ) - the contents of an erroneous field encountered during data-direct input, causing the name condition to be raised.

58. ONFILE( ) - the name of the file being processed when an input-output condition was raised.

59. ONKEY ( ) - the name of an erroneous key that caused the KEY-condition to be raised during record input-output.

60. ONLOC( ) - the name of the procedure entry-point active when the current on-unit was raised.

61. ONSOURCE( ) - the current onsource-value. When the CONVERSION-condition is raised, the current ONSOURCE-value is set to the string whose conversion was being attempted,

62. PAGENO(*fx*) - the number of the current page within the print file *fx*.

63. POINTER(*ofe*,*ar*)- the result of converting the offset *ofe* within the area *ar* to a pointer.

64. PRECISION(*x*,*p*[,*q*]) - the result of converting *x* to precision (*p*,*q*) or (*p*,0) if *x* is fixed, and to precision (*p*) if *x* is float.

65. PROD(*x*) - the product of all the elements of the array *x*.

66. REAL (*x*) - the real part of the complex number *x*.

67. REVERSE(*sa*) - the bits or characters of the string *sa* taken in reverse order.

68. ROUND(*x*,*n*) - the result of rounding up the numerical value of *x*. If *x* is fixed, the result has a scale-factor of *n*; otherwise the result has a number-of-digits of *n*. *n* must be an integer constant.

69. SIGN(*x*) - the value +1, 0, or -1 according to whether *x* is positive, zero or negative.

70. SIN(*x*) - the sine of *x*.

71. SIND(*x*) - the sine of *x*, with *x* given in degrees. *x* must not be complex.

72. SINH(*x*) - the hyperbolic sine of *x*.

73. SOME(*x*) - the value '1'B if at least one bit in *x* is a one-bit, and '0'B otherwise. For this purpose, all scalar-elements of *x* are converted to bit.

74. SQRT(*x*) - the square root of *x*.

75. STRING(*sa*) - the result of concatenating together the scalar-elements of *sa* after converting them to bit.

76. SUBSTR(*sa*,*st*[,*le*]) - the substring of *sa* consisting of *le* characters or bits of *sa* beginning with the *st*-th one.

Table 5. Continued

If *le* is omitted, the substring consists of the characters or bits from the *st*-th one to the last.

77. SUBTRACT(*x*,*y*,*p*[,*q*]) - the difference of *x* and *y* with precision (*p*,*q*) or (*p*,0) if the result is fixed, and with precision (*p*) if it is float.

78. SUM(*x*) - the sum of all the elements of the array *x*.

79. TAN(*x*) - the tangent of *x*.

80. TAND(*x*) - the tangent of *x*, with *x* given in degrees. *x* must not be complex.

81. TANH(*x*) - the hyperbolic tangent of *x*.

82. TIME( ) - the current time, in the form *hhmmss* where *hh* gives the hour, *mm* gives the minute, and *ss...s* gives the second carried to an implementation-defined number of fractional decimal places.

83. TRANSLATE(*sa*,*ra*[,*pa*]) - the result of replacing, within *sa*, each character of *pa* by the corresponding character of *ra*. If *pa* is omitted, it is taken to be the collating sequence.

84. TRUNC(*x*) - the result of truncating *x* to the nearest integer in the direction of zero. *x* must not be complex.

85. UNSPEC(*x*) - the internal representation of *x*, as a bit string.

86. VALID(*xx*) - the value '1'B if the current value of the pictured variable *xx* conforms to the picture and '0'B otherwise.

87. VERIFY(*sa*,*sb*) - the position within the string *sa* of the first character or bit of SA that does not appear within *sb*. *sb* thus behaves as a set rather than a sequence. If all characters or bits of *sa* occur within *sb*, the value of VERIFY is 0.

the string x that is not a character of the string y. If all the characters of x appear in y, then the value of VERIFY is 0. For example:

value of VERIFY('CABDRIVER','ABCDE') = 5

value of VERIFY('CEDE','ABCDE') = 0

The builtin function REVERSE(x) reverses its argument, so for example:

value of REVERSE('GALLOP') = 'POLLAG'

This function is useful in right-to-left scanning; the string to be scanned is reversed and then scanned left to right. The builtin function COLLATE( ) (the "( )" indicates that COLLATE is a function of no arguments) has as its value the implementation-defined collating sequence, i.e., the string consisting of all acceptable characters ordered from least to greatest. The builtin function COPY(x,n) creates n copies of the string x. For example,

value of COPY('CHA',3) = 'CHACHACHA'

The arithmetic builtin functions enable the user to control the attributes of arithmetic results. These fall into two groups. First, there are builtin functions ADD, SUBTRACT, MULTIPLY, and DIVIDE that behave like the corresponding infix operators, except that the precision of the result is explicitly specified. For example,

MULTIPLY(M1,M2,5,3)

produces a result whose precision is (5,3), and whose remaining attributes are determined by the attributes of M1 and M2.

Second, there are builtin functions FIXED, FLOAT, BINARY, and DECIMAL that convert their argument to the specified attribute with the specified precision. For instance,

DECIMAL(M1,4,2)

converts M1 to fixed decimal with precision (4,2); the mode of the result is the mode of M1.

The conversion builtin functions FIXED, FLOAT, BINARY, and DECIMAL can be used not only to convert among arithmetic types but also to convert from the string types. The rules for the conversion are discussed under "Type Conversion" below. There are further conversion functions REAL(x) which converts x to real type (and for a complex number, takes its real part); IMAG(x), which takes the imaginary part of the complex number x (and yields 0 if x is not a complex number); and COMPLEX(x,y), which converts x and y to a common real type and then forms the complex number $x + iy$. For conversion to string types, the builtin functions CHARACTER(x,n) and BIT(x,n) can be used. CHARACTER(x,n) first converts x to character type and then adjusts the length of the result to n either by truncating on the right or by filling on the right with blanks. BIT behaves similarly, either truncating or filling with zero-bits. The second argument of either of these functions may be omitted, in which case no truncation or filling is done.

The mathematical builtin functions included in PL I are listed in Table 6. With the exceptions indicated, they can accept arguments of any arithmetic type, including complex types. For some mathematical functions there is more than one

possible range for the result value, and the choice of principal value is specified in the table.

The function ATAN (arctangent) can accept either one or two arguments. The two-argument version is useful in converting rectangular coordinates to polar coordinates. If the rectangular coordinates are given by the pair (x,y), then ATAN(x,y) gives the corresponding polar angle in the range from $-\pi$ to $+\pi$. Since the value of the tangent function repeats every $\pi/2$ radians, the sign of y is needed to determine the correct value.

A number of the builtin functions fall into no particular category. The builtin function SUM(x) accepts an array as argument, and returns as value the sum of all the elements of the array. The builtin function PROD, for "product", behaves similarly. The builtin function DOT(x,y) expects its arguments to be one-dimensional arrays both having the same bounds; it takes the mathematical dot product of x and y. The builtin function BOOL(x,y,z) takes as arguments two bit strings x and y of arbitrary length, and a third bit string z of length 4. z determines a boolean function that is applied to x and y. If z is the sequence $b_1b_2b_3b_4$, then the function is defined by:

bit of x	bit of y	result
0	0	b_1
0	1	b_2
1	0	b_3
1	1	b_4

Table 6. Mathematical Builtin Functions

PL/I Name	Mathematical Description	Complex Arguments?	Constraints on Result R (Principal Value)
ABS	absolute value	yes	$R \geq 0$
ACON	arc cosine	no	$0 \leq R \leq \pi$
ASIN	arc sine	no	$-\pi/2 \leq R \leq \pi/2$
$ATAN_1$	arc tangent (one argument)	yes	$-\pi/2 < R < \pi/2$ (real argument)
$ATAN_2$	arc tangent of quotient (two arguments)		$-\pi < Re(R) < \pi$ (complex argument)
$ATAND_1$	arc tangent in degrees (one argument)	no	$-90 < R < 90$
$ATAND_2$	arc tangent of quotient in degrees (2 arguments)	no	$-180 < R \leq 180$
ATANH	hyperbolic arctangent	yes	
COS	cosine	yes	
COSD	cosine in degrees	no	
COSH	hyperbolic cosine	yes	
ERF	error function	no	
ERFC	complement of error function	no	
EXP	exponential	yes	
LOG	natural logarithm	yes	$-\pi < Im(R) \leq \pi$
LOG2	base 2 logarithm	no	
LOG10	base 10 logarithm	no	
SIN	sine	yes	
SIND	sine in degrees	no	
SINH	hyperbolic sine	yes	
SQRT	square root	yes	$Re(R) > 0$ or $Re(R)=0$ and $Im(R) \geq 0$
TAN	tangent	yes	
TAND	tangent in degrees	no	
TANH	hyperbolic tangent	yes	

The builtin function VALID can be used to check the validity of pictured data, i.e., to ensure that the value stored in a pictured variable fits the description given by the picture. VALID(x) returns '1'B if the pictured variable x contains a valid value, and '0'B otherwise. Invalid values can arise since an arbitrary character string can be read into or assigned to a pictured varible, and ordinarily no validity check is made at the time of reading or assignment.

The builtin functions EVERY and SOME are useful in testing properties of aggregates. EVERY(x) returns '1'B if its argument (after conversion to bit type, if necessary) consists entirely of one-bits, and '0'B otherwise. SOME(x), on the other hand, returns '1'B if its argument contains at least one one-bit, and '0'B otherwise. For example, if A is an array of arithmetic type, then the expression A > 0 will be an array with a one-bit in each position i where A(i) > 0. Therefore EVERY(A>0) will return '1'B if all elements of A are greater than 0, while SOME(A>0) will return '1'B if at least one element of A is greater than 0.

Type Conversion

In PL/I it is possible to convert from any printable type to any other, although for certain values the conversion may be illegal. Conversions may be invoked either explicitly, using builtin functions such as FLOAT or CHARACTER, or implicitly in contexts such as operands of operators or arguments of functions. For instance, the concatenation

operator requires that its operands be strings of the same type (bit or character), so that the operands must be converted appropriately -- even if they are of arithmetic type. The text of a procedure defines the types of its parameters, and if the arguments of a procedure do not already have the expected types, they too must be converted. In fact, PL/I provides implicit conversions in almost every context where conversion is possible.

The conversions among arithmetic types generally follow the principle of preserving the meaning of the converted value. For example, the result of converting the fixed value 7.3 to complex float decimal with precision (8) is .73000000E+01+0I. In conversion to a fixed type when digits must be dropped, the result value is obtained by truncating towards 0, although in certain unusual cases an implementation may produce a slightly different result. When converting from real to complex an imaginary part of 0 is added, while when converting from complex to real the imaginary part is dropped.

The conversion between bit and character is straightforward; zero-bits correspond to the character "0", and one-bits correspond to the character "1". A character string to be converted to bit type must consist entirely of these two characters, or an error is signalled -- specifically, the CONVERSION-condition. It is possible for the programmer to modify the converted value so as to correct the error (see "Categorization of the ON-Conditions" below).

The most complicated conversions are those between the string types and the arithmetic types. A character string is converted to a number by treating the string as the representation of a number. Thus, given the statements

```
DECLARE NUMV FIXED DECIMAL(5,2);
NUMV = 'ØØ2.13E1Ø';
```

the string 'ØØ2.13E1Ø' is converted first to the float value that it represents, and then to the fixed decimal value 21.30. The blanks surrounding the number are always permissible. An all-blank value converts to zero. If the character string does not represent a valid number, then the CONVERSION-condition is signalled. As in the case of conversion from character to bit, it is possible for the programmer to correct the error.

Conversion from a number to a character string yields, in effect, the result of printing the number. Ordinarily that result includes leading blanks. For instance, the effect of

```
DECLARE NUMV FIXED DECIMAL(4);
DECLARE CONV_RESULT CHARACTER(20) VARYING;
NUMV = 17;
CONV_RESULT = NUMV;
```

is to assign the string 'ØØØØØ17' to NUMV. In most cases the length of the resulting string is the number-of-digits after conversion (if necessary) to fixed decimal, plus three. Three spare positions are needed in order to accommodate a possible sign, a possible decimal point, and a possible leading zero.

Conversion from an arithmetic value to a bit string is accomplished by first converting the arithmetic value to real fixed binary and then converting the integer part of the value to the corresponding bit string. For instance, converting the value 12.6 to a bit string yields '001100'B, with an intermediate conversion from fixed decimal with precision (3,1) to fixed binary with precision (10,4). (The rules for obtaining the intermediate precision are somewhat complicated, but it can be seen that two digits to the left of the decimal point may require as many as six nonzero bits to represent their value.) Conversion from a bit string to an arithmetic type is accomplished by treating the bit string as a binary number, and then converting from that number to the desired type.

