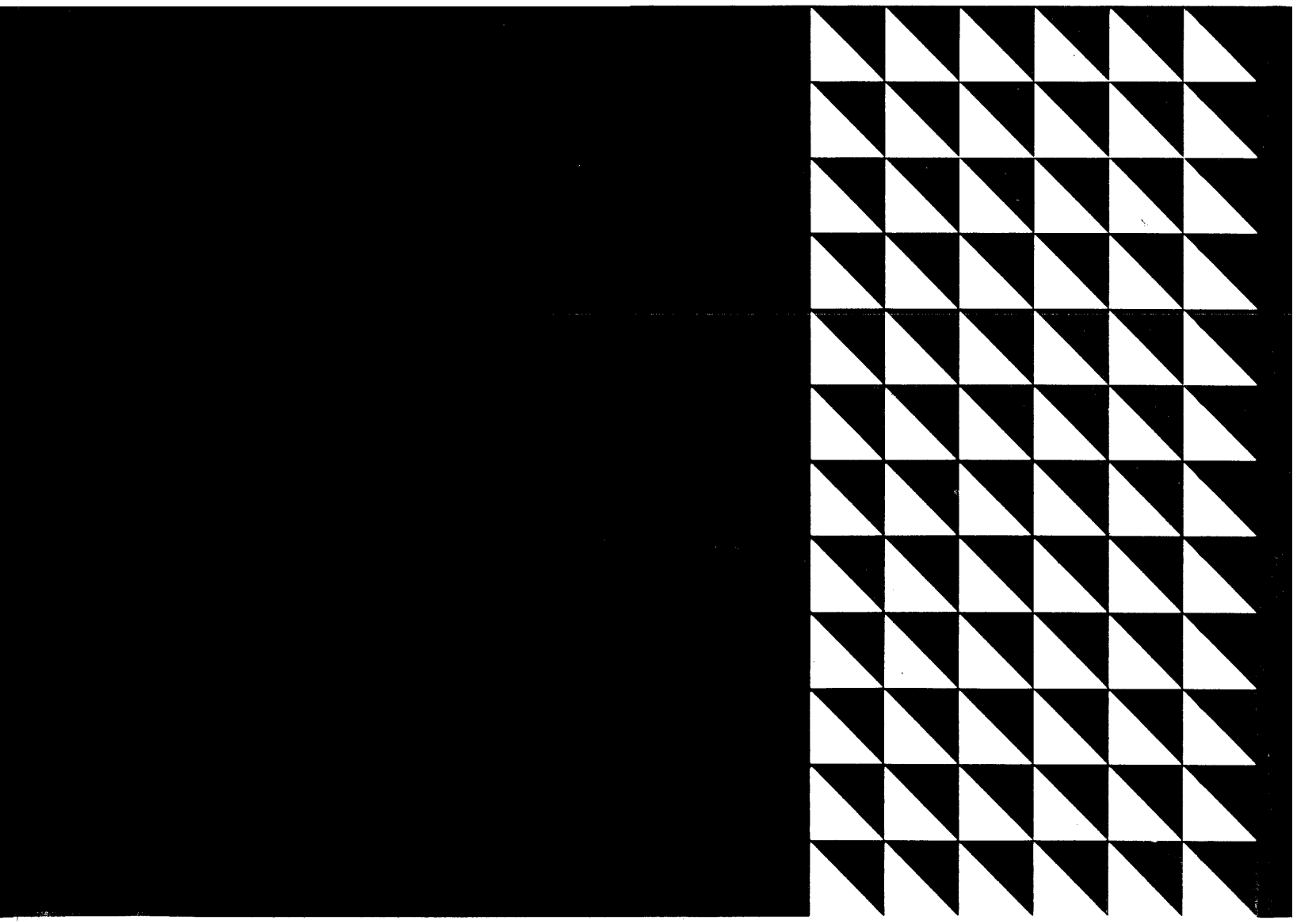
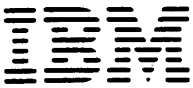
IBM

A Guide to PL/I for FORTRAN Users



Student Text



A Guide to PL/I for FORTRAN Users

Student Text

Preface

This manual is an introductory guide to PL/I written especially for those who have a working knowledge of FORTRAN II or IV. No particular machine implementation of FORTRAN or PL/I has been assumed. Part 1 gives a broad survey of PL/I. A sample program illustrating some of the principal features of the language is explained step by step.

Part 2 gives sufficient detail for the user to be able to write a straightforward program for himself. It is not a rigorous exposition. Examples have been used to clarify the text. The terminology used is intended to be that which is familiar to a FORTRAN user.

Part 3 describes concepts not familiar to FORTRAN users, although some of them are familiar to those who know COBOL or ALGOL. Where references to COBOL or ALGOL will help readers who know these languages, the reference is made, but the explanation is also given in full for those who do not.

This guide does not attempt to cover all the features of the language. Many facilities have not been mentioned at all, and some of the statements and features have not been explained in full detail.

Other related publications are "A PL/I Primer," Form C28-6808, "A Guide to PL/I for Commercial Programmers," Form C20-1651, and IBM System/360; PL/I Reference Manual, Form C28-8201.

Major Revision (May 1968)

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. Address comments concerning the contents of this publication to IBM Corporation, DPD Education Development - Publications Services, Education Center, South Road, Poughkeepsie, New York 12602.

Contents

Why a New Language?

FORTRAN began as a very early experiment in efficient compilation of an algebraic language. It uses the conventions of algebra to specify computation on floating-point numbers. A few simple control statements, DO, IF, and GO TO, specify the flow of control. Input and output statements are a compromise between simplicity of use and flexibility of format.

FORTRAN has proved such an effective compromise between generality and efficiency that it has become the most widely accepted higher-level language in use today. It has progressed from a language for occasional use by amateur programmers to the major language in use in most scientific installations. This acceptance has been possible because FORTRAN has developed from the early single compilation FORTRAN I, to FORTRAN II with subprogram facilities, and FORTRAN IV, which includes improved logical facilities, more data types, and better facilities for storage assignment and initialization.

A joint SHARE and IBM committee was set up in 1963 to investigate what modifications should be made to FORTRAN to make it as useful in the newly developing computer environment as it had been in scientific and engineering installations in the past.

This committee reached the conclusion that they could cater to a wider variety of users and applications if they abandoned the conventions of FORTRAN. The main reason is that FORTRAN was designed to operate most efficiently in installations performing scientific and engineering calculations on a computer that uses cards and magnetic tape for input and output. This type of environment is becoming less common. Scientific computing centers now have a number of programmers who are concerned with activities outside the scope of FORTRAN, such as systems programming; and scientific programmers are increasingly concerned with the problems of efficient input/output operation and the generation of reports.

PL/I is designed to meet the needs of all these kinds of users. For many installations, even with a wide range of scientific and commercial applications, it will be the only language used. The language includes the ability to handle character, bit, and numeric data as well as the floating and integer arithmetic of FORTRAN. Input/output statements provide for the detailed control of physical records and files, and for efficient use of data channels.

Besides meeting the needs of a wider range of users, PL/I is designed for efficient use with a wider range of machines. Most modern computers have an interrupt system; PL/I recognizes this and provides statements to use the interrupt facility. An increasing proportion of machines are used with an operating system, and the concurrent performance of tasks is becoming more common. PL/I is designed with this type of machine and operating system in mind.

Another feature of PL/I is that it is possible to teach and use subsets of the language. For example, an open-shop scientific user can program using a short and simple manual, without being concerned with the complex facilities that might be required by a programmer who wanted to use PL/I to write an operating system. This feature is essential if the language is to be used by programmers of widely varying interests, experience, and ability.