It is also possible to convert from pointer to offset, or vice versa, provided that an area is given. Thus if AR is an area and P is a pointer, the expression OFFSET(P,AR) gives the result of converting P to an offset relative to A. Similarly, if OFS is an offset, POINTER(OFS,A) gives the result of converting OFS to a pointer relative to A. A pointer can be declared with an area-reference, as in

```
DECLARE AR2 AREA;
DECLARE O2 OFFSET(AR2);
```

In this case, O2 can implicitly be converted to a pointer, and the area AR2 is used in the conversion.

Promotion

The PL/I operators, and many of the builtin functions also, can be applied to aggregates as well as to scalars. When two aggregates of the same organization, i.e., two structures with equal numbers of components or two arrays of the same dimensionality, are used as the operands of an operator, the result also has that organization, and the result is formed by combining corresponding components of the operands. For example, in

```
DECLARE A(3,4) FIXED BINARY;
DECLARE B(3,4) FIXED BINARY;
DECLARE C(3,4) FIXED BINARY;
A = B + C;
```

the assignment to A is accomplished by adding B(1,1) to C(1,1), B(1,2) to C(1,2), etc., to form a new array of sums with dimensionality (3,4). The array of sums is then copied into A. In certain peculiar cases the temporary array containing the sum is actually needed, and it does not suffice simply to add the elements of B and C one by one and place the result directly in A.

It is also possible to combine scalars with structures, scalars with arrays, and structures with arrays. However, it is not possible to combine structures having different numbers of members, or arrays having different dimensionalities. A scalar is combined with a structure by promoting it to a similar structure, all of whose members have the same value as the scalar. Similarly, a scalar is combined with an array by

promoting the scalar to a similar array. The case of combining a structure with an array is more complicated; first the original structure is promoted to an array of structures, and later each element of the original array is promoted to a structure. A simple instance of promotion is given by the expression A+1, where A is an array of arithmetic type. The value of this expression is obtained by creating an array of 1's, having the same dimensionality as A, and then adding this new array to A, element by element. The effect is just the same as adding 1 to each element of A.

The Assignment-Statement

The assignment-statement contains a left side, which is a list of *targets*, and a right side, which is an expression. Each of the targets designates a location capable of receiving a value. The statement is executed by evaluating the expression and then storing its value, after appropriate conversion, into the location designated by each target, in order from left to right. For instance, the assignment-statement

```
A, B(I), C=1;
```

causes 1 to be stored into A, B(I), and C. The targets of an assignment-statement may be variables or pseudo-variables. For instance, the assignment-statement

```
SUBSTR(TEXT,I,LEN) = WORD;
```

stores the value of WORD into the indicated substring of TEXT (after adjusting the size of the value to be LEN). Similarly,

```
IMAG(Z) = SIN(X);
```

causes the imaginary part of the (necessarily) complex variable Z to be set to the value of SIN(X), while the real part of Z is left undisturbed.

Since the type of the value obtained from the right side of an assignment-statement may disagree with the type of a target, promotion or conversion, or both, may be necessary. If the target is a scalar, then the usual rules for scalar conversion are applied; the type of the target defines the type to which the value must be converted. If the target is a structure or array, then the value must be promoted to the type of that structure or array, by replicating elements as necessary. Following the promotion, element-by-element scalar conversion may be necessary. For instance, in the example

```
DECLARE HVAR(30) FLOAT BINARY;
HVAR = 0;
```

the scalar value 0 is promoted to an array of 30 fixed zeros. Each of these is then converted to an appropriate float zero, and assigned to the corresponding element of the array. *

A variation on the assignment-statement, called *by-name assignment*, can be used to assign elements from one structure to another according to the names of the elements rather than according to their positions in the structure. For instance, given the declarations

* In actual practice, the conversion is usually done before rather than after the promotion. The result is the same.

```
DECLARE
   1  RED,
      2  BLUE,
      2  GREEN,
         3  ORANGE,
         3  WHITE,
      2  BLACK,
      2  GRAY;
DECLARE
   1  VIOLET,
      2  BLACK,
      2  WHITE,
      2  GREEN,
         3  ORANGE,
         3  VIOLET,
      2  TAN
      2  BLUE;
```

the effect of the assignment-statement

```
RED = VIOLET, BY NAME;
```

is to perform theindividual assignments

```
RED.BLUE = VIOLET.BLUE;
RED.GREEN.ORANGE = VIOLET.GREEN.ORANGE;
RED.BLACK = VIOLET.BLACK;
```

Those members not in common between the two structures are ignored. By-name assignment can be extended to accommodate expressions that involve structures.

STORAGE TYPES

PL/I provides a variety of ways to manage the storage of variables. Each variable has a *storage type*, which can be either *parameter*, *defined*, or a storage class. The parameter and defined storage types indicate that the variable is an alias, i.e., an alternate name, for storage that has already been obtained by other means. The *storage classes* provide different ways of allocating and freeing storage; the storage classes are *static*, *automatic*, *controlled*, and *based*. The storage type of a variable is determined by its declaration after all defaulting of declarations has been done; in most cases the default is the automatic type. The storage used to hold the value of a variable is called a *generation*. A generation can exist even though it is not currently associated with any variable.

Static Storage

The static storage class is the simplest one. When a variable is declared to be static, its generation is allocated at the start of program execution and remains allocated throughout program execution. When a static variable is declared within a procedure, the values of that variable are kept from one call of the procedure to the next. Even in the case of a recursive procedure, there is just one copy of the variable, and that copy is available at all levels of recursion. Static storage is much like the standard form of storage in FORTRAN.

Automatic Storage

Storage for an automatic variable is allocated on entrance to the block where the variable is declared, and freed on exit from that block. Whenever the block is entered, a fresh generation is obtained for the variable. In practice it sometimes happens that values of automatic variables are retained from one block entrance to the next, but this behavior is not anything that the programmer can rely upon. When an automatic variable is declared within a recursive procedure, a new generation is created for each level of recursion, and remains associated with the variable at that recursion level until the recursion level is terminated. Automatic storage resembles the ordinary local storage of Algol.

Controlled Storage

Controlled storage is explicitly allocated and freed by the programmer using the ALLOCATE-statement and the FREE-statement. Each time the variable is allocated, a new generation for it is created and placed on a pushdown stack; each time the variable is freed, the generation at the top of the stack is destroyed. There is one such stack for each controlled variable, and the current value of the variable is always obtained from the generation at the top of the stack. In other words, the generations follow a last-in-first-out rule.

The values of string lengths and array dimensions in the declaration of a controlled variable can be given by expressions.

The expressions are evaluated when a new generation is allocated, and so the different generations need not all have the same sizes. For example, suppose that we are given the statements:

```
DECLARE N FIXED;
DECLARE CONTV CHARACTER(N) CONTROLLED;
N = 5;
ALLOCATE CONTV;
CONTV = 'FIRST';
N = 7;
ALLOCATE CONTV;
CONTV = 'SECOND';
```

If CONTV has not been previuosly allocated, these statements will create a stack consisting of two generations. The generation at the top of the stack will have length 7 and value 'SECONDb' (the assignment adds a blank on the right), while the other generation will have length 5 and value 'FIRST'. Thus the current value of CONTV will be 'SECONDb'. If the statements

```
FREE CONTV;
PUT LIST(CONTV);
```

are executed, then the generation at the top of the stack will be destroyed and CONTV will refer to the first generation. Consequently the PUT-statement will cause FIRST to be printed.

Based Storage

Based variables are useful in creating linked data structures, and also have applications in record input-output. A based variable does not have any storage of its own; instead, the declaration acts as a template and describes a generation of storage. In order to use the variable to refer to a

particular generation of storage, a pointer to that generation must also be provided. The pointer and the based variable, taken together, constitute a *based reference*. In many cases, the pointer is given implicitly rather than explicitly.

An example of a declaration of a based variable and a pointer is

```
DECLARE
   1  ARRAY_ELT BASED,
      2  ARRAY(10) FLOAT,
      2  NEXT_ELT POINTER;
DECLARE AP POINTER;
```

ARRAY_ELT describes a generation of storage, namely, a structure containing a float array and a pointer. The based reference

```
A -> ARRAY(4)
```

designates a particular element within the ARRAY_ELT structure pointed at by the pointer AP, and if AP does not point at such a structure the reference is invalid. PL/I does not provide any mechanism for checking that a pointer is indeed pointing at a generation of the correct type, and so it is entirely the programmer's responsibility. The errors that result when a pointer points at an object of the wrong type can often be extremely difficult to track down.

The statement

```
ALLOCATE ARRAY_ELT SET(AP);
```

causes a generation matching the type of ARRAY_ELT to be created and also causes the pointer AP to point at that generation. Thus, after this ALLOCATE-statement has been executed,

a reference to AP -> ARRAY(4) will be valid. If subsequently the statement

```
FREE AP -> ARRAY_ELT;
```

is executed (and the value of AP has not been changed in the meantime), the generation pointed at by AP will be destroyed, and subsequent references to that generation will be meaningless.

In this example, the structure includes not only the array but also a pointer. That pointer can be used to form a list of arrays, each one pointing to its successor. An element is added to the head of the list by allocating it and setting its NEXT_ELT component to the previous list head. Similarly, the head of the list is deleted by setting the new list head to the NEXT_ELT component of the old list head and then freeing the old list head. One of the main uses of based variables and pointers in PL/I is constructing lists such as this one. In order to end a list, a special null pointer is needed, and that pointer is provided by the NULL builtin function.

It is convenient to have to write a pointer with every based reference. Therefore it is possible to declare an implicit pointer in the declaration of a based variable, e.g.,

```
DECLARE BFIX BASED(BFP) FIXED;
DECLARE BFP POINTER;
```

A reference to BFIX by itself is taken to mean BFP -> BFIX. Moreover, the statement

```
ALLOCATE BFIX;
```

is equivalent to

```
ALLOCATE BFIX SET(BFP);
```

and the statement

```
FREE BFIX;
```

is equivalent to

```
FREE BFP -> BFIX;
```

The template given by a based variable can be applied to storage of types other than based. In order to obtain a pointer to a generation, the ADDR builtin function is used. ADDR(v) yields a pointer to the generation specified by v. As an example, the statements

```
DECLARE BCOMP FLOAT COMPLEX BASED;
DECLARE SCOMP FLOAT COMPLEX STATIC;
ADDR(SCOMP) -> BCOMP = 2E0 + 3E0I;
```

cause the static variable SCOMP to be set to the value 2E0+3E0I.

The Refer-Option

The string lengths and array bounds of a based variable can be specified by expressions as well as by constants. For example, the declaration

```
DECLARE BCS CHARACTER(M) BASED;
```

indicates that the length of BCS is given by the current value of M. When BCS is allocated, the generation that is created will have a length given by the current value of M, and when reference is made to BCS, the value of M must agree with the

length of the string in the generation referred to. If a number of generations, all corresponding to BCS, exist, it may be difficult to ensure that the current value of M is correct, since the generations may have different string lengths. In order to deal with this difficulty, PL/I allows the string length to be specified along with the string itself; both the string and the length are stored in a single structure, sometimes called a *self-defining* structure. For instance, the structure

```
DECLARE
   1  STRING_STRUC BASED(STP),
      2  LEN FIXED,
      2  NEXT POINTER,
      2  STRING CHARACTER(LEN1 REFER(LEN));
DECLARE STP POINTER;
```

could be used to create a list of strings, each having a different length. When one of these structures is allocated, the length of the string is obtained as the current value of LEN1, and at the same time the current value of LEN1 is automatically stored within the LEN component of the newly created generation. When one of these structures is referenced, the length of STRING is obtained from the LEN component of that structure. Both LEN and LEN1 are needed, for the following reason. Were LEN used without the so-called *refer-option*, the allocation size would be taken from STP->LEN prior to the allocation, which would be either undefined or the string length of a previously allocated generation. On the other hand, were LEN1 used, it would then be necessary to reset it to LEN before referencing STRING, since otherwise the length

of STRING would not be correct.

A based variable may contain any number of refer-options. These can be used to specify upper or lower array bounds as well as string lengths and area sizes.

Left-to-Right Correspondence

It is often necessary to create data structures in which the elements do not all have the same type, as in the following example:

```
DECLARE
   1 FLOAT_ELEMENT BASED(ELPTR),
     2 ELTYPE FIXED,    /* 1 FOR FLOAT */
     2 NEXT POINTER,
     2 VALUE FLOAT;

DECLARE
   1 FIXED_ELEMENT BASED(ELPTR),
     2 ELTYPE FIXED,    /* 2 FOR FIXED */
     2 NEXT POINTER,
     2 VALUE FIXED;

DECLARE
   1 CHAR_ELEMENT BASED(ELPTR),
     2 ELTYPE FIXED,    /* 3 FOR CHARACTER */
     2 NEXT POINTER,
     2 VALUE CHARACTER(24);

DECLARE ELPTR POINTER;
```

A list can be formed containing elements of all three kinds, storing a type code in ELTYPE in order to distinguish among them. In order to reference an element, it is necessary to specify either FLOAT_ELEMENT, FIXED_ELEMENT, or CHAR_ELEMENT even before the type of the element is known, since a reference to ELTYPE by itself is syntactically ambiguous. Therefore under certain conditions PL/I allows a reference to a component

of a based structure even when the variable in the reference does not agree with the generation being referenced. The primary condition is that the generation and the variable must agree up to that component, although there are further detailed requirements that are beyond the scope of this article. Based references satisfying this constraint are said to be in *left-to-right correspondence*, since they agree reading from left to right. Thus it is permissible to use FLOAT_ELEMENT.ELTYPE to refer to, and therefore to test, the type code stored in any one of the three kinds of elements. Even if ELPTR is pointing at a generation having the type of CHAR_ELEMENT, the ELTYPE component of that generation can be referenced using FLOAT_ELEMENT.ELTYPE. Since ELTYPE is the first component of each element, the elements necessarily agree up to that component. Moreover, the NEXT components of the three kinds of elements can be referenced interchangeably since in each kind of element NEXT has type pointer and is preceded by an element having type fixed (with the remaining attributes defaulted identically in all cases).

Allocation in Areas

A based variable can be allocated in a specified area, as in the following example:

```
DECLARE BV FIXED BASED(P);
DECLARE A AREA(200);
ALLOCATE BV IN(A);
```

Since BV has been allocated in A, the OFFSET builtin function

can be used to convert P into an offset relative to A, as given by

```
OFFSET(P,A)
```

The allocation can assign a value directly to an offset variable, as in the example

```
DECLARE OFS OFFSET(A);
DECLARE BV1 BASED(OFS);
ALLOCATE BV1 IN(A);
```

Since BV1 is based on OFS, the offset of BV1 relative to A is assigned to OFS when the ALLOCATE-statement is executed.

Parameter Storage

A variable acquires the parameter storage type by virtue of its appearance in a parameter list of either a PROCEDURE-statement or an ENTRY-statement. The PARAMETER attribute can, but need not, be declared for a parameter; it is invalid to use that attribute for any other kind of variable. A parameter describes a generation of storage passed as an argument to the procedure that declares the parameter. Thus, allocation and freeing of the parameter is the responsibility of the procedure's caller. Since a parameter is allocated before the procedure declaring it is entered, the procedure itself cannot specify an initial value for the parameter. See "Arguments and Parameters" below for further information about parameters.

Defined Storage

The defined storage type, like the parameter storage type, is an alias. The declaration of a defined variable specifies a *base item*, which is a portion (or possibly all) of some other variable. The defined variable provides another way of referencing part or all of the storage occupied by the base item. The base item can be part of a variable having any storage type other than defined or based, and so circular defining is excluded.

There are three kinds of defining: *simple-defining*, *isub-defining*, and *overlay-defining*. The sort of defining that is in effect is determined by the relation between the defined variable and the base variable. Since defined variables are aliases, they are not allocated nor freed, nor are initial values specified for them.

An example of simple-defining is

```
DECLARE A(5,8) FIXED;
DECLARE ADEF(2:4) DEFINED(A(1,*));
```

ADEF is defined to consist of the elements A(1,2), A(1,3), and A(1,4). For simple-defining to be in effect, the attributes of the defined variable must agree with those of the base item, except that the array bounds of the defined variable may be more restrictive than the corresponding bounds of the base item. A major use of simple-defining is to specify portions of arrays that are to be passed as arguments to procedures.

Isub-defining is in effect when the base item contains special subscripts, known as *isubs*. These subscripts have the form 1SUB, 2SUB, etc. An example of isub-defining is

```
DECLARE A(10,10) FIXED;
DECLARE ADEF(9,8) DEFINED(A(1SUB+1,2SUB+2));
```

A reference to an element of ADEF is translated into a reference to an element of A by substituting the first subscript for 1SUB and the second subscript for 2SUB. For instance, ADEF(8,6) refers to A(9,8). The defined array need not have the same dimensionality as the base item. For example, in

```
DECLARE B(30,30) FLOAT;
DECLARE BDIAG(30) DEFINED(B(1SUB,1SUB));
```

the one-dimensional array BDIAG consists of the diagonal elements of the array B, while in

```
DECLARE C(15) POINTER;
DECLARE C2(5,3) DEFINED(C(3*(1SUB-1)+2SUB));
```

the two-dimensional array C2 is defined onto the one-dimensional array C.

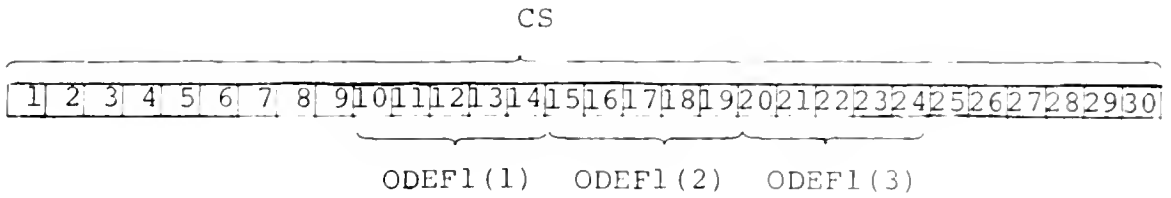
Overlay-defining is used in order to apply different descriptions to strings. For the purposes of overlay-defining, character data and pictures are together considered as *character-class* data, while bit strings are considered as *bit-class* data. An example of overlay-defining is:

```
DECLARE CS CHARACTER(30);
DECLARE ODEF1(3) CHARACTER(5) POSITION(10) DEFINED(CS);
DECLARE
     1  ODEF2 DEFINED(CS),
        2  OCS1 CHARACTER(14),
        2  OCS2 PICTURE '$$$V.$$';  /* 6 CHARACTER POSITIONS*/
```

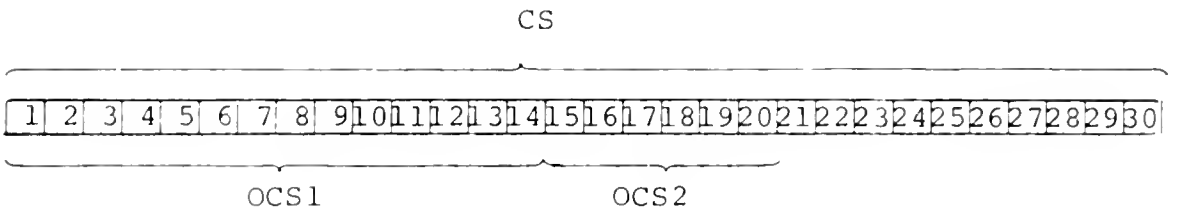
The relationship between CS and ODEF1, and between CS and ODEF2, is illustrated in Figure 3. The POSITION attribute in the declaration of ODEF1 indicates that the character sequence comprising ODEF1 starts at character 10 of CS. ODEF(1) consists of characters 10-14 of CS, ODEF1(2) of characters 15-19 of CS, and ODEF1(3) of characters 20-24 of CS. The treatment of ODEF2 is similar. In overlay-defining both the defined variable and the base item must consist entirely of unaligned data (see "Alignment" below) of the same class, but a string can be overlaid onto an array as well as the other way round.

Alignment

The declaration of a variable can specify an *alignment*, either ALIGNED or UNALIGNED. An aligned variable is stored so as to favor speed of access over space; typically, storage for an aligned variable is placed at a word boundary or other natural demarcation for the machine at hand. An unaligned variable is stored so as to favor space over speed of access, and is arranged in storage so as to minimize unused space. The default alignment for nonvarying strings and for pictures is unaligned; for everything else it is aligned.



(a) Overlaying ODEF1 onto CS



(b) Overlaying ODEF2 onto CS

Figure 3. Example of Overlay-Defining.

In most situations, the alignment of a variable has no effect on its behavior. The exception is that aggregates composed of unaligned strings and pictures are stored with all their components adjacent, i.e., as a sequence of adjacent characters (or bits, in the case of unaligned bit strings). The sequence can then be used as a base item for overlay-defining.

Initialization

It is possible to specify an *initialization* for a variable, as long as its storage type is not an alias, i.e., is neither parameter nor defined. The initialization is specified using the INITIAL-attribute. For example, in the declaration

```
DECLARE A(40) FIXED INITIAL((40)0);
```

the array A is initialized to all zeroes. The initialization can be specified by a single item, by a repeated item, or by a repeated list, which can itself contain items of these types. Nesting to any depth is permitted. Thus

```
DECLARE B(20) FIXED INITIAL(2,3,(5)4,(3)(-1,-2));
```

causes the first 13 elements of B to be initialized to the sequence

```
2 3 4 4 4 4 4 -1 -2 -1 -2 -1 -2
```

For a multidimensional array, initializations are performed with the last subscript varying most rapidly. Thus

```
DECLARE C(3,2) FIXED INITIAL(1,2,3,4,5,6);
```

causes the initializations

```
C(1,1) = 1     C(1,2) = 2
C(2,1) = 3     C(2,2) = 4
C(3,1) = 5     C(3,2) = 6
```

Initialization always takes place at the time of allocation. Thus, for static variables, the initialization is performed at the start of program execution. For automatic variables, it is performed at each entrance to the declaring block. For controlled and based variables, it is performed when an ALLOCATE-statement for the variable is executed, and is applied to the newly allocated generation. Parameters and defined variables cannot be initialized with the INITIAL-attribute.

Textually, a *procedure* is a body of code, delimited by a PROCEDURE-statement at the beginning and an END-statement at the end. Associated with the procedure are one or more *entry points*, each of which provides a way of invoking some portion of the code contained within the procedure. The entry points are defined by the PROCEDURE-statement, as well as by any ENTRY-statements that appear within the procedure. The characteristics of an entry point include its name, the number and types of its parameters, and the type of its returned value, if any. Each entry point, in turn, defines an entry constant.

A procedure is called either by means of a function reference within an expression or by means of a subroutine reference within a CALL-statement. The procedure call itself consists of an entry-valued reference and an argument list, possibly empty. For instance, the procedure call F(X+3,'INVALID') has an entry-valued reference F and an argument list consisting of the two arguments X+3 and 'INVALID'. The value of F must be an entry point of the procedure being called. Usually F is just the name of the procedure, but F could also be, for instance, an entry variable. An empty argument list for a function reference must be indicated explicitly by ( ). If an entry point returns a value, then it must be called by a function reference; otherwise it must be called by a subroutine reference.

An example of a procedure definition is:

```
P1:
P2:   PROCEDURE(QVAL,SIZE) RETURNS(FLOAT BINARY);
      DECLARE QVAL FLOAT BINARY;
      DECLARE SIZE FIXED DECIMAL(4);
      DECLARE J FIXED BINARY;
      DECLARE TOTAL FLOAT BINARY INITIAL(0);
      DO J = 1 TO SIZE;
         TOTAL = TOTAL + F(QVAL,J);
         END;
      RETURN(TOTAL);
P3:   ENTRY(RVAL,RES,SIZE);
      DECLARE RVAL FLOAT BINARY;
      DO J = 1 TO SIZE;
         RES = RES + F(RVAL,J);
         END;
      RETURN;
      END P1;
```

This procedure has three entry points. P1 and P2 are synonymous (but do not compare equal), and are entry constants designating the entry point at the PROCEDURE-statement. Since that entry point returns a value (with attributes FLOAT BINARY), P1 and P2 can only be called as function references, i.e., as components of an expression. P3 is the entry constant naming the entry point starting at the ENTRY-statement. It does not return a value, and so P3 can only be invoked from a CALL-statement, e.g., by

```
CALL P3(A(2),B(2),22);
```

P2 has two parameters, namely, QVAL and SIZE, while P3 has three parameters, namely, RVAL, RES, and SIZE. As this example shows, the entry points need not have the same parameters, and if any parameters are in common, they need not appear in the same position. It is invalid to reference a parameter not associated

with the entry point actually used to enter a procedure. For instance, it is invalid to reference RVAL if the procedure is entered through P1 or P2.

The RETURN-Statement

The RETURN-statement is used to end execution of a procedure. It may have either the form

```
RETURN(expr);
```

or the form

```
RETURN;
```

If the procedure is called by a function reference, then the RETURN-statement must contain an expression. Conversely, if the procedure is called by a subroutine reference, then the RETURN-statement must not contain an expression.