It was felt that the advantages to be gained from having a single higher-level language which could be used for all the activities of an installation, which would work efficiently in a modern environment, and which would be suitable for all varieties of programming, were sufficiently great to justify the launching of a new programming language.

Among the advantages of PL/I are:

1. Better integration of sets of programs covering several applications.
2. Easier interchangeablility of files.
3. Fewer programs to be written in a machine oriented language.
4. Reduced training costs; fewer people to learn a machine oriented language.
5. Better machine utilization, greater use of channels, more flexible storage assignment, better interrupt handling.

Features of the Language

In PL/I, as in FORTRAN, computation is specified by arithmetic expressions designed to resemble algebraic expressions. Statements such as IF, GO TO, and DO appear in both FORTRAN and PL/I, and have similar but not identical meanings. The PL/I statements in most cases have a more general construction and a wider range of meaning than their FORTRAN equivalents. Input and output are specified by READ or WRITE and GET or PUT statements, the format-

directed input/output statements in FORTRAN corresponding to one of three possible types of GET and PUT statements in PL/I.

Several restrictions imposed by FORTRAN are absent from PL/I. Statements may be written in free format and are not affected by card boundaries. The semicolon is used to terminate statements. Names, and not numbers, are used to refer to statements and files. The same set of rules is used for constructing all names, whether for statements, functions, files, or varibles. There are many more different kinds of data, and data types may be mixed within expressions. In nearly all cases where a single variable is allowed, an expression may be used. This eliminates the limitations on subscript expressions and the restrictions on the elements in a DO statement which exist in FORTRAN.

Several features of the language permit more effective use of the computer at object time. Storage allocation may be regulated so that storage is assigned only when it is actually required by the program. This is in contrast to most systems where every variable has its storage assigned at load time and has sole use of that storage throughout the program. Language features are provided that permit the programmer to make use of the more sophisticated machine features now available, such as interrupts and asynchronous operation. These features are described in Part 3 of this guide.

The range of input/output facilities has been extended beyond those available in FORTRAN. GET and PUT statements may be used with a format list to provide the equivalent of FORTRAN READ and WRITE. However, it is also possible to omit the format specification and have them deduced from the data. In addition, there are READ and WRITE statements in PL/I to provide a wide range of I/O operations including the transmission of records without conversion, the handling of devices with various techniques of organization and access, and the use of variable length and blocking.

The Structure of a PL/I Program

A strength of FORTRAN is its ability to combine separately compiled subprograms into a single program, but it is usually unable to specify a subprogram within a compilation, except in the limited form of an arithmetic statement function. Consequently, subprograms of a few statements are rare, and if a programmer wishes to achieve the effect of a subprogram without incurring the overhead of a separate compilation and copying a set of COMMON and DIMENSION statements, he uses a device such as the assigned GO TO.

PL/I allows more than one subprogram to be compiled as a unit, but incorporates the FORTRAN feature of combining separate compilations. In PL/I a subprogram is called a procedure, whether it is the equivalent of a function subprogram or a subroutine subprogram.

A procedure may contain other procedures within it. Procedures are nested in a manner similar to DO loops, and each procedure must be completely contained within the next higher level of the nest. As with arithmetic statement functions in FORTRAN, inner procedures may make use of variables specified in outer procedures without requiring explicit communication such as the use of argument lists or COMMON. An internally nested procedure in PL/I is not limited to a single statement, as the FORTRAN arithmetic statement function is, and so has a much wider range of usefulness.

Another feature of the language is that for the purpose of the flow of control in a program, groups of statements may be bracketed together. The pairs of statements DO...END; BEGIN...END; PROCEDURE ...END; act as brackets to enclose a group of statements. The second two pairs may include declaration statements that control the assignment of storage. The use of the PROCEDURE and BEGIN statements is illustrated in the example below.

Sample Program: Matrix Inversion

The following example has been chosen to illustrate some of the features of PL/I. It is a modification of the Gaussian elimination method of matrix inversion, which inverts a matrix in its own space.

It is certainly not meant to be a good example of an inversion program, nor is it necessary to follow the algorithm in detail to understand the way the language is used. The individual statements are explained in the commentary that follows the program; the commentary will enable the reader with a basic knowledge of FORTRAN to understand, in general, the function of each statement.

The principal features illustrated are:

1. The ability to assign storage, with variable dimensions at object time, and initialize it.
2. The use of * in a subscript position, to indicate evaluation of the expression for all values of the subscript.
3. The extension of the DO statement to control a loop from 1 to J−1 and then from J+1 to N.
4. The use of list-directed input to avoid writing FORMAT statements.

```
1     INVERT: PROCEDURE OPTIONS (MAIN);
2      DECLARE HEADING CHARACTER (20), RIGHT FIXED;
3      GET LIST (LEFT, RIGHT); /* ITEMS MAY BE ANYWHERE IN RECORD, SEPARATED
                                  BY BLANKS. "LEFT" AND "RIGHT" WILL SPECIFY NUMBER OF PLACES
                                  TO LEFT AND RIGHT OF DECIMAL POINT. */
4      DELTA = .5/ (10** RIGHT);
5      GET LIST (HEADING,N) ; /* NAME AND ORDER OF MATRIX */
6      PUT PAGE LIST (HEADING || ' ROW COL ELEMENT');
7      MATIN: BEGIN; DECLARE A(N,N); A=0 ; /* STORAGE FOR MATRIX A IS
                                                ASSIGNED AND INITIALIZED AT THIS
                                                POINT */
10           ON ENDFILE (SYSIN) GO TO INVERSION;
12           INPUT: GET LIST (I, J, A(I,J)); GO TO INPUT;
             /* ONLY NON-ZERO ELEMENTS ARE READ. END OF FILE ENDS READ */
14           INVERSION: ON ZERODIVIDE BEGIN;
16                              PUT LIST('ZERO ON DIAGONAL'); STOP;END;
19           PIVOT: DO K= 1 TO N; D=A(K,K) ; A(K,K)=1 ;
22                PIVOT_ROW: A(K,*) = A(K,*) / D;
23                NON_PIVOT: DO I = 1 TO K-1, K+1 TO N;
24                           D = A( I,K ) ; A(I,K) = 0;
26                           INNER_LOOP:        A(I,*) = A(I,*) - D * A(K,*) ;
27           END PIVOT;
29           OUTPUT: DO I = 1 TO N;    DO J = 1 TO N;
31                   IF ABS ( A(I,J)) > DELTA THEN
32                                        PUT SKIP(2) EDIT(I,J,A(I,J))
                                         (F(23),F(5),F(LEFT+RIGHT+4,
                                                      RIGHT)) ;
33           END OUTPUT;
35     END INVERT;
```

(The numbers at the left are not part of the PL/I source program, and are included here for reference purposes only.)

5. The use of variable and expressions in FORMAT specifications.
6. The use of the ON statement to handle machine interrupts.

The PROCEDURE statement in the first line names the routine. The DECLARE statement in the next line specifies HEADING as a 20-character string and RIGHT as a fixed point number. The remaining attributes of RIGHT and all attributes of LEFT are defaulted to the system.

The first GET statement reads in two numbers that will later be used to control the format of output. A typical input record might be:

```
5 4
```

The word LIST specifies that the input is list-directed. Numbers appear as constants and are separated by commas or blanks.

Comments, which may occur in any position where blanks are allowed, are enclosed in brackets /*---*/.

DELTA is the smallest number that will appear on the output file; numbers that would appear as a zero are not written. The next GET statement reads in the heading associated with the matrix and the order of the matrix. The data for this might be:

```
'DX20 STIFFNESS' 48
```

The PUT statement on line 6 causes an eject to a new page, and the heading to be printed followed by the words ROW, COL, and ELEMENT.