When a RETURN-statement containing an expression is executed, the expression is evaluated. The value of the expression is then taken as the value of the function reference that called the procedure. If necessary, the value of the expression is converted to the type specified in the RETURNS-clause of the entry point where the procedure was entered. A procedure can return an aggregate as well as a scalar. Moreover, the returned type may have asterisks in its specification, e.g.,

```
RETURNS(CHARACTER(*));
```

for an entry point that returns a character string of arbitrary length.

Execution of a RETURN-statement not containing an expression ends execution of the procedure and causes control to return to the point of call. The END-statement of a procedure is treated as having an implicit RETURN-statement just in front of it, so that if control flows to the END-statement, execution of the procedure is terminated. It is an error to allow control to flow to the END-statement of a procedure that was called as a function reference.

Arguments and Parameters

An entry point of a procedure can have a sequence of parameters associated with it. A call on the entry point must include a corresponding sequence of arguments, which act as inputs to the procedure. If

(1) the argument is a reference to a variable (possibly subscripted or name-qualified), and

(2) the attributes of the argument agree with those of the parameter,

then the parameter becomes an alias for the argument, and assignments to the parameter affect the argument. In all other cases, the argument is considered to be a *dummy*. That is, when the call is made, a generation of storage -- the dummy -- is set aside for the argument, and the value of the argument is copied into that generation. If the type of the argument disagrees with the type of the parameter, the argument is converted to the parameter type and the converted value is assigned to the dummy. The parameter is then an alias for

the dummy, and after the call is completed the dummy is discarded. Thus, assignments to a parameter that corresponds to a dummy argument have no effect at the point of call. Constants and expressions are always passed as dummy arguments.

PL/I uses the call-by-reference model of argument transmission, i.e., the location of the argument is passed to the procedure. The conventions for argument transmission are shown by the following example:

```
CALLER:     PROCEDURE;
            DECLARE X FIXED DECIMAL(5);
            DECLARE Y FLOAT DECIMAL(7);
            CALL CALLEE(X);  /* X IS SET TO 12 BY THE CALL */
            CALL CALLEE(Y);  /* DUMMY CREATED, SO Y IS UNCHANGED
            CALL CALLEE(24962);  /* DUMMY CREATED */
            CALL CALLEE((X));  /* DUMMY CREATED SINCE (X) IS
                                  AN EXPRESSSION */
            CALL CALLEE(X+14);  /* DUMMY CREATED HERE, TOO */
            END CALLER;
CALLEE:     PROCEDURE(P);
            DECLARE P FIXED DECIMAL(5);  /* P IS THE PARAMETER*/
            P = 12;
            END CALLEE;
```

Array sizes, string lengths, and area sizes of parameters must be given either by constant-valued expressions or by asterisks. An asterisk size is used when the size of the

corresponding argument is unknown, or varies from one call to another. Thus a parameter declared as CHARACTER(*) will match an argument declared as CHARACTER(*e*), where *e* is any expression. However, such a parameter will not match an argument declared as CHARACTER(*e*) VARYING.

Options

Implementation-defined information can be attached either to a PROCEDURE-statement or to the declaration of an entry constant by means of the OPTIONS-attribute. A particularly common option (but not a universal one) is illustrated by

```
PROCEDURE OPTIONS(MAIN);
```

where the MAIN option indicates that execution of the program is to start with this procedure. In general, the information given in an OPTIONS-attribute affects the manner in which the procedure is compiled.

When a PL/I procedure references a procedure written in a different programming language, the OPTIONS-attribute can be used to specify the language of that foreign procedure so that appropriate calling sequences can be compiled. For instance,

```
DECLARE PRIMFN ENTRY(FLOAT) RETURNS(FLOAT)
    OPTIONS(FORTRAN);
```

would describe a procedure written in Fortran to be called from a procedure written in PL/I.

Recursion

A PL/I procedure is permitted to call itself, either directly or indirectly. A procedure that calls itself is said to be *recursive*, and the RECURSIVE option must be specified on the PROCEDURE-statement of such a procedure. An example of a recursive procedure is one that counts the number of nodes in a binary tree. Each node is represented as a based structure, and contains a value, a left son, and a right son. Each son is either itself a pointer to a binary tree, or null. The procedure in PL/I is:

```
COUNTNODES:
  PROCEDURE(NODEPTR) RECURSIVE RETURNS(FIXED);
  DECLARE(LCOUNT,RCOUNT) FIXED INITIAL(0);
  DECLARE
    1 NODE BASED(NODEPTR),
      2 LEFT_SON POINTER,
      2 RIGHT_SON POINTER,
      2 VALUE FIXED;
  DECLARE NODEPTR POINTER;
  DECLARE NULL BUILTIN;
  IF LEFT_SON ¬= NULL THEN
     LCOUNT = COUNTNODES(LEFT_SON);
  IF RIGHT_SON ¬= NULL THEN
     RCOUNT = COUNTNODES(RIGHT_SON);
  RETURN(LCOUNT+RCOUNT+1);
  END COUNTNODES;
```

The procedure is given a pointer to a binary tree as an argument, and it returns the number of nodes in the tree as its value. Recursiveness is a property of a procedure rather than of its entry points, so that even if the recursive call is on a different entry point, the procedure is still considered to be recursive.

The GENERIC-Attribute

Often it is useful to create a family of entry points that perform a similar function but that expect somewhat different arguments, and to assign a single name to the family. The GENERIC-attribute allows a single name, known as a *generic function*, to be used for such a family of entry points; the choice of entry points then depends on the nature of the arguments. The GENERIC-attribute specifies a list of entry-point names, and associates a sequence of generalized descriptors with each name. A reference to the generic function is translated into a reference to the first entry point whose descriptors, as given by the GENERIC-attribute, match the arguments of the generic function. An asterisk indicates a descriptor that matches anything. The test for descriptor matching is satisfied if the descriptor in the GENERIC-attribute is contained in the attribute set of the argument; the attribute set can contain attributes not in the descriptor. Only data attributes can be tested in this way.

An example of a GENERIC-attribute is

```
DECLARE GF GENERIC (G1 WHEN (FIXED, FIXED),
   G2 WHEN (FIXED, *),
   G3 WHEN (*));
```

Using this declaration, and assuming the further declarations

```
DECLARE X FIXED BINARY;
DECLARE Y FLOAT DECIMAL;
```

the reference

```
GF(X,X+1)    translates to  G1(X,X+1)
```

The reference

```
GF(X,Y+1)    translates to  G2(X,Y+1)
```

since the expression Y+1 has data type float. The reference

```
GF(X)        translates to  G3(X)
```

since the first two descriptor sequences each require two arguments.

Another application of the GENERIC-attribute is illustrated by the declaration

```
DECLARE VARFN GENERIC(NF1 WHEN(FLOAT(1:20) BINARY),
   NF2 WHEN(FLOAT(21:40) BINARY),
   NF3 WHEN (*));
```

In this case, the entry point represented by VARFN is selected on the basis of the precision of the argument, which is assumed to be float binary. If the argument has from 1 to 20 binary digits, NF1 is used; if the argument has from 21 to 40 binary digits, NF2 is used; and if the argument has more than 40 binary digits, NF3 is used.

Blocks and Scopes

A block consists of a sequence of statements, starting with a PROCEDURE-statement or a BEGIN-statement and extending to the matching END-statement. Blocks of either kind can be nested. The primary effect of the block structure of a program is to define the scope of a name, i.e., the set of statements from which the name can be referenced. A name declared in a

DECLARE-statement belongs to the innermost block containing that DECLARE-statement. However, a name can also be declared by virtue of its appearance as a parameter or as a statement-name. A statement-name that names a PROCEDURE-statement, an ENTRY-statement, or a BEGIN-statement belongs to the block *outside* the one that contains that statement; any other statement-name belongs to the block containing the statement that it names. This rule is needed in order to allow procedures to be called from the outside. A reference to a name is resolved by searching the nest of blocks for a declaration of the name, working from the inside out, and starting with the statement containing the reference. Another way of looking at it is that the scope of a name consists of the block declaring the name and all contained blocks except for those in which the scope is occluded by an inner declaration of the same name.

An example illustrating the scope of names is given in Figure 4. The parenthesized numbers are used to distinguish different declarations of the same identifier. There is an imaginary outer block used to hold the declarations of the entry points of the external procedure (A and B in this case). This block is needed since the entry points of a procedure belong, not to the block of the procedure itself, but to the next outer block. Since there is no such block for the external procedure, one must be created.

Figure 4. Example Illustrating Scopes of Names

```
 1   A:      PROCEDURE;
 2           DECLARE X CHARACTER(1);
 3           DECLARE B FIXED;
 4           statement sequence 1
 5   B:      ENTRY(Y);
 6           statement sequence 2
 7   C:         BEGIN;
 8              DECLARE W FIXED;
 9              DECLARE Y PICTURE '(6)$';
10              statement sequence 3
11   D:            PROCEDURE;
12              DECLARE W FLOAT COMPLEX;
13              statement sequence 4
14              END D;
15           END C;
16   E:         PROCEDURE;
17              DECLARE W FLOAT;
18              statement sequence 5
19   F:         B = 3;
20   G:         ENTRY;
21              statement sequence 6
22           END E;
23         END A;
```

Statements belonging to different blocks:

outer	none
A(1)	1,2,3,4,5,6,23
C(7)	7,8,9,10,15
D(11)	11,12,13,14
E(16)	16,17,18,19,20,21,22

Names belonging to different blocks:

outer	A(1), B(5)
A(1)	X(2),B(3),Y(5),C(7), E(16),G(20)
C(7)	W(8),Y(9),D(11)
D(11)	W(12)
E(16)	W(17),F(19)

Statements in scopes of these names:

A(1)	1-23	Y(9)	7-15
X(2)	1-23	D(11)	7-15
B(3)	1-23	W(12)	11-14
B(5)	none	E(16)	1-23
Y(5)	1-6,16-23	W(17)	16-22
C(7)	1-23	F(19)	16-22
W(8)	7-10,15	G(20)	1-23

Internal and External Scope

In most cases, declarations are defaulted to have *internal* scope, meaning that the declaration designates an object distinct from the objects designated by other dcclarations of the same identifier. For instance, if the variable Q is declared in three different blocks of a procedure with the INTERNAL-attribute (possibly by default), then each of these blocks has its own distinct Q. However, several declarations can be made to refer to the same object by giving them *external* scope. External scope cannot be applied to just any declaration; it is restricted to the static and controlled storage classes, and to named constants. Identifiers declared to be external must necessarily have the same attributes. As an example, if the declaration

```
DECLARE A(14) STATIC EXTERNAL;
```

appears in two different blocks, then both declarations refer to a single array. If an assignment is made to A(4) while executing one of these blocks, then the change will be visible in the other. Declarations of an external identifer can appear both within a single external procedure and among several external procedures. The default scope for manifestly declared entry constants is external, since external procedures have to be declared by the programmer while internal procedures are automatically declared.

Entry Values and Environments

On account of the rules for scope of names in PL/I, a procedure can refer to names in blocks surrounding the procedure. Moreover, an entry point defines an entry value, and that value can be assigned to an entry variable and subsequently invoked. Invocation of the entry point, in turn, requires that references to outer-block names be resolved properly. In order to achieve this effect, an entry value contains not only a designation of an entry point but also an environment. When the block surrounding the entry point is entered, the environment of the entry point is defined. The entry value corresponding to the entry point then consists of the entry point itself together with a record of all names inherited from outer blocks and the variables (or constants) that these names denote. In the case of recursive procedures, the environment implicitly designates not only a set of variables, but also a recursion level. The following example illustrates these concepts:

```
P:      PROCEDURE;
Q:      PROCEDURE(R,LEVEL) RECURSIVE;
        DECLARE R ENTRY;
        DECLARE LEVEL FIXED;
        IF LEVEL=10 THEN
           CALL R( );
        ELSE IF LEVEL=6 THEN
           CALL Q(S,7);
        ELSE
           CALL Q(R,LEVEL+1);
S:      PROCEDURE;
        PUT DATA(LEVEL);
        STOP;
        END S;
        END Q;
T:      PROCEDURE;
        END T;
        CALL Q(T,1);
        END P;
```

The call on Q on the next-to-the-last line initiates a nest of recursive calls. On each call, the value of LEVEL increases by 1. At the top level, the entry value T is passed as an argument to Q; but since this entry value is never invoked, it serves only as a place-holder. At the sixth level of recursion, the entry value S is passed as part of the recursive call on Q. The environment of this entry value consists of the current set of outer-block variables -- outer, that is, to S. In particular, since S is internal to Q, the current value of LEVEL -- 6 in this case -- is part of the environment accompanying the entry constant S. On subsequent recursive calls, the entry value is simply passed along (cf. the call with parameters R and LEVEL+1). When the recursion level reaches 10, R is called. The value of R is the entry constant S obtained at level 6, and so

LEVEL = 6

is printed out and the program halts.

The PL/I rules for block structure, scoping of names, and environments are derived from Algol 60. In fact, it is possible to transcribe the example above rather directly into Algol 60, and the behavior in Algol 60 would be the same.

ON-UNITS AND ON-STATEMENTS

One of the more innovative aspects of PL/I is the facility that it provides for handling exceptional conditions -- the so-called ON-conditions. An exceptional condition may arise either as the result of an error, such as a subscript out of range, or from an anticipated event, such as encountering end-of-file while reading from a dataset or reaching the end of the program. PL/I has a set of ON-conditions corresponding to these exceptional conditions. When the condition occurs, it is said to be *raised*. Using an ON-statement, the programmer can specify a response to the condition in the form of a statement (or begin-block) to be executed. That response is known as the *ON-unit*. If no ON-unit has been specified, a standard ON-unit, known as the *standard system action*, is executed. Depending on the nature of the ON-condition, it may be possible for the program to continue where it left off after the ON-unit is executed. The different kinds of ON-conditions are listed in Table 7.

An example illustrating the use of ON-conditions and ON-units is:

```
P:        PROCEDURE;
          ...
          ON ENDFILE(SYSIN)
             GO TO PROCESS;
          DO WHILE('1'B);
             READ FILE(SYSIN) INTO(LINE_IMAGE);
             ...
             END;
PROCESS:  ...
          END P;
```

Table 7. The ON-Conditions

Condition	Cause	Standard System Action	Enablement Possible?	Normal Return Allowed?
AREA	allocation in an area with insufficient space left	comment, signal ERROR	no	yes
CONDITION (identifier)	programmer signal	comment	no	yes
CONVERSION	value being converted from character type not in proper form	comment, signal ERROR	yes	yes
ENDFILE(file)	end of data on input file	comment, signal ERROR	no	yes
ENDPAGE(file)	end of page on print file	start a new page	no	yes
ERROR	miscellaneous execution-time errors	implementation-defined	no	no
FINISH	end of program execution	no action	no	yes
FIXEDOVERFLOW	result of fixed arithmetic operation exceeding maximum permitted value	comment, signal ERROR	yes	no
KEY(file)	duplicate or unacceptable key encountered during record input-output	comment, signal ERROR	no	yes
NAME(file)	unrecognized name encountered in data-directed stream input	comment and ignore item	no	yes
OVERFLOW	result of float arithmetic operation exceeding maximum representable value	comment, signal ERROR	yes	no
RECORD(file)	record of incorrect length encountered during record input-output	comment, signal ERROR	no	yes

Table 7. Continued

Condition	Cause	Standard System Action	Enablement Possible?	Normal Return Allowed?
SIZE	fixed value exceeds capacity of target of assignment	comment, signal ERROR	yes	no
STORAGE	available storage exhausted	implementation-defined	no	yes
STRINGRANGE	argument of SUBSTR references nonexistent portion of a string	comment, signal ERROR	yes	no
STRINGSIZE	length of string exceeds length of target of assignment	no action	yes	yes
SUBSCRIPTRANGE	subscript out of range	comment, signal ERROR	yes	no
TRANSMIT(file)	transmission error during input-output operation	comment, signal ERROR	no	yes
UNDEFINEDFILE (file)	file cannot be opened	comment, signal ERROR	no	yes
UNDERFLOW	result of float arithmetic operation smaller than minimum representatable value	comment	yes	yes
ZERODIVIDE	attempted division by zero	comment, signal ERROR	yes	no

Initially enabled conditions:
CONVERSION
FIXEDOVERFLOW
OVERFLOW
STRINGSIZE
UNDERFLOW
ZERODIVIDE

Initially disabled conditions:
SIZE
STRINGRANGE
SUBSCRIPTRANGE

In this case, the ON ENDFILE statement specifies that when end-of-file is encountered on the file SYSIN, the statement GO TO PROCESS; is to be executed. Thus the file will be read up to its end, and afterwards the statements at PROCESS will be executed.

The ON-Statement, REVERT-Statement, and SIGNAL-Statement

The ON-statement specifies a list of ON-conditions together with an ON-unit. The association of the ON-unit with the ON-conditions is not made until the ON-statement is actually executed. Moreover, execution of a subsequent ON-statement can supersede the effect of an earlier one. For instance, after execution of the two ON-statements

```
ON OVERFLOW, FIXEDOVERFLOW
   GO TO TOOBIG;
   ...
ON FIXEDOVERFLOW
   GO TO FIXEDBIG;
```

the ON-unit associated with the OVERFLOW condition is

```
GO TO TOOBIG;
```

while the ON-unit associated with the FIXEDOVERFLOW condition is

```
GO TO FIXEDBIG;
```

ON-statements have block scope, in the sense that they are effective only until the block containing them is terminated. When execution of a block is completed, the association between ON-conditions and ON-units reverts to what it was in the previously-executing block. Thus a procedure can activate a collection

of ON-units appropriate to its circumstances without affecting the ON-units set up by its caller.

The ON-unit itself can be either a BEGIN-block (delimited by BEGIN and END) or a single unconditional statement. In particular, an ON-unit cannot be either a DO-group or an IF-statement. Actual execution of the ON-unit is carried out as though the ON-unit were a procedure. In particular, ON-units carry environments with them, and so any names occurring in an ON-unit have the meaning applicable at the point of execution of the corresponding ON-statement.

An ON-statement can specify the standard system action as an ON-unit, using the keyword SYSTEM. Thus

```
ON SUBSCRIPTRANGE
   SYSTEM;
```

specifies that the standard system action is to be taken if the SUBSCRIPTRANGE-condition is raised. This facility can be used to nullify the effect of previously executed ON-statements. It is also possible to specify that a traceback, or other debugging information, is to be printed in the event that a condition is raised. That effect is gotten by using the SNAP keyword, as in

```
ON SIZE SNAP SYSTEM;
```

If the SIZE-condition is raised, the standard system action will be taken, but in addition debugging information will be printed. The actual choice of debugging information is implementation-defined. The statement

```
ON SIZE SNAP;
```

would produce a different effect: if the SIZE-condition is raised, the null-statement, which does nothing, will be executed. In fact, the null-statement is not a valid ON-unit for the SIZE-condition because it does not terminate in a GOTO-statement. The question of validity of such ON-units is discussed below.

The REVERT-statement can be used to cancel the effect of an ON-statement, or several of them, without knowing what ON-condition was in effect previously. The REVERT-statement specifies a list of ON-conditions. Execution of the REVERT-statement causes the ON-unit for each of these conditions to revert to what it was in the previously-executing block. Thus, in the sequence:

```
ON ENDFILE(SYSIN)
   CALL ENDER;
BEGIN;
   ...
   ON ENDFILE(SYSIN)
      GO TO ALT_END;
   ...
   REVERT ENDFILE(SYSIN);
   ...
   END;
```

the REVERT-statement causes the on-condition

```
CALL ENDER;
```

to again be associated with the ENDFILE-condition for the file SYSIN.

The SIGNAL-statement is used to raise a specified ON-condition. For example, the statement

```
SIGNAL ZERODIVIDE;
```

causes the ZERODIVIDE-condition to be raised and the appropriate ON-unit (possibly the standard system action) to be invoked. This statement is particularly useful in debugging program logic for handling ON-conditions. It is also the only way to raise a programmer-defined condition (discussed below).

Enablement and Disablement

A number of the ON-conditions require time-consuming code (on most machines, at least) in order to check whether or not they have occurred. The time needed to check whether a subscript is out of range, for instance, well may dominate the time needed for the retrieval of a subscripted variable. Therefore PL/I allows the programmer to either enable (turn on) or disable (turn off) the check. Enablement and disablement are provided only for certain ON-conditions. They are specified by means of a *condition prefix*, which consists of either an ON-condition name or the negation of an ON-condition name, in parentheses and followed by a colon. The condition prefix can be applied either to a single statement or to a block. For example, in the sequence:

```
                (OVERFLOW,NOSIZE):
P:              PROCEDURE;
                ...
(NOOVERFLOW):   Q = A + BTR(I);
                ...
                END;
```

the SIZE-condition is disabled throughout the procedure P, while the OVERFLOW-condition is enabled throughout P except for the

single statement where NOOVERFLOW is indicated. For that statement, the condition is disabled, and no test will be made for it. If a condition is raised in a statement where it has been disabled, that is considered to be a programmer error, and the implementation is not to be held responsible for its consequences.

Enablement and disablement are static properties of a program. In other words, it is possible to tell whether a particular ON-condition is either enabled or disabled for a particular statement just by looking at the program, without considering what its sequence of execution is. In this respect, enablement and disablement differ from the ON-statements, whose execution depends on program flow. Enablement and disablement affect whether a condition is or is not to be tested for, while ON-statements determine what action is to be taken if the condition is raised. The statement

```
ON UNDERFLOW;
```

does not disable the UNDERFLOW-condition; it merely states that if that condition is detected, the null-statement is to be executed.

Builtin Functions for ON-Conditions

During the execution of an ON-unit, a number of builtin functions are available in order to determine the circumstances that caused the corresponding ON-condition to be raised. Some of these apply to all ON-conditions and are discussed here;

other are specific to particular ON-conditions and are discussed in connection with those conditions. In general, these builtin functions do not have meaningful values except in the context of an ON-unit. They are all functions of no arguments.

The ONCODE builtin function has as its value an implementation-defined integer used to indicate why the active ON-condition was raised. A particular condition may have more than one code value associated with it. One common convention is that the value of ONCODE is zero if the ON-condition was raised by a SIGNAL-statement. The ONLOC builtin function returns as its value the name of the innermost entry point active when the condition was raised. For input-output-related ON-conditions, the ONFILE builtin function has as its value the name of the file that was being operated upon when the condition was raised. The values of both ONLOC and ONFILE are in the form of character strings.

Categorization of the ON-Conditions

The various ON-conditions listed in Table 7 can be broken down into three groups. The first group consists of the computational ON-conditions. Most conditions in this group are raised in response to a particular kind of error. The computational ON-conditions are the only ones that can be enabled and disabled. They are:

```
CONVERSION
FIXEDOVERFLOW
OVERFLOW
SIZE
STRINGRANGE
STRINGSIZE
SUBSCRIPTRANGE
UNDERFLOW
ZERODIVIDE
```

The occurrence of one of these conditions usually means that a bad result has been generated, and so the active computation cannot be continued. For this reason, the ON-units associated with most of these conditions must not terminate normally, i.e., must cause a transfer of control out of the ON-unit by means of a GOTO-statement or similar construction. Normal termination would mean that the active computation would be resumed, and the nature of the condition is such that the computation cannot be resumed. For instance, if a subscript is out of range on an array reference, there is no way to obtain an appropriate value for the reference.

Three of the computational ON-conditions are treated somewhat differently. The CONVERSION-condition is raised when data is being converted from character to some other type. When this condition is raised, the programmer can modify the character string to be converted. If a normal return takes place from the ON-unit, i.e., the ON-unit completes without a transfer of control, the conversion is reattempted with the modified input string.

Two builtin functions are available for the modification: ONSOURCE and ONCHAR. ONSOURCE has as its value the character string to be converted, while ONCHAR has as its value the left-

most character in that string for which no valid continuation exists. By examining ONSOURCE and ONCHAR, the programmer may be able to determine the difficulty and what to do about it. Moreover, ONSOURCE and ONCHAR can be used on the left side of an assignment (within an ON-unit for the CONVERSION-condition), and so the string to be converted can be modified by assignments to either ONSOURCE or ONCHAR (which can also be used as pseudovariables). For example, if a character string is being converted to a bit string the following ON-unit might be appropriate:

```
ON CONVERSION BEGIN;
   DECLARE ONCHAR BUILTIN;
   IF ONCHAR='Ø' THEN
      ONCHAR = '0';
   ELSE ONCHAR = '1';
   END;
```

If the string to be converted does not consist entirely of ones and zeros, each blank in that string will be replaced by a zero, and each other deviant character will be replaced by a one.

The UNDERFLOW-condition also receives slightly different treatment. Normal return from the UNDERFLOW-condition is permitted, and the value of the computation that underflowed is taken to be zero. The STRINGSIZE-condition arises when a string is shortened as a result of a conversion or assignment. Upon normal return from the ON-unit, the string is truncated on the right to the required length. Since the standard system action in this case is to do nothing, this condition is often ignored. However, it can be used in either of two ways. If it is disabled, then the compiler need not produce code to check for string

overflow. Moreover, if a nonstandard ON-unit is provided, then the programmer can take some action. However, there is no way that the programmer can modify the result produced either for the UNDERFLOW-condition or for the STRINGSIZE-condition.