The BEGIN statement isolates declarations in this block from the block containing it. The storage for the matrix is assigned. The statement, A=0, means that each element of the array, A, is to be set to zero.

The next GET statement reads in each element, preceded by its row and column numbers. A typical input record might be:

```
3 5 17.7
```

which would cause a value of 17.7 to be read into A(3,5).

The standard input unit (SYSIN) will be read since no FILE name was include in the GET statements. The ON statement on line 10 sets the action to be taken when the end-of-file is encountered on SYSIN. The GET statement on line 12 is repeatedly executed until then, when a transfer is made to the block labeled INVERSION, on line 14.

This next ON statement says that if an attempt is made to divide by zero, the program is to type a message and abandon the job.

The first DO statement sets up the outer loop of the calculation. In PL/I the commas of the FORTRAN DO statement are replaced by TO and BY. The DO statement does not itself specify the range of the loop; that is done by the terminating END statement, END PIVOT. The underscore may be used within a variable name to improve readability.

The statement PIVOT_ROW performs the loop that divides the pivot row by the pivot. The * in a subscript position specifies that the operation is to be repeated for all values of the subscript. It is equivalent to:

```
DO J=1 TO N; A(K,J) = A(K,J)/D; END;
```

The statement NON_PIVOT is an example of more than one looping specification in a DO. The loop is executed first for I=1 TO K–1, then for I=K+1 TO N.

The statement INNER_LOOP uses * in a subscript position in the same way that PIVOT_ROW used it earlier, to operate on a row at a time.

The END PIVOT statement ends all the DO loops included between the statement PIVOT : DO--; and this statement. In PL/I, an END statement does not terminate a compilation.

The loop OUTPUT tests each element. If its absolute value is large enough to print as a nonzero number, it is written on the standard output file. The statement END INVERT ends the procedure, and also ends the block MATIN.

Character Set

PL/I uses a larger character set than FORTRAN. Besides the set of special characters used by FORTRAN, nine additional characters appear in PL/I statements. Their use simplifies the rules for constructing statements, and improves the appearance of source programs.

However, since many installations will wish to make use of equipment with a smaller character set, a conversion to the FORTRAN character set from the larger PL/I set is specified.

The nine additional PL/I characters are shown below, together with the alternative representation suitable for use on equipment with only the smaller, 48-character set.

PL/I	NAME	EQUIVALENT IN 48-CHARACTER SET
:	colon	..
;	semicolon	,.
&	and	AND
|	or	OR
¬	not	NOT
>	greater than	GT
<	less than	LT
_	underscore	no equivalent
%	percent	//

In addition some combinations of these special characters have a special representation in the FORTRAN character set.

PL/I	NAME	EQUIVALENT IN 48-CHARACTER SET
>=	greater than or equal to	GE
<=	less than or equal to	LE
¬=	not equal to	NE
¬<	not less than	NL
¬>	not greater than	NG
||	concatenation operator	CAT

When the nature of these translated characters could result in their being confused with the preceding and following characters, they should be separated from them by blanks. For example, in the 48-character set X>Y should be written X GT Y.

Basic Syntax of PL/I

Under this heading are included the rules for forming symbols and statements, and for using separating symbols such as the comma.

FORTRAN assumes that the card is the normal method of input for source programs. With the growth in the use of remote terminals and other forms of Tele-processing, this assumption is no longer valid. In PL/I a program is treated as a continuous string of characters. The boundaries of physical records are ignored, and there is no need for a continuation column. Punctuation marks such as the comma and semicolon are used to separate statements and parts of statements.

PL/I uses the same basic components as FORTRAN for writing programs. A program may consist of a number of separately compiled subprograms, each consisting of a number of statements. Both the program and subprograms are known as procedures. A statement consists of a collection of symbolic names, operators, keywords such as DO (which have a special meaning to the compiler), and separators such as commas and parentheses. A glance at the sample program in Part 1 will show that much of the program construction is familiar.

One immediately obvious difference between PL/I and FORTRAN is that statement numbers are not used in PL/I. A statement number is merely a way of identifying a statement. PL/I has the rule that all words (or identifiers) are constructed in exactly the same way. Thus the rules for forming names for statements, files, and keywords are the same as those for forming variable names.

The rules for constructing a name are similar to the FORTRAN rules. The first character must be alphabetic, and the remaining characters, if any, may be alphabetic or numeric. The language definition limits the length of a name to 31 characters; implementations may make further restrictions. Because PL/I does not ignore blanks, as most versions of FORTRAN do, it is not permissible to embed blanks in a name. However, to improve the clarity and mnemonic significance of names it is permissible to include the special character _ (underscore) within a name when using the larger character set.

The names BIG_A, JOB_NO, and TEMP_1 are valid PL/I symbols, as are BIGA, JOBNO, and TEMP1. The underscore is a part of the name, so BIGA is not the same as BIG_A. BIG A is invalid since it contains a blank within the symbol.

Since the end of the card is no longer available to end statements, the semicolon is used. Statements may be punched more than one to a card, or may

extend over several cards; the semicolon marks the end of a statement. The label (which takes the place of the FORTRAN statement number) is separated from the remainder of the statement by a colon.

PL/I contains many more keywords than FORTRAN, since it expresses a much wider variety of operations. It would be contrary to the philosophy of PL/I if the programmer had to learn all these words whether or not he intended to use them. PL/I therefore allows the programmer to use keywords as names for his own use – for example, as variable names or as statement labels. The word IF may therefore exist in the same program both as a name and as a keyword. Similarly, the programmer may use the name KEY without knowing, or being affected by, its special meaning to the compiler when used as an option in a READ statement.

Another consequence of the fact that many key words and options can also be used as names, is that the compiler cannot rely on programmer names being separated by special characters or reserved words. The language therefore requires that where words do appear without a special character separating them, they should be separated by one or more blanks.

Parentheses are used, as in FORTRAN, for containing subscripts and other lists of names, such as data and format lists. The names of variables in a list are separated by commas. Attributes (which describe variables, procedures, files, etc.) are separated by blanks and not commas. This is illustrated by the following example:

```
DECLARE A(M,N) FLOAT COMPLEX;
```

This statement specifies that A is an m by n array of complex floating point numbers. The name A may be followed immediately by the left parenthesis (although blanks are allowed). The names M and N in the subscript list are separated by a comma. The attributes FLOAT and COMPLEX, which describe A, are separated by blanks.

```
STEP: A=X+Y;
```

is a labeled statement. The label is separated from the remainder of the statement by a colon. Other examples of PL/I statements are:

```
IF A > B THEN C = A**2; ELSE C = B**2;

DO J = 1 TO M*N+1 BY K;

ALPHA: K2 = Z**2 + X/I;

A(I,J) = (B(I+1,J) + B(I-1,J))
+(B(I, J-1) + B(I,J+1))*.25;

L = COS (OMEGA) *COS(U) — SIN(OMEGA)
*SIN (U) *COS(C);

CALL CONVERT (X,DEG,MIN,SEC);
```

Data Types

FORTRAN II originally recognized two types of variables: integer and floating. This was extended in some versions of FORTRAN II to include double precision, complex, logical, and character data. PL/I provides an even wider range of data types, and a wider range of operations on character and bit data. At the same time, by starting afresh, it has been possible to classify this wider range in a more logical way.

FORTRAN is concerned mainly with engineering calculations, where numbers can conveniently be represented in floating point since engineers are concerned primarily with precision and not with the form in which the number is represented. However, some computer users are concerned with the particular way in which numbers are handled. For example, in some financial problems conversion of decimal numbers to binary floating point is not acceptable. PL/I gives the user the ability to declare in detail the way in which arithmetic data shall be stored and used in computation. If the user is not interested in this level of detail, he omits declarations, and his variables assume a standard form.