The second group of ON-conditions is the input-output conditions. Each of these conditions is associated with a particular file, specified along with the condition name. The input-output conditions are:

ENDPAGE
ENDFILE
KEY
NAME
RECORD
TRANSMIT
UNDEFINEDFILE

Some of these conditions are discussed further in connection with input-output.

The remaining conditions are more varied. These are:

AREA
CONDITION
ERROR
FINISH
STORAGE

The AREA-condition is raised when an allocation is attempted in an area, and there is insufficient space for the allocation. If the associated ON-unit returns normally, the area-reference in the ALLOCATE-statement is reevaluatd, and the allocation is reattempted. Therefore an appropriate response to the AREA-condition is to assign a new area value to the area variable referenced in the ALLOCATE-statement.

The programmer can define ON-conditions using the keyword CONDITION and an identifier, known as the *condition-name*. ON-units can be provided for programmer-named conditions, but they can only be raised by a SIGNAL-statement. For instance, a programmer might write:

```
ON CONDITION(TABLE_OVERFLOW)
   CALL OVERFLOW_RECOVERY;
```

and then, in some other part of the program, write:

```
IF T > TABSIZE THEN
   SIGNAL TABLE_OVERFLOW;
```

The ERROR-condition is raised under a variety of circumstances, some of which can be implementation-defined. The standard system action in response to a number of other ON-conditions is to comment (i.e., display diagnostic information) and then to raise the ERROR-condition. (It is quite acceptable to have an ON-unit raise an ON-condition itself.)

The FINISH-condition is raised when the program completes. It differs from all other conditions in that it is raised as a normal aspect of program execution. The STORAGE-condition is raised when the program runs out of storage. Since programs consume storage in many different ways, the exact circumstances under which it is raised are implementation-defined. Recovery from this condition may or may not be possible.

OTHER STATEMENTS AFFECTING FLOW OF CONTROL

Conditional Statements

The conditional statement is used in order to test a condition and take some action depending on the result. A conditional statement starts with an IF-statement, specifying the test, and may include an ELSE-part that specifies what action to take if the test fails. For instance, the sequence

```
IF Q <= QMAX THEN
   INDEX = INDEX+1;
ELSE
   GO TO PART_7;
```

causes the assignment

```
INDEX = INDEX+1;
```

to be executed if the condition Q <= QMAX is true, and the statement

```
GO TO PART_7
```

to be executed otherwise. The statement following either THEN or ELSE can itself be a conditional statement, so that nests of conditional statements can be built up. Moreover, either THEN or ELSE can be followed by a DO-group (discussed below), so that several statements can be executed after the test rather than just one. An example of a more complicated conditional statement is:

```
IF A(I)=0 THEN DO;
     SIZE1=SIZE1+INCR;
     SIZE2=SIZE2-INCR;
     IF SIZE2<SIZE1 THEN
          CALL ADJUST;
     END;
END IF A(I)>0 THEN
     SIZE2=SIZE2+INCR;
ELSE
     SIZE1=SIZE1-INCR;
```

It is not necessary that each IF-statement have a corresponding ELSE-statement. In complicated conditional statements, each ELSE is paired with the nearest preceding unpaired IF, working from front to back.

The test in an IF-statement actually takes the form of an expression, which is evaluated and converted to a bit string. Since the comparison operators all produce one-bit results, and since the logical operators also produce one-bit results when their operands are one-bit values, the conversion is usually unnecessary. If the bit string obtained by evaluating the test expression has at least one one-bit in it, the test succeeds, and otherwise it fails. The test expression must be scalar-valued, although if it is not scalar-valued the SOME and EVERY builtin functions can be used to reduce it to a scalar value.

The DO-Statement

The DO-statement has three main variants: the simple DO, the DO-WHILE, and the specified DO. The simple DO is used in order to convert, syntactically, a sequence of statements into a single statement. A simple DO-group has the form:

```
DO;
  statement-sequence
  END;
```

The statements in the sequence are executed just once. Transfers of control into and out of the sequence are permitted. The main use of the simple DO-group is as part of a conditional statement.

The DO-WHILE variant has the form:

```
DO WHILE(expression);
```

A DO WHILE-group consists of a DO WHILE-statement followed by a statement sequence and a matching END-statement. The statements in the group are executed repeatedly, and the expression in the DO WHILE-statement is tested before each execution. If the test fails, control is transferred to the statement following the group. If the expression is initially false, the group is not executed at all. An example of a DO WHILE-group is:

```
DO WHILE(CVAL>0);
   DVAL=DVAL+G(CVAL);
   CVAL=CVAL-DVAL;
   END;
```

The specified DO itself has a number of variants. As with the other two forms, a DO-group consists of a specified DO-statement followed by a statement sequence followed by an END-statement. The most common variant is illustrated by:

```
DO M = 0 TO 100 BY 2;
```

In this case the statements in the group are executed repeatedly. Before the first execution, M is assigned the value 0. M is then increased by 2 on each execution of the group, and has

the value 100 on the last execution of the group. Upon completion of the entire group, the value of M is 102. However, transfer out of the group is permitted, and if that happens, M retains the value assigned to it on the most recent iteration.

The TO-clause and the BY-clause can be written in either order, and either of them can be omitted. If the TO-clause is omitted, the group is iterated indefinitely, i.e., until a transfer of control out of the group takes place. If the BY-clause is omitted, a value of 1 is assumed for it. On each iteration, the control variable (the variable following the keyword DO) is incremented by the value given in the BY-clause. If the BY-clause has a negative value, then the control variable is decremented rather than incremented. The loop terminates when the value of the control variable is greater than the value of the TO-clause (for a positive BY-value) or less than the value of the TO-clause (for a negative BY-value). If the termination test is satisfied by the initial value of the control variable, the group is executed zero times. If neither the TO-clause nor the BY-clause appears, the group is executed for a single value of the control variable.

The TO-clause and the BY-clause are both evaluated prior to execution of the statements within the DO-group. Thus any changes to values of variables that appear within the TO-clause or the BY-clause have no effect once the iteration has started. The control variable need not have arithmetic type; a string or pictured type is also acceptable. A WHILE-clause can also be specified, e.g.,

```
DO JV = X BY Y WHILE(PROP(JV)<PROP(JV+1));
```

Another variant is illustrated by:

```
DO P = LIST_HEAD REPEAT (P->NEXT) WHILE(P¬=NULL);
```

or

```
DO STRING='' REPEAT(STRING||CHARS(I)) WHILE(LENGTH(STRING)<LMAX);
```

The control variable is assigned the given initial value on the first iteration. On subsequent iterations, the value of the REPEAT-clause is recalculated and assigned to the control variable. The WHILE-clause can be omitted, although usually it is desirable to include it.

The specified DO can consist of a sequence of specifications rather than a single one. For instance, the statement

```
DO M = 3,7,M+2 BY 3 TO 15,0;
```

executes the group of statements that follows for the sequence of values 3, 7, 9, 12, 15, 0. Each specification in the group can have the general forms described above.

The GOTO-statement

The GOTO-statement causes control to be transferred to the label specified in the statement. The statement actually specifies a label-valued expression, and although that expression normally is a constant, i.e., a statement-name, it need not be. For instance, it could be a subscripted reference to an array of statement-names, so that the appropriate destination is selected by the value of an index.

The destination of a GOTO-statement need not be in the same block as the statement itself. If the destination is in a different block, then the effect of the statement is to terminate execution of the current block and all blocks between the statement and its destination. In other words, at the moment when the GOTO-statement is executed, there will be a hierarchy of active blocks, with the current block last in the hierarchy. The label value obtained from the GOTO-statement must designate, as its environment component, some block in the hierarchy. Then all blocks between the designated block and the current block, as well as the current block itself, are terminated. The designated block then becomes the current block, and control is transferred to the statement named by the label value.

A GOTO-statement whose destination is not in the same block as the statement itself is called a *nonlocal goto*. A nonlocal goto is expensive to execute relative to a local one. Therefore the programmer is allowed to declare the LOCAL-attribute for a label variable. The LOCAL-attribute constitutes a claim by the programmer that any GOTO-statement using the value of that label variable will be a local goto. Thus the compiler need not examine the environment associated with the label, and can generate instructions to execute the requested transfer of control directly.

The STOP-Statement and the Null-Statement

The STOP-statement has the form

```
STOP;
```

and is used to stop execution of the program. It has the effect of terminating the execution of all currently active blocks.

The null-statement has no text at all; it is written as just a semicolon. Its main uses are to place a statement-name, to fill out a branch of a conditional statement where no action is to be taken, and to specify that no action is to be taken in response to a specified ON-condition.

File Attributes

The attributes of a file determine the kinds of operations that can meaningfully be applied to that file. Moreover, they dictate to some extent the characteristics of the dataset associated with the file. The final determination of file attributes takes place when the file is opened, i.e., associated with a dataset. If a file is opened and closed several times, it can have different attributes at different openings.

The file attributes INPUT, OUTPUT, and UPDATE determine the direction of information flow in an obvious way. An opened file must have either the RECORD-attribute or the STREAM-attribute. A record file is associated with a dataset consisting of a sequence of *records*, which are read or written as single units. A record may or may not have a *key* associated with it. If it does, then the file has the KEYED-attribute. If the records are sequenced, i.e., the notion of "next record" is meaningful, then the file has the SEQUENTIAL-attribute; otherwise it has the DIRECT-attribute. A direct file is necessarily keyed, since without a key there is no way to designate a record within the file, while a sequential file may or may not be keyed.

A stream file is associated with a dataset consisting of a sequence of characters. Within the sequence of characters, *linemarks*, *pagemarks*, and *carriage-returns* can appear. A linemark marks the break between the characters on two succes-

sive lines (either input or output). A pagemark is meaningful only for a dataset that is to be printed, and marks the start of a new page. A carriage-return is meaningful only for a dataset that is to be printed, and indicates that the following line is to be overprinted, i.e., printed without advancing the carriage. A file associated with a dataset to be printed has the PRINT-attribute, and consequently the OUTPUT-attribute also.

A file may also have an ENVIRONMENT-attribute associated with it. The ENVIRONMENT-attribute contains implementation-defined information describing the associated dataset. Typical items found in the ENVIRONMENT-attribute are record length, blocking factors, record formats, character set selection, and tape densities.

File Opening and Attribute Determination

A file can be opened either explicitly, through execution of an OPEN-statement, or implicitly, through execution of some other input-output statement referencing an unopened file. When the file is opened, any attributes given in the file declaration are combined with those given in the OPEN-statement. The resulting partial set of attributes is then checked for consistency. If any inconsistency is found, the UNDEFINEDFILE-condition is raised for the file. Then defaulting rules are applied to generate a complete set of file attributes. The possible sets of file attributes are given in Table 8. For

TABLE 8. Complete Sets of File Attributes

```
STREAM INPUT FILE
STREAM OUTPUT FILE
STREAM OUTPUT PRINT FILE
STREAM INPUT  SEQUENTIAL FILE
RECORD INPUT  SEQUENTIAL KEYED FILE
RECORD INPUT  DIRECT KEYED FILE
RECORD OUTPUT SEQUENTIAL FILE
RECORD OUTPUT SEQUENTIAL KEYED FILE
RECORD OUTPUT DIRECT KEYED FILE
RECORD UPDATE SEQUENTIAL FILE
RECORD UPDATE SEQUENTIAL KEYED FILE
RECORD UPDATE DIRECT KEYED FILE
```

Note: The ENVIRONMENT-attribute may be added to any of these combinations.

example, in the sequence

```
DECLARE CHANGES INPUT FILE;
   ...
OPEN FILE(CHANGES) KEYED;
```

the initial set of attributes used for opening the file CHANGES is INPUT KEYED. Since these attributes are consistent with each other, the opening can proceed. The RECORD and FILE attributes are implied by the KEYED attribute, and so these are added to obtain the set RECORD INPUT KEYED FILE. Although both DIRECT and SEQUENTIAL are consistent with this set, the default choice is SEQUENTIAL, and so SEQUENTIAL is added to obtain the complete set

RECORD INPUT SEQUENTIAL KEYED FILE

The FILE-option in an OPEN-statement contains a file-valued expression, which is either a constant or something that evaluates to a file constant. It is assumed that each dataset has a name (typically, known to the surrounding operating system) and so the file opening has to specify the name of the dataset to be linked to the file. The name can be specified by a TITLE-option; if it is not, the name of the file constant obtained by evaluating the FILE-option is used. Thus the statements

```
DECLARE CHANGENAME CHARACTER(4);
...
CHANGENAME = 'C437';
OPEN FILE(CHANGES) TITLE(CHANGENAME) STREAM INPUT;
OPEN FILE(OLDSET) RECORD INPUT DIRECT;
```

cause the file CHANGES to be associated with the dataset C437, and cause the file OLDSET to be associated with the dataset OLDSET.

For stream files, other information can be given in the OPEN-statement. For instance,

```
OPEN FILE(LISTING) PRINT LINESIZE(110) TAB(10,40,70)
     PAGESIZE(50);
```

causes the file LISTING to be opened with the understanding that a new line will be started after at most 110 characters, a new page will be started after at most 50 lines, and tabstops (discussed under "Edit-Directed Input-Output" below) will be placed at column positions 10, 40, and 70. Were any of these values to be omitted, implementation-defined values would be assumed.

If an input-output statement is executed on a closed file, then the file is implicitly opened. For instance, if the statement

```
PUT FILE(ANS)(M,N);
```

is executed and the file ANS is not open, the file will be opened with the implicit attributes STREAM and OUTPUT. Similarly, if the statement

```
DELETE FILE(INV) KEY(PART_NAME);
```

is executed and the file INV is not open, it will be opened with the implicit attributes RECORD and UPDATE. The implicit attributes are treated as though they appeared on an OPEN-statement, so any attributes given in the declaration of the file are combined with those derived from the implicit opening.

File Closing

Just as the OPEN-statement creates the connection between a file and a dataset, the CLOSE-statement breaks the connection. All files are automatically closed at program termination. The programmer can also specify actions such as dataset disposition by means of an ENVIRONMENT-attribute attached to the CLOSE-statement, e.g.,

```
CLOSE FILE(BIBLIO) ENVIRONMENT(REWIND);
```

A file can be closed and later reopened with different attributes. An attempt to open a file that is already open, or to close a file that is closed, has no effect.

Operations on Record Files

There are five statements applicable to record files:

```
READ
WRITE
LOCATE
REWRITE
DELETE
```

Each of these, in turn, has a number of clauses that can be applied to it. The attributes of a file determine which statements and clauses can meaningfully be applied to that file. For instance, READ cannot be applied to an output file, DELETE can only be applied to an update file, and any clause that references a key can only be applied to a keyed file. The meanings of the statements are summarized in Table 9, and the meanings of the clauses are summarized in Table 10. Table 11

Table 9. Record Input-Output Statements

READ	read a record from a dataset (input and update files only)
REWRITE	replace a record on a dataset (update files only)
WRITE	add a record to a dataset (output and update files only)
LOCATE	obtain buffer space for a record (output files only)
DELETE	delete a record from a dataset (update files only)

Table 10. Clauses on Record Input Output Statements

FILE	specifies the file accessed by this statement
INTO	specifies the generation to receive a record being read
FROM	specifies the generation containing a record to be written
KEY	specifies a record to be read, replaced, or deleted
KEYFROM	specifies the source for a key to be attached to a record to be written
KEYTO	specifies where to put the key associated with a record being read
IGNORE	specifies the number of records to be skipped by execution of a read statement
SET	specifies where to put a pointer to a newly created generation

Table 11. Permissible Record Input-Output Statements

Form of Statement	INPUT			OUTPUT			UPDATE		
	S	KS	KD	S	KS	KD	S	KS	KD
READ FILE(f) INTO(v);	x	x					x	x	
READ FILE(f) INTO(v) KEY(k);		x	x				x	x	x
READ FILE(f) INTO(v) KEYTO(kv);		x	x				x	x	x
READ FILE(f) SET(p);	x	x					x	x	
READ FILE(f) SET(p) KEY(k);		x	x				x	x	x
READ FILE(f) SET(p) KEYTO(kv);		x	x				x	x	x
READ FILE(f) IGNORE(n);	x	x					x	x	
READ FILE(f) INGORE(n) KEY(k);		x						x	
READ FILE(f) IGNORE(n) KEYTO(kv);		x						x	
WRITE FILE(f) FROM(v);				x					
WRITE FILE(f) FROM(v) KEYFROM(kv);					x	x			x
LOCATE v FILE(f) SET(p);				x					
LOCATE v FILE(f) SET(p) KEYFROM(kv);					x	x			
REWRITE FILE(f);							x	x	
REWRITE FILE(f) FROM(v);							x	x	
REWRITE FILE(f) FROM(v) KEY(k);								x	x
DELETE FILE(f);							x	x	
DELETE FILE(f) KEY(k);								x	x

x indicates a permissible operation for this file type.

S sequential
KS keyed sequential
KD keyed direct

v variable
f file
p pointer
k key
kv key variable
n skip count

shows how the different statement forms relate to the different combinations of file attributes.

The chief characteristic of record input-output is that it involves a direct transfer of information between the dataset and addressable memory, without any formatting or editing. It is therefore the programmer's responsibility to be sure that the format of information on the dataset agrees with the format in memory, as defined by the implementation. If the dataset is itself created by a PL/I program, this is not too difficult.

The READ-statement causes a single record to be read from a specified file. It can also be used to skip records on a sequential file. Its simplest form, applied to a sequential file, is illustrated by

```
READ FILE(CUST) INTO(CUST_INFO);
```

A single record is transferred from the dataset associated with CUST into the variable CUST_INFO. If the size of the record disagrees with the size of the variable, the RECORD-condition is raised for the file. If there are no more records left in the dataset, the ENDFILE-condition is raised for the file.

The destination of a newly read record can be specified either by an INTO-clause or by a SET-clause. If the INTO-clause is given, then the record is read into a buffer, and the pointer specified in the SET-clause is set to the location of the record within the buffer. Although the SET-clause is

less convenient to use than the INTO-clause, it has two advantages. First, the SET-clause can be used to read records whose length is specified within the record itself. The INTO-clause cannot be used for such records because the size of the necessary variable is not known. Second, operating on the record in the buffer avoids the need to copy the record from the buffer to the variable.

The KEY-clause is used to specify the position within the dataset of the record to be read. For instance, the statement

```
READ FILE(EMPL) INTO(EMPLOYEE_RECORD) KEY(EMP_NUMBER);
```

causes the record whose key is the character-string form of EMP-NUMBER to be read into the variable EMPLOYEE_RECORD. (If the key value is not a character string, it is converted to one.) If the key designates a nonexistent record, the KEY-condition is raised for the file.

The KEYTO-clause, in contrast, does not influence the dataset position at all. Instead the key in the record being read is assigned to the variable specified in the KEYTO-clause. This facility is necessary since the key may not be part of the record itself. For instance, the statement

```
READ FILE(EMPL) INTO(EMPLOYEE_RECORD) KEYTO(EMP_NUMBER);
```

causes the next record from EMPL (which must be keyed sequential) to be read into EMPLOYEE_RECORD, and the key associated with that record to be assigned to EMP_NUMBER.

The IGNORE-clause can be used in order to skip records. For instance, the statement

```
READ FILE(EMPL) IGNORE(2);
```

causes two records on the dataset associated with EMPL to be skipped. (EMPL must necessarily be sequential.) The IGNORE-clause and the KEY-clause can be used together, and if the count in the IGNORE-clause is zero the effect is to position the dataset at the record designated by the KEY-clause. The IGNORE-clause and the KEYTO-clause can be used together to read a key without reading the associated record.

The WRITE-statement can be used only with output or direct update files. It causes the variable named in the FROM-clause to be written onto the dataset. If the dataset is sequential, the new record is written at its end; otherwise the position of the new record is arbitrary. The KEYFROM-clause, which specifies the key of the new record, must be used if the file is keyed.

The LOCATE-statement can be used only with output files. It is analogous to the READ-statement with the SET-option. Execution of a LOCATE-statement causes space for the variable named in the statement to be allocated in an output buffer, and then causes the pointer named in the SET-clause to be set to the location of this space. If the variable is declared with a pointer, then the SET-clause can be omitted and the pointer obtained from the declaration will be used. For instance, given the statements

```
DECLARE RECSPACE CHARACTER(200) BASED(CHPTR);
DECLARE CHPTR POINTER;
```

the statements

```
LOCATE RECSPACE FILE(NEWSET);
```

and

```
LOCATE RECSPACE FILE(NEWSET) SET(CHPTR);
```

are equivalent. After execution of a LOCATE-statement, the programmer can construct a record within the buffer, referencing the record using the variable and pointer specified in the LOCATE-statement. The record is written when either another LOCATE-statement or a WRITE-statement is executed for the file. Closing the file after executing the LOCATE-statement also causes the record to be written.

The REWRITE-statement can only be used with update files; it causes a record in the dataset to be replaced. The FROM-option can be omitted if the preceding operation on the file was a READ with the SET-clause; in this case the record just read (which is assumed to have been modified) replaces its old copy in the dataset. Otherwise the FROM-clause specifies the source of the replacement record, and the KEY-clause, if given, specifies which record is to be replaced. If the KEY-clause is not given, then the file must be sequential, and the record at the current position in the file is replaced. For the replacement to be acceptable, the current position must be well-defined, and the preceding operation on the file must not have been a DELETE.

The DELETE-statement, like the REWRITE-statement, can only be used with update files. If a key is specified, then the record with that key is deleted. Otherwise the behavior is similar to that of REWRITE: the record at the current position is deleted; the current position must be well-defined; and the preceding operation must not have been another DELETE.

STREAM INPUT-OUTPUT

Stream input-output differs from record input-output in that a transformation is performed when information is moved between a generation of storage and a dataset. On output, the transformation consists of translating a data value (which must be of a printable type) into a character representation of that value; on input, the transformation goes in the opposite direction. The character representation need not represent the value directly, since formatting conventions can be used. For instance, the character sequence "4387" could represent the value 43.87 on either input or output, were an appropriate format to be used. The GET-statement is used for stream input, while the PUT-statement is used for stream output. Each of these statements has three variants: list-directed, data-directed, and edit-directed. With the exception of a few pathological cases, input is the inverse of output. Thus, if information is written by a PUT-statement and later read by a GET-statement of the same form, the original values in storage will be unchanged. However, it is not in general true that reading from a file and then writing what was read will yield the contents of the original file.

The file to be operated on by a GET-statement or a PUT-statement can be specified either explicitly or implicitly. The statement

```
PUT FILE(TABLES) LIST(A,B);
```

causes the printable representation of the values of A and B

to be written onto the file TABLES. If the FILE-clause is omitted, the standard input file SYSIN is assumed for GET-statements, and the standard output file SYSPRINT (a print file) is assumed for PUT-statements.

The stream input-output statements can be used to encode and decode strings in storage by means of the STRING-clause. For example, the statement

```
GET STRING(STR) LIST(A,B);
```

causes the values of the variables A and B to be "read" from the string STR, just as though STR was a sequence of characters on an input file. Similarly,

```
PUT STRING(STR) LIST(A,B);
```

causes the values of A and B to be converted to their printable representations, and then causes the sequence of representations (with an intervening blank) to be assigned to the character variable STR.

Line and page skips can be specified in the GET-statement and the PUT-statement, although certain forms are applicable only to a PUT-statement that designates a print file. For instance,

```
PUT SKIP(2) LIST(A,B);
```

causes two lines to be skipped before A and B are printed, while

```
PUT PAGE LIST(A,B);
```

causes a new page to be started before A and B are printed. The statement

```
PUT LINE(2) LIST(A,B);
```

causes the printer to be positioned to the second line on the page before A and B are printed; it differs from SKIP in that it selects an absolute page position rather than a page position relative to the previous line. PAGE and LINE can only be specified for print files. SKIP on an input file causes the remainder of the input line to be skipped.

Data Lists

A GET-statement or a PUT-statement can contain one or more *data lists*, specifying the items to be read or written. In its simplest form, the data list is merely a sequence of scalars, e.g.

```
PUT LIST(RATE,TIME,RATE*TIME);
```

As this example shows, expressions as well as variables can appear in the data list of a PUT-statement. The items in the data list of a GET-statement, however, have the same restrictions as the targets of an assignment-statement, since one cannot read a value into an expression. A data list can also contain iterations, e.g.,

```
PUT LIST((I,F(I) DO I = 1 TO FNLIMIT));
```

Since any item can itself be an iteration, iterations can be nested to any depth. The DO that controls the statement is subject to the same restrictions as the specified-DO-statement.

A GET-statement can use an input value as an iteration count, e.g.,

```
GET LIST((N,(COST(K) DO K = 1 TO N));
```

Moreover, aggregates can be included in the data list, e.g.,

```
DECLARE MIX(40,40) FIXED;
PUT LIST((MIX(*,I) DO I = 1 TO 40));
```

In this example the elements of MIX will be printed out with the first subscript varying most rapidly, as in Fortran. On the other hand, the statement

```
PUT LIST (MIX);
```

with MIX declared as above will print out the elements of MIX with the last subscript varying most rapidly (since this is the implicit order in storage, as defined by the rules of PL/I).