PL/I recognizes the following characteristics of numeric information:

```
Base: which can be BINARY or DECIMAL
Scale: which can be FIXED or FLOAT
Mode: which can be REAL or COMPLEX
```

An arithmetic variable will have one each of base, scale, and mode attributes. The language also allows the programmer to choose precision, although in most implementations the choice will be limited by the machine available.

Data Declaration

One of the consequences of having a variety of data types is that it is necessary to be able to specify the attributes of a variable. In PL/I this is done by the DECLARE statement, which combines the functions of the various specification statements of FORTRAN. Besides the type statements of FORTRAN IV (REAL, INTEGER, DOUBLE PRECISION, COMPLEX, LOGICAL, and EXTERNAL), DECLARE includes the functions of DIMENSION and DATA, and replaces the functions of COMMON and EQUIVALENCE. The principle of the DECLARE statement is that all the attributes of a variable should be specified in a single statement.

To avoid undue repetition, names with the same attribute may be grouped together by parentheses.

Examples of DECLARE statements specifying attributes of arithmetic variables follow:

```
1. DECLARE ALPHA REAL FLOAT BINARY,
   BETA COMPLEX FLOAT BINARY;
2. DECLARE (ALPHA REAL, BETA COMPLEX)
   FLOAT BINARY;
```

NOTES:

Example 1. A variable name is followed by its attributes, separated by blanks. The name, with its attributes, is separated from the next name in the list by a comma.

Example 2. Parentheses are used to indicate that the attributes FLOAT and BINARY apply to both ALPHA and BETA. This example is equivalent in meaning to the preceding one.

Precision

The language allows the user to specify the precision of data, together with its other attributes, in a DECLARE statement. The precision specification constitutes a constant, or constants, enclosed in parentheses immediately following a base, scale, or mode attribute. If the variable is binary, the precision is specified in bits; if decimal, in decimal digits. In PL/I, fixed-point variables are not restricted to integers. The precision of any fixed-point number is specified by one or, more usually, two numbers. The first (w) indicates the total number of bits or digits in the variable. The second (d) is a scale factor indicating the number of binary or decimal places in the variable. The magnitude of the scale factor need not be less than w, and it may be negative; it may also be omitted altogether, in which case the scale factor is assumed to be zero (that is, the variable is an integer). Where, as is more often the case, both w and d are specified, the two numbers are separated by a comma. Examples of precision specifications in DECLARE statements are:

```
DECLARE A BINARY (20),
B DECIMAL FIXED (6,2);

DECLARE X BINARY FLOAT (48),
Y BINARY FIXED (17,2);
```

A precision specification specifies the minimum number of digits to be used, and in the case of fixed point, the scaling to be performed. The actual number of digits to be held will depend on the implementation.

Implicit Declaration

One of the ground rules of PL/I is that the user need not specify details in which he is not interested. There are therefore some simple rules for determining characteristics that are not explicitly declared:

1. Data is assumed arithmetic unless declared otherwise.
2. If no base is specified, decimal is assumed. (Decimal is chosen because for most people it is a more familiar representation than binary, though, of course, many FORTRAN users are accustomed to binary.)
3. If no scale is specified, floating point is assumed.
4. If no mode is specified, real is assumed.
5. Precision will depend upon the implementation.

An exception to these rules is that if a variable has none of these characteristics declared, and its name begins with one of the letters I through N, it is assumed to be a binary integer. This preserves the useful FORTRAN convention of using names beginning with I through N for indexing and counting. For some machines a different internal representation may be used. For example, not all machines have decimal arithmetic capability, in which case a binary internal representation might be used.

Character and Bit Data

FORTRAN was at first concerned only with numbers. However, it soon became clear that there were occasions when it was necessary to be able to move alphabetic data in order to produce readable reports. This requirement was fulfilled by the introduction of the A format specification, which allowed alphabetic information to be read into arithmetic variables. The alphabetic data could then be moved using an arithmetic statement.

PL/I has a new type of variable, the character string. There are therefore no exceptions to the rule that a variable within a procedure can be of only one data type. Character strings may be of fixed or variable length. They may be moved using the assignment statement, which is the PL/I counterpart of the FORTRAN arithmetic statement; they may also be compared, and character fields may be connected, using the concatenation operator, to form a new field. The operations on character fields will be described in more detail under "Expressions and Assignment Statements."

In FORTRAN, A-type format is used to modify format statements; PL/I achieves this by allowing the FORMAT specifications to be variables, thus simplifying the task of altering them. An I/O statement may also have any one of a number of format statements associated with it. The particular one can be selected at object time without the necessity of having multiple I/O statements. Character data can be held in variable character strings that can be easily manipulated. FORMAT statements are described under "Input/Output."

Character strings must be described in a DECLARE statement; examples of DECLARE statements for character variables are

```
DECLARE ALPHA CHARACTER (20);
DECLARE TITLE CHARACTER (80) VARYING;
```

The attribute CHARACTER must be followed by a specification of the length of the string (in characters), or if it is a varying-length string, by the maximum length followed by the attribute VARYING. The length specification is enclosed in parentheses. In Part 3, when dynamic storage assignment is discussed, examples are given of cases where the length specification can be a variable or even an expression.

FORTRAN IV does include the logical variable as a data type, but this is limited to operations with a single bit of information. PL/I deals with a wider class of bit variables, the bit string. As its name implies, the bit string is a sequence of bits; the complete string, which may be of fixed or varying length, may be used in expressions. The permissible operations are explained in Part 2, under "Evaluation of Expressions." The logical variable of FORTRAN IV exists in PL/I as a bit string of length 1; the logical constants .TRUE. and .FALSE. are replaced by '1'B and '0'B, which describe bit strings one bit long. The statement:

```
DECLARE (B1,B2,B3) BIT (32);
```

specifies three bit strings, each of 32 bits.

Constants

In FORTRAN the form in which a constant was written specified the form in which it was to be stored. For example, the constant 1 specified a fixed-point number, and 1. specified a floating-point number, the form chosen depending on the expression in which it was used. This led to X=X + 1, a very frequent beginner's error in FORTRAN.

In PL/I, a constant may be written in any way allowable for any type of number. The internal representation that will actually be used depends on the expression in which the constant appears. The details of this are explained in the section on expression evaluation, but a few examples will illustrate the principle; for example, in the expression (J + 1.) the constant would be represented internally as fixed point, but in X + 1 the constant would be floating point. In each case, the context is used to determine the scale and base.

A constant is written as one or more digits, with an optional decimal point. It may be multiplied by a power of 10 by following the constant with the letter E, an optional sign, and one or two digits. To specify floating point, the E and the exponent must appear. The rules correspond to those for FORTRAN, except that blanks are not allowed. Examples of constants are:

```
1
9.
.117
1.1
9.7E-2
```

Only when an E is present is the constant held as floating point. In the above examples, only the last is floating point.

An imaginary constant is a real constant followed by the letter I. Where a complex number is required within an expression, it is represented by a real number followed by a signed imaginary number. Examples of complex numbers appearing in expressions and statements are:

```
A = C*(27.1+17.3I);
B = 1I;
C = 4+7I;
X = 11.2+17.4E+3I;
```

Constants can also be used in character and bit string expressions. A string constant is enclosed in quotation marks. For example:

```
'THIS IS A CHARACTER STRING'
'SO IS * % + $))'
```

To represent a quotation mark or an apostrophe in the string, two quotation marks must be written; for example:

```
'MARY"S LAMB'
```

Bit strings may contain only the digits 0 and 1, and the letter B follows the terminating quotation mark, for example:

```
'1010111'B
```

To specify a string repeated a number of times, it may be preceded by an integer constant in parentheses, for example:

```
(8) '1'B is equivalent to '11111111'B
(2) 'TOM' is equivalent to 'TOMTOM'
```

Initial Values

A programmer will often wish to assign initial values to variables on entry to his program. In FORTRAN this is done using the DATA statement. In PL/I initial values are specified in a DECLARE statement, as one of the attributes of a variable. The statement

```
DECLARE A FIXED INITIAL (0),
        B CHAR(10) INITIAL ('FREQUENCY='),
        (C,D,E,F,G) INITIAL (1E0);
```

would assign a value of zero to the fixed point variable A, assign to variable B the character constant FREQUENCY=, and set all the five elements, C through G, to 1. The initial value attribute is followed by a list of constants enclosed in parentheses. Repetition may be indicated by preceding the constant by a replication factor enclosed in parentheses.

Expressions and Assignment Statements

In PL/I, as in FORTRAN, the foundation of the language is the expression. It is the expression that specifies the computation to be performed. And it is partly because the rules for forming expressions are the same as those of elementary algebra that FORTRAN has achieved its popularity.

Evaluation of Expressions

In nearly all cases, a PL/I expression that is an exact copy of a FORTRAN expression will have precisely the same meaning algebraically. For example:

```
X(I) = F(I) * XFX/(XFX+V*(E(I)-1.))
```

evaluates the same expression in FORTRAN and PL/I and assigns a value to X(I). In PL/I, an assignment statement may have several variables on the left-hand side, all of which would be assigned the value of the expression.

For example:

```
A, B, C=X/2;
```

assigns the value of X/2 to each of A, B, and C.

For arithmetic expressions where all the operands have the same attributes, as is the case with all FORTRAN expressions, the rules for PL/I and FORTRAN are almost identical. Where there are differences, they are designed to make PL/I more consistent with normal algebraic usage.

In FORTRAN, because no two operators may appear in sequence, A**–3 is invalid. In PL/I the prefix plus and minus signs (those that *precede* a single operand or expression) are distinguished from infix operators (those that *connect* two operands). For example, in the expression:

```
-A+B
```

the minus sign is a prefix operator, and the plus sign an infix operator.

The hierarchy of arithmetic operations in PL/I is basically the same as in FORTRAN, with certain exceptions. In PL/I, the prefix operators, just defined, share the highest priority with exponentiation.

In the example A**–3, the – sign is a prefix operator of the same precedence as **, and the operations are, therefore, performed from right to left. The expression is treated as A** (–3), which in normal algebraic convention would be what was expected. Similarly, the expression E** –X**2, in which the minus sign is a prefix operator, would be evaluated as E** (–(X**2)).

PL/I, however, includes many more data types than FORTRAN, and allows much greater freedom in combining types in expressions. In FORTRAN II, only two types of variables exist, and they cannot be mixed. In FORTRAN IV there are five data types, and some combinations of different types are permissible (for example, complex and real). In PL/I, any numeric or string data can be combined in an expression. The consequence of this freedom is that more detailed rules are specified for the sequence in which operations are carried out, and for the order and method of conversion from one kind of variable to another.

The rules for expression evaluation are chosen so that conversions to higher types are not carried out unless necessary, and so that precision can be maintained in the result.

Each operation is considered to be associated with one or two operands. Each operand may either be a single variable, or a subexpression that has previously been evaluated. An arithmetic operator will usually be associated with two arithmetic operands.

Each operand will be either decimal or binary, floating or fixed, and complex or real, and will have a precision associated with it.

From the point of view of the possibility of conversion to a higher type, each arithmetic operation may be considered as producing a single, resultant operand. This resultant operand may in turn become one of the two associated with another operator in a subsequent operation. Thus the evaluation of an expression can be broken down into a number of basic operations, each consisting of:

```
RESULT = operand 1, operator, operand 2
```

If either operand is a constant, there will normally be no conversion at object time. The constant will have been converted at compilation to correspond to the base of the variable. If the constant's scale is explicitly floating — that is, if the constant is written with an exponent Exx where xx is a numeric exponent — the variable will be converted to floating point.

If the operands are both constants or both variables and have different characteristics, either operand may be converted. If either is floating point, the fixed-point operand will be converted to floating point. If either is binary, the decimal operand will be converted to binary. If either is complex, the result is complex, but the real operand is not converted.

In all cases, the precision and significance of intermediate results is maintained.

These rules must be exhaustive for the operations to be fully defined. The main concern for the programmer is that expressions are evaluated as he expects them to be, that he is not restricted by arbitrary conventions, but at the same time that he should not lose efficiency by unnecessary conversions of data.

To illustrate the rules, let us consider the steps in the following example:

```
DECLARE C COMPLEX;
A = X**2 - 2*X*Y + Z/(J+1) + C;
```

①	X**2	No conversion – operation is actually X*X – result is floating
②	2*X	2 is converted to floating at compile time – no object time conversion
③	(2*X) *Y	No conversion
④	J + 1	No conversion – integer result
⑤	Z/(J + 1)	Intermediate integer result (J + 1) converted to floating – result floating
⑥	(X**2) – (2*X*Y)	No conversion
⑦	(X**2 – 2*X*Y) + (Z/(J + 1))	No conversion
⑧	(X**2 – 2*X*Y + Z/(J + 1)) + C	No conversion – result complex

From the programmer's point of view the expression is evaluated as he would expect it to be, and conversion is performed only when needed.

Logical and Bit String Expressions

In FORTRAN II, the arithmetic expression was the only type of expression; FORTRAN IV introduced the logical expression. In PL/I logical expressions and logical variables exist as a subset of bit string expressions.

PL/I has the same set of relational operators as FORTRAN IV, although the expanded character set allows a more mathematical notation

.GT. becomes >
.LT. becomes <
.EQ. becomes =
.GE. becomes >=
.LE. becomes <=
.NE. becomes ¬=

The result of a relational operation is always a bit string of length 1. These operators have a lower priority than arithmetic operators; therefore comparisons are carried out after the arithmetic operations have been performed, as in FORTRAN.

Relational operators may be used to compare operands with any characteristics, and even with different characteristics. If both operands are arithmetic, the conversion rules for operands are the same as for the arithmetic operators + and –. If both operands are character or bit strings, the comparison is carried out from left to right, character by character, or bit by bit. If the strings are of unequal length, the shorter is padded on the right with blanks (in the case of character strings) or with zeros (in the case of bit strings).

Three FORTRAN operators and their PL/I equivalents are shown below:

FORTRAN	PL/I
.AND.	&
.OR.	|
.NOT.	¬

These operators require bit string operands. If other data types are used, conversion to bit string will be performed. The operations are performed bit by bit from the left; if the strings are of unequal length, the shorter is padded on the right with zeros.

Character and Bit String Operations

Two strings may be combined into one by the operator || (concatenation). This operation may be performed only on bit or character strings. If both operands are bit strings, no conversion is performed and the result is a bit string. In all other cases, the operands are converted, where necessary, to character strings. The concatenation operator is used in expressions such as:

```
'ITERATION NO ' ||N
```

which, if the value of N were 153, for example, would produce the character string ITERATION NO 153. Another typical usage would be:

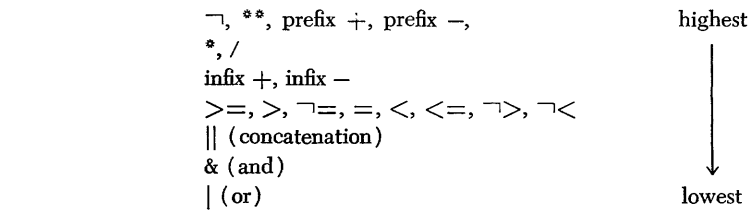
```
FULL_NAME = FIRST_NAME || LAST_NAME;
```

where all three variables are character strings.

In addition to the concatenation operator a number of functions are provided to assist in processing strings. Among them are SUBSTR, which extracts a part of a string, and INDEX, which will locate the position of a character, or substring, within a string. These functions are described in "Appendix 1: Built-In Functions."

Hierarchy of Operations

The hierarchy of operations in PL/I is:



Operations of equal priority are performed sequentially from left to right through an expression *except* in the case of the four operators of highest priority (**, prefix +, prefix –, and ¬), which are dealt with from *right to left.*

Data Aggregates

Types of Aggregates

The concept of an array as a collection of variables all with the same attributes is well known to FORTRAN users. This concept is extended in PL/I to include larger and more varied arrays and to operate on them as units.

In addition, there is a second type of aggregate in PL/I known as a structure. This is a collection of variables each of which has its own name and may have widely different attributes. More will be said about structures in Part 3.

Subscripts and Declaration

As with FORTRAN, an individual element within an array is identified by attaching subscripts to the name of the array. Unlike FORTRAN, subscripts may be an expression and may, in fact, include subscripted expressions. This eliminates another of the more frequent sources of error, particularly for beginners, in FORTRAN. In PL/I, A(3+J) is as valid as A(J+3).

In PL/I, both the upper and lower limits of a subscript can be specified, in contrast to FORTRAN, where the lower limit is always 1. This sometimes simplifies the use of subscripts; for example, an algorithm may more naturally require zero as a starting point. To specify both limits, the DECLARE statement uses either one or a pair of numbers to define the range of a subscript. Examples of DECLARE statements that define arrays are:

```
DECLARE A(10,10), AX(0:99,–2:7);
DECLARE CA (M:X+1, N:X+1,J,K);
```

Where the dimensions alone (as opposed to the actual numeric limits) of an array are specified, they are separated by a comma. Thus, A is an array 10 x 10.

Where the numeric limits are specified, the limits of each dimension are separated by a colon, and the dimensions themselves by a comma. Thus AX is an array 100 x 10, whose first element is identified as AX(0, – 2) and whose last element is AX(99,7).

The DECLARE statement for CA specifies a four-dimensional array with subsequent limits as follows:

```
M to (X + 1)
N to (X + 1)
1 to J
1 to K
```

These dimensions will be calculated at object time (the rules for doing so are discussed under the heading "Variable Dimensions" in Part 3). Note that since the expression (X + 1) is used here as a subscript, if X is floating point the result of (X + 1) will be converted to an integer.

Array Expressions

One of the most successful features of FORTRAN is its ability to specify arrays of variables. FORTRAN, however, does not permit operations on an array as a whole, but only on elements of an array. Part of the power of the DO loop lies in its ability to vary the subscripts of an array name, thereby permitting operations to be performed, in turn, on each element in the array.

PL/I uses arrays similarly to the way in which they are used in FORTRAN but, in addition, allows whole arrays to be used as variables within an expression. For example, if A and B are arrays, then A = B will move each element of B into the corresponding position in A.

The interpretation of an array name appearing in a statement is that the statement is to be performed repetitively, using each element of the array in turn. A restriction on the use of two or more arrays in an expression or assignment statement is that the arrays must have identical dimensions and identical upper and lower bounds. If an array appears on the right of the equal sign, there must also be an array on the left. Some examples of array statements follow.
Given the declare statement:

```
DECLARE A(10,10), B(10,10), C(10,10),
        D(12, 6), E(12, 6), F(12, 6),
        G(12, 12);
```

the PL/I statement:

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

is equivalent to the FORTRAN loop

```
  DO1 I=1,10
  DO1 J=1,10
1 A(I,J)=B(I,J)+C(I,J)
```

The statements:

```
D= .5*E*F+X;
G=0;
```

are equivalent to the FORTRAN statements:

```
D(I,J)= .5*E(I,J)*F(I,J)+X
G(I,J)=0
```

enclosed in DO loops similar to those in the first example.

Note that scalars may be used in array expressions, and also that multiplication of arrays does not correspond to matrix multiplication.

Asterisk Subscripts

Besides allowing a whole array to be used in an expression, PL/I permits the programmer to specify a part of an array — for example, a row or a column, or, in a three-dimensional array, a plane. The notation used is that if * appears as a subscript, the expression is to be evaluated for all values of the subscript. One example of the use of * is in adding one row of a matrix to another. The PL/I statement:

```
A(I,*)=A(I,*)+A(J,*);
```

would add the Jth row of a matrix to the Ith. In FORTRAN, this would require the loop

```
DO 1 K = 1,N
1A(I,K)=A(I,K)+A(J,K)
```

Any number of subscript positions may be asterisks. For example, given A(0:3,0:2), then A(1,*) is A(1,0), A(1,1), A(1,2). A(*,*) represents the entire array and is therefore equivalent to A.

Control Statements

PL/I uses the three principal control statements of FORTRAN: DO, IF, and GO TO. These statements have been altered to remove some of the restrictions of FORTRAN and to improve their power and clarity.

GO TO Statement

Since, in PL/I, the statement number has been replaced by a symbolic label, the GO TO n of FORTRAN becomes GO TO label — for example:

```
GO TO INVERT;
```

To produce the effects of the computed and assigned GO TO in a consistent way, a new type of variable exists in PL/I: the label variable. This variable, which may be in an array, has a label as its value — for example:

```
M = INVERT;
GO TO M;
```

The statement GO TO M has the same effect as GO TO INVERT, and can be used in PL/I similarly to the way in which the assigned GO TO is used in FORTRAN.

If the label variable is subscripted, the destination of the GO TO may be controlled by the subscript. Consider the following:

```
DECLARE SWITCH (4) LABEL;
SWITCH(3) = HERE;
GO TO SWITCH (3);
```

GO TO SWITCH (3) is equivalent to GO TO HERE. The use of a subscripted label variable in the GO TO replaces the computed GO TO of FORTRAN.

To facilitate the use of subscripted label variables for switching, a statement label may be a label variable with a constant subscript. Thus a program might be:

```
L(3) : X= ___;
       Y= ___;
L(2) : IF  ___;
       Z= ___;
L(1) : A= ___;
       CALL P(X,J);
       GO TO L(J);
```

If the value of J is 2 after the return from the CALL, the branch is to the IF statement at L(2).

IF Statements

PL/I has one basic form of the IF statement which is more general than the logical IF of FORTRAN IV, and more readable and less error-prone than the arithmetic IF of FORTRAN II. The components of the IF statement are:

```
IF expression THEN group ELSE group
```

The IF can test any scalar expression. The expression is evaluated and converted to a bit string. If any bit in the result is 1, the THEN group is executed; if all bits are zero, the ELSE group is executed.

The groups following THEN and ELSE can be either single statements or a number of statements bracketed together using the statements DO; and END;. The word DO by itself can be considered to specify a DO loop with only one iteration. The word END is used to terminate groups of statements; it is not the same as the END statement in FORTRAN, which indicates the end of a compilation.

An example of a simple IF statement is:

```
IF X<1E-4 THEN Y=2*X+1;
ELSE Y=A*X**2+X+SIN(X) + 1;
```

which corresponds to the FORTRAN II statements:

```
    IF (X-1.E-4) 1,2,2
2 Y=A*X**2+X+SINF(X) + 1.
    GO TO 3
1 Y=2.*X+1.
3 ----
```

or to the FORTRAN IV statements:

```
    IF (X.LT.1.E-4) GO TO 1
    Y=A*X**2+X+SIN(X) + 1.
    GO TO 2
1 Y=2.*X+1.
2 ----
```

Points to note in writing IF statements in PL/I are: Because the word IF is followed by an expression, not a statement, the word THEN is not preceded by a semicolon. The statements following THEN and ELSE would each be terminated by a semicolon, as usual. There is no semicolon associated with the IF statement as a whole.

The ELSE path of the IF statement may be omitted. For example:

```
IF ABS (X(I,J)) <DELTA THEN X(I,J) = 0;
  X(I,J) = X(I,J) + D;
```

would set an element of an array to zero if its magnitude were below a limiting value DELTA; otherwise, the computation would skip the statement following the word THEN. The program would continue with the statement:

```
X(I,J) =X(I,J)+D;
```

An example of a part of a program to find the roots of the equation $ax^2 + bx + c = 0$, shows some more examples of the use of the IF statement:

```
/*FIND ROOTS OF EQUATION A*X**2+B*X+C = 0
*/
DECLARE (R1,R2,F)COMPLEX;
IF A = 0 THEN LINEAR: R1,R2 = -C/B;
ELSE QUAD:DO;D = B**2 - 4*A*C;
E = -B/(2*A);
  IF D = 0 THEN R1,R2 = E;
  ELSE IF D>0 THEN REAL:DO;
                      F = SQRT(D)/(2*A);
                      R1 = E+F; R2 = E-F; END;
                ELSE IMAG:DO;
                      F = SQRT (-D)/(2*A) *1I;
                      R1 = E+F; R2 = E-F;
END QUAD;
```

NOTES:

1. More than one name may appear on the left-hand side of an assignment, in which case the names are separated by commas – for example, R1, R2=E.

2. The entire quadratic case is enclosed in the statements:

```
QUAD: DO; ...; END QUAD;
```

This forms the ELSE group for the first IF statement.

3. The second IF statement contains another IF statement as its ELSE path. Although this statement contains groups bracketed by DO; and END;, the statement itself does not require an enclosing DO-END, since it counts as a single statement, known as a compound statement.

4. The statement labels LINEAR, REAL, and IMAG are not required but are used to illustrate the fact that the statements following THEN and ELSE may be labeled. They also show how labels can be used as commentary.

5. The END QUAD; statement terminates the groups QUAD and IMAG. An end statement, followed by a label, terminates all the groups nested within the labeled group.

6. The free format is used to emphasize the structure of the program.

7. Comments are enclosed in /* ----- */.

8. As in FORTRAN, the rules for expression evaluation result in the expression for the imaginary F being interpreted as:

```
(SQRT(-D) / (2*A))*1I
```

where 1I is the imaginary constant i. The result F is complex.

9. Though REAL and IMAG are keywords in the language, and have a special meaning, they can also be used by the programmer as labels. The general principle is that the programmer is not hurt by what he does not know.

10. The constants in the floating-point expressions do not need a decimal point.

DO Statements

One use of the word DO, in the IF statement, has already been illustrated. The following example shows some of the differences between the PL/I and the FORTRAN DO statements:

```
MPY:DO I = 1 TO L; DO J = 1 TO N;
    C(I,J) = SUM (A(I,*)*B(*,J));
     END MPY;
```

is equivalent to the FORTRAN statements

```
    DO 1 I = 1,L
    DO 1 J = 1,N
    C(I,J) = 0.
    DO 1 K = 1,M
1 C(I,J) = A(I,K) * B(K,J) + C(I,J)
```

NOTES:

1. In PL/I, the statement END MPY; ends both of the two loops.

2. SUM is a function that returns the sum of all the elements of an array. Its argument is the array expression A(I,*)*B(*,J), whose result is a vector.

3. The first comma in the FORTRAN DO statement is replaced by TO in PL/I.

4. As in FORTRAN, if the increment is not specified, it is assumed to be 1. As in FORTRAN, an increment other than 1 may be specified, in which case the second comma in the FORTRAN DO is replaced by the word BY.

```
DO 99 I = J,K,L
```

becomes

```
DO I = J TO K BY L;
```

In PL/I the initial value, limiting value, and increment may be expressions. Hence, the following DO statements are valid in PL/I:

```
DO I = N TO 1 BY -1;
DO X = 0 TO 10.1 BY 0.2;
DO P = Q(I,J)+1 TO X**2 BY DELTA;
```

A further extension is that there may be more than one specification in the statement, and a specification may be a single value. For example:

```
DO X = 0 TO 4.95 BY .1,
5 TO 10.1 BY .2,
10.5 TO 20.1 BY .5, 50;
Y = SOMEF(X);
PUT LIST (Y,X);
END;
```

would evaluate the function SOMEF for values of X from 0 to 20, with a varying interval. The first specification controls X from 0 to 4.9; when X exceeds 4.95, the second iteration specification controls X from 5 to 10, in steps of .2, and the third from 10.5 to 20 in steps of .5. The final specification causes a single iteration with X=50.

The ON Statement and the Prefix

In FORTRAN, special IF statements are used to test for such conditions as overflow and divide check. However, many modern machines signal exceptional conditions such as these by means of an interrupt. In PL/I, the concept of an interrupt is extended beyond those conditions detected by the hardware of a particular machine, and includes checking for such conditions as subscript out of range, or checking the execution of a particular statement.

Interrupt handling in PL/I may be thought of as follows:

1. A special condition arises. This condition may be a machine condition, such as end of file or overflow; a condition recognized in a subroutine, such as a data conversion error; or a condition requiring that a special code be created so that the condition can be detected, such as a subscript exceeding its declared bounds.

2. If the condition has been enabled, an interrupt occurs; if the condition is disabled, no action is taken. Some conditions are always enabled (for example, ENDFILE); others may be enabled or disabled by a statement prefix.

A prefix may enable a condition within a statement, a procedure, or a begin block. The prefix consists of a condition name, or a list of condition names enclosed in parentheses and separated from the remainder of the statement by a colon. The prefix precedes the remainder of the statement, including the label (if any). If the prefix is attached to a BEGIN or PROCEDURE statement, the condition is enabled throughout the block headed by the statement. For all other statements, the condition is enabled only for the single prefixed statement.

The prefix may also disable a condition. In this case the condition name is preceded by the word NO.

A prefix applies only to the statement or block to which it is attached, and not to other procedures which are called or which appear as function references within the statement or block.

Examples of enabling and disabling are given in the following procedure:

```
(CHECK(L)) : ATTENUATE: PROCEDURE
(T,D,M);
(SUBSCRIPTRANGE) :C1 = ALPHA(M**2+1);
(NOOVERFLOW) :C2 = A**(C1+T);
IF C2 > MAX THEN L: Y=0;
            ELSE Y = D/C2;
RETURN (Y);
END;
```

In this example the prefix CHECK (L) specifies that an interrupt is to occur whenever the statement labeled L is executed. This prefix applies to the whole of the procedure. The prefix SUBSCRIPTRANGE specifies that the subscript (M**2+1) is to be checked to ensure that it lies within the bounds declared for ALPHA. The overflow condition is enabled by default throughout the procedure, except for the assignment statement C2=..., where it is disabled by the prefix NOOVERFLOW.

A full list of conditions and default enabling or disabling is given in Appendix 2.

3. The ON statement specifies the action to be taken when the interrupt occurs; if no ON statement has been executed, a standard system action is assumed. The ON statement has the following form:

```
ON condition-name SNAP on-unit.
```

The word SNAP may be omitted. If used, it specifies that when the interrupt occurs, debugging information relevant to the status of the program at the time of the interrupt is listed on a debugging file.

The on-unit may be a statement or a BEGIN block. The only way in which control can pass to the unit is through an interrupt. When execution of the unit is complete, control returns to the machine instruction following the one that caused the interrupt.

When an ON statement is executed, it remains in force until the completion of the block in which it appears; it also remains in force during the execution of procedures or blocks entered from that block. An ON statement in a subsequent block can temporarily override an ON given in a previous block: on the completion of the later block, the previous ON will be restored.

The effect of an ON statement can also be canceled by a REVERT statement, which must be in the same block as the ON.

The prefix option may be thought of as an instruction to the compiler to initiate checking. The scope of the prefix option depends on the order in which statements are written and compiled – that is, on the static structure of the program. ON statements, on the other hand, may be thought of as inserting branch instructions in a trap location. The actual branch taken depends on the last insertion, and therefore depends on the sequence in which statements are executed – that is, on the dynamic flow of the program. Examples of ON statements are:

```
ON ENDFILE (INFILE) CALL LAST_CARD;
ON OVERFLOW BEGIN;
OCOUNT = OCOUNT + 1;
IF OCOUNT>100 THEN STOP;
ELSE PUT LIST ('OVERFLOW NUMBER', OCOUNT);
END;
```

Besides providing a convenient and efficient way of dealing with special conditions, an interrupt can be used as a powerful tool for program checking. Among the conditions that may be enabled by means of a prefix is:

```
CHECK (identifier list)
```

The list of names may include variable names (which must not be subscripted, but may include names of arrays and structures), statement labels, and procedure names. An interrupt will then occur when the value of a variable is altered, or before the labeled statement or procedure is executed. The system action when the interrupt occurs is to write the identifier name, and its value, on a debugging file. The system action may of course be overridden by an ON statement. For example, the prefix:

```
(CHECK (L1, LOOP, BEGIN, HI, LO)):
PROCESS: PROCEDURE (X,Y);
```

would cause the specified label names in the procedure PROCESS to be written out before the statements to which they refer were executed.

```
(CHECK (Y)): BEGIN Y = A (I,P*Q–R) *Z**A;
```

would cause the value of Y to be written as for data-directed output.

PAUSE and STOP

The PAUSE statement in FORTRAN provides a primitive way of communicating with the operator. It is replaced in PL/I by the DISPLAY statement, which causes a message to be displayed to the operator. An example of a DISPLAY statement is:

```
DISPLAY ('END OF PHASE'||N);
```

which, if N had the value 2, would cause the message END OF PHASE 2 to be displayed at the console.

The parentheses following the word DISPLAY may contain any expression that is evaluated and converted to a character string when the statement is executed.

If the message must be acknowledged by the operator, a statement such as the following would be used:

```
DISPLAY ('NO CONVERGENCE AFTER'
||N||'ITERATIONS;
TYPE "GO" TO CONTINUE, "STOP" TO
ABANDON') REPLY (ANS);
```

Here ANS is a character variable. The program will be suspended until a reply has been typed. The reply will be entered into the variable ANS, and so can be tested by the program.

The STOP statement, in PL/I as in FORTRAN, terminates the execution of the program. The statement CALL EXIT, which is used in some FORTRAN systems, is replaced by EXIT in PL/I, and is used to terminate a task.

There are no tasks for sense lights or sense switches in PL/I, nor do functions such as SLITET exist.

Subprograms

The subroutine facilities in FORTRAN II were very largely responsible for its acceptance as a language for production programming. PL/I uses much of the basic philosophy of the FORTRAN function and subroutine, but has greatly increased the facilities provided. In FORTRAN, four types of subprogram exist:

1. The built-in function
2. The arithmetic statement function
3. The function subprogram
4. The subroutine subprogram

PL/I provides similar but extended facilities. The number of built-in functions has been increased; a list is given in Appendix 1. There is a much larger list of mathematical subroutines, and there are also subroutines for handling the new data types such as arrays, character strings, and bit strings.

An important extension to the concept of the FORTRAN built-in function is the generic function. Each generic function name refers to a group of subroutines, the actual subroutine chosen depending on the characteristics of the argument. For example, the single name MAX replaces AMAX0, AMAX1, MAX0, MAX1, and DMAX1; the single name SIN replaces SIN, DSIN, and CSIN.

The Procedure Statement

The form of the PROCEDURE statement differs from that of its FORTRAN equivalents. The PL/I format is chosen to be consistent with the rest of the language. The statement:

```
FUNCTION SOMEF(X,Y)
```

becomes:

```
SOMEF: PROCEDURE (X,Y);
```

The name of the procedure appears as a label.

```
SUBROUTINE RANDOM
```

becomes:

```
RANDOM: PROCEDURE;
```

The dummy arguments (which in PL/I are known as parameters) follow the word PROCEDURE. The list of parameters is enclosed in parentheses; items in the list are separated by commas. The last statement of a procedure is END; control is returned to the calling procedure by a RETURN or an END statement. The only difference between a subroutine and a function subprogram in PL/I is that the word RETURN in a function subprogram is followed by an expression enclosed in parentheses. The value of the expression is the value returned by the function.

An example of a procedure, used as a function is:

```
AGMEAN: PROCEDURE (X,Y);

AGAIN: T = (X + Y)/2;

Y = SQRT(X*Y);

X = T;

IF (X–Y) > .001 THEN GO TO AGAIN;
                ELSE RETURN (X);

END;
```

This function would be used in a statement such as:

```
A = AGMEAN(B,C);
```

The parameters used in a PROCEDURE statement may refer to variables (including label variables), arrays, files, or the names of other procedures. Parameters must not be subscripted, although the arguments in a CALL statement, or a function reference, may be. The mechanism of relating arguments to parameters in PL/I may be thought of as substituting the addresses of arguments for the addresses of parameters. These addresses will be evaluated at the time the procedure is called. Thus, in the example above, the arguments B and C will be replaced by their arithmetic geometric mean. Another consequence of this mechanism is that if an argument is subscripted, the subscript is evaluated when the CALL is initiated; any change in the value of the subscripts within the procedure will not alter the address. If an argument is an expression, the expression will be evaluated and the parameter will be replaced by the address of a dummy containing the value of the result.

Internal Procedures

In FORTRAN, the most elementary form of programmer defined subprogram is the arithmetic statement function. It has the advantages of not requiring a separate compilation, and of being able to refer to variables used in the calling program without using the devices of argument lists or COMMON. However, since the arithmetic statement function is limited to a single statement, it is seldom used.

PL/I allows the full flexibility of the subprogram-defining facility to be used either for separately compiled procedures or for procedures contained in, and compiled with, other procedures. The rules for nesting procedures are similar to those for DO groups: the nesting must not overlap. When a procedure is nested within another procedure, it can make use of

variables declared in the outer procedure. This eliminates the need for the equivalent of DIMENSION and COMMON statements at the beginning of every procedure.

Although procedures may be nested in this way, control must always pass to the procedure by a CALL statement or a function reference and will normally leave by a RETURN statement. GO TO statements may transfer control out of a procedure; the effect of the GO TO is as if a RETURN were executed and followed by a GO TO. A GO TO from a procedure called as a function will not return a value, and the evaluation of the expression containing the function reference will be terminated.

This provides a convenient means of specifying an error return within a subroutine. The GO TO may transfer to a label variable passed as a parameter, so that the error return can be specified in the CALL statement.

Separately Compiled Procedures

PL/I retains the facility, which has proved so valuable in FORTRAN, for communicating between separately compiled procedures.

Separately compiled procedures in PL/I are known as external procedures. Each external procedure has a PROCEDURE statement as its first statement and an END statement as its last statement. Other procedures may be nested within it.

External procedures may communicate through an argument list, in the same way as internal procedures. They may also communicate through names declared to have the EXTERNAL attribute, which corresponds in function to the COMMON statement in FORTRAN. Names declared external in two or more procedures are assigned the same storage and can be used by all the procedures in which they are declared. Procedure names and labels may be passed as parameters