List-Directed Input-Output

The GET LIST-statement reads an unformatted sequence of items from the input stream. The data to be read consists of a list of items separated by blanks or commas, e.g.,

```
23,47,'DAVID DAVIS'
```

Since the dataset is left positioned after the last character that was read, it is possible to read items from the same line using several separate GET LIST-statements in succession. The successive items on the dataset are assigned to the successive items in the data list. If the data list contains an aggregate, then enough items are read from the dataset to fill the aggregate. The dataset can also indicate empty items by means of two commas in a row. For instance, the statement

```
GET LIST(M1,M2,M3);
```

applied to the input

```
22,,891
```

will cause M1 to be set to 22, M3 to be set to 891, and M2 to be left unchanged.

The types of the items read from the dataset need not agree with the types of the items in the data list; if there is any disagreement, the item read is converted to the type of the item in the data list. For instance, the input value for a float variable can be written as an integer. A character-string item on the dataset need not be quoted unless it contains a comma or blank or starts with a quote. Thus

```
THIS ISN'T BAD
```

can be read into a list of three character variables, and the variables will receive the strings exactly as written. However, if the dataset contains the item

```
'ISN''T'
```

the usual rules for interpreting a character-string constant will be applied and the receiving variables will receive the value

```
ISN'T
```

The PUT LIST-statement writes a sequence of items onto the specified output file. Successive items are placed at successive tabstop positions, and when a line is filled (as

defined by the linesize for the dataset) a new line is started. If the output file is not a print file, however, items are separated by single blanks rather than placed at tabstops. Thus the effect of

```
PUT SKIP LIST((N DO I = 1 TO 10));
```

might be to print the lines

```
1    2    3    4    5
6    7    8    9   10
```

under some appropriate assumptions about the linesize and tabstop positions. Since a PUT LIST-statement does not force an end of line, several PUT LIST-statements can place output onto the same line.

Data-Directed Input-Output

The GET DATA-statement reads a sequence of variable names and associated values, e.g.,

```
A = 3    B = 12    D = 0;
```

The pairs in this sequence can be separated by either blanks or commas (as with the GET LIST-statement), and the sequence is ended by a semicolon. The GET DATA-statement itself can, but need not, specify a list of variables, e.g.,

```
GET DATA(A,B,C,D,E);
```

or

```
GET DATA;
```

The second form is easier to use, but it has the disadvantage

that it forces a complete symbol table to be included in the compiled program. The items in the input stream need not be given in the same order as the variables in the list. Moreover, variables in the data list can be repeated or omitted in the input stream. Subscripted and name-qualified variables can also be included in the input stream, although name qualifications must be complete. For instance,

```
A(3,7) = 'JOE', A(4,7) = 'SAM', ST.COUV = 19.3;
```

is a valid input line, assuming an appropriate data list.

The PUT DATA-statement writes a list of variables, together with their values, onto the output stream. As with the GET DATA-statement, the PUT DATA-statement need not contain a list. Execution of the statement

```
PUT DATA;
```

causes the values of *all* printable variables to be written onto the output stream. Otherwise, the listed variables are written out. If any of these variables are aggregates, the scalar elements of the aggregate are written out, using fully-qualified names and appropriate subscripts. Unlike the other two forms of the PUT-statement, the PUT DATA-statement cannot include expressions in the data list, since expressions do not have names. If the output file is a print file, successive items are placed at successive tabstops. Thus, assuming appropriate stored values and tabstops, the effect of the statements

```
DECLARE
   1 AGG(3)
     2 (RED,BLUE) FIXED DECIMAL(2);
PUT DATA(AGG);
```

is to print out the lines

```
AGG.RED(1) =  4   AGG.BLUE(1) = 8   AGG.RED(2)  = 0
AGG.BLUE(2)= -5   AGG.RED(3)  = 7   AGG.BLUE(3) = 1;
```

Edit-Directed Input-Output

Edit-directed input-output is accomplished through the GET and PUT EDIT-statements. For edit-directed input-output, the transformation between the internal and external forms of the data is governed by a *format list*. When an item in a data list is read, a format is obtained from the format list and used to transform the character representation of the item, as it appears in the input stream, into a stored value. When an item in a data list is written, a format is used to transform the value of the item into a sequence of characters to be inserted into the output stream. For example, the statement

```
GET EDIT(I,J) (F(4),F(3));
```

applied to the input stream

```
ØØ25.00ØØØ3.14
```

causes the variables I and J to be assigned the values 15 (obtained from the first four characters of the stream) and 3 (obtained from the next three characters of the stream). Similarly,

```
PUT EDIT(25,3.142) (2F(7,2));
```

causes the characters

```
ØØ25.00ØØØ3.14
```

to be placed into the output stream. In this case, the 2 in front of the format item F(7,2) indicates that the item is to be repeated twice.

The available formats are listed in Table 12. There are two kinds of formats: control formats and data formats. Control formats control positioning of the dataset; they cause skipping of information on input, and generation of blanks, linemarks, pagemarks, and carriage-returns on output. When a format list is interpreted, control formats are executed as they are encountered; a control format does not use up an item from the data list. A data format, on the other hand, requires a corresponding item in the data list, and the format item together with the data item determines what action is to be taken.

When the data list is exhausted, interpretation of the format list ceases. For instance, execution of the statement

```
PUT EDIT(J) (E(9),SKIP);
```

does not cause the control format SKIP to be executed, since after J is paired with E(9) the data list is exhausted and execution of the statement is complete. If the format list is exhausted while items still remain in the data list, then interpretation of the format list starts over again from the

Table 12. Format Types

(1) Data Formats

A(*w*)	alphanumeric with field width *w* (*w* can be omitted on output)
B(*w*)	bitstring with field length *w*
B1(*w*)	(*w* can be omitted on output;
B2(*w*)	B1 indicates base 2, B2 indicates base 4,
B3(*w*)	B3 indicates base 8, and
B4(*w*)	B4 indicates base 16)
F(*w*,*d*,*s*)	fixed with field width *w*, *d* digits to right of decimal point, scaling *s* (*d* and *s* are optional)
E(*w*,*d*,*s*)	float with field width *w*, *d* digits to right of decimal point in mantissa, *s* digits in mantissa (*d* and *s* are optional)
P *pic*	pictured according to picture *pic*
C(*f1*,*f1*)	complex with real part formatted using *f1*, imaginary part formatted using *f2* (*f1* assumed the same as *f1* if *f2* omttted)

(2) Control Formats

X(*w*)	blank or ignore field of width *w*
COLUMN(*n*)	continue reading or writing at column position *n*
TAB(*n*)	skip *n* tabstops
SKIP(*n*)	skip *n* lines
LINE(*n*)	position at *n*th line of printed page
PAGE	start new printed page

(3) Remote Format

R(*ref*)	use format obtained by evaluating reference *ref*

Table 13. Examples of Input Formats

Input Field	Format	Value
TWO␣CATS	A(8)	TWO␣CATS
1011	B(4)	'1011'B
ONE	B(3)	none -- CONVERSION raised
␣␣1011␣	B(7)	'1011'B
2437	B3(4)	'010100011111'B
2437	F(4)	2437
␣2437␣	F(6)	2437
␣2437␣	F(6,1)	243.7
.2437	F(4,1)	.2437
2437	F(4,1,2)	24370
CATS	F(4,1,2)	none -- CONVERSION raised
2437E1	F(6)	none -- CONVERSION raised
2437	E(4)	2437E0
2437E-2	E(7)	24.37E0
2437	E(4,1)	243.7E0
2437E-2	E(4,1)	2.437E0
2.437E-2	E(5,1)	243.7E0
2437E-2	E(4,1,1)	2.437E0
␣␣␣	F(3)	0
␣␣␣	E(3)	0
␣␣2437	P '(6)Z'	2437
2437␣␣	P '(6)Z'	none -- CONVERSION raised
2437	C(F(2))	24+37I
-2437	C(F(3,1),F(2))	-2.4+37I

Table 14. Examples of Output Formats

Value	Format	Output Field
DYNASTY	A	DYNASTY
2.48	A	ØØ2.48
DYNASTY	A(6)	DYNAST
DYNASTY	A(10)	DYNASTYØØØ
'1011001'B	B	1011001
'1011001'B	B3	544
'1101001'B	B4	D2
'1101001'B	B4(5)	D2ØØØ
'1101001'B	B4(1)	D (and STRINGSIZE raised)
2.48	F(6)	ØØØØØ2
2.48	F(6,1)	ØØØ2.5
2.48	F(6,3,1)	Ø0.248
-25	F(2)	none -- SIZE raised
24.86	E(13)	ØØØ2.486E+001
24.86	E(13,0)	ØØØØ2486E-002
-24.86	E(13,1,3)	ØØØ-24.9E+000
2.48	P '$$$V.99'	Ø$2.48
17.9+6I	C(F(6,1))	ØØ17.9ØØØ6.0
-17.9+6I	C(F(6,1),F(6))	Ø-17.9ØØØØØ6

beginning. For instance,

```
DECLARE A(*) FLOAT;
PUT EDIT(A) (SKIP, 6E(14,5));
```

causes the contents of the array A to be written out with six values on a line. As many lines as necessary are used.

A format list can contain repeated items and repeated lists. The repetition counts and parameters in a format list are not limited to constants; they can be arbitrary expressions, as in the format list

```
((M)E(14,11-K),(N)(A(L),X(5),A(L+1)))
```

Repetition counts that are not integer constants must be written in parentheses.

An edit-directed statement can have more than one pair of data lists and format lists, e.g.

```
PUT EDIT(A(*,3)) (SKIP,10E(14,DECPT))
        (A(*,4)) (SKIP,6E(14,DECPT);
```

The behavior of input formats applied to various data values is illustrated in Table 13. Except for the P-format and the C-format, all of the input formats specify a field width, which is the number of characters to be read from the input stream. (The P-format and C-format specify the field width implicitly.) For the B-formats, the E-format, and the F-format, the numerical value to be read can have leading and trailing blanks, which are ignored. Linemarks within a field are ignored, so that a field can be split over two lines. For the F-format and the E-format, the second parameter, if present,

indicates where an implicit decimal point should be inserted if none appears explicitly in the input. For the F-format, the third parameter specifies scaling to be applied to the input value, while the third parameter of an E-format is ignored.

The behavior of output formats is shown in Table 14. The output formats have behavior inverse to the input formats. The rules are not entirely inverse, however, since the input formats assign meaning to certain fields that cannot be produced by the corresponding output formats. For instance, an output F-format cannot produce trailing blanks, but an input F-format can read trailing blanks. Moreover, unscaled output fields always have explicit decimal points unless they are integers, but unscaled input fields may have implicit decimal points. Another difference is that on output, the A-format and the B-formats need not contain field widths, since a field width can be deduced by converting the output value to character type or to bit type as appropriate.

A remote format can appear in a format list, either as the only item in the list or in combination with other items. The format item R(*ref*) is interpreted by evaluating the reference *ref*, which must be format-valued, i.e., it must evaluate to the statement-name of a FORMAT-statement. The contents of that FORMAT-statement are then interpreted. For instance, the effect of the statements

```
     GET EDIT(A,B,C) (F(3),R(FF),F(2));
FF:  FORMAT(F(5));
```

is the same as that of the statement

```
GET EDIT(A,B,C) (F(3),F(5),F(2));
```

Remote formats are useful when the same format is to be used in a number of different PUT EDIT or GED EDIT-statements. The same format can be used for both input and output.

BIBLIOGRAPHY

1. American National Standards Institute, "American National Standard - Programming Language PL/I," Report ANSI X3.53-1976, New York.
2. Beech, D., A Structural View of PL/I, ACM Computing Surveys (2)1, March 1970, pp. 33-64.
3. Beech, D., and Marcotty, M., Unfurling the PL/I Standard, SIGPLAN Notices (8)10, October 1973, pp. 12-43.
4. Frieburghouse, R., The MULTICS PL/I Compiler, 1969 Fall Joint Computer Conference (35), AFIPS Press, Montvale, New Jersey, pp. 187-199.
5. Honeywell Information Systems, Inc., "PL/I Language Manual," Cambridge, Massachusetts, 1974.
6. IBM Corporation, "PL/I Language Specifications," Number GY33-6003-2, 1970.
7. Lucas, P., and K. Walk, On the Formal Definition of PL/I, Annual Review in Automatic Programming (6)3, 1969, Pergamon Press, pp. 105-181.
8. Pollack, S., and Sterling, T., "A Guide to PL/I," Second Edition, Holt, Reinhart and Winston, New York, 1976.
9. Radin, G., and Rogoway, H., NPL: Highlights of a New Programming Language, Communications of the ACM (8)1, January 1965, pp. 9-17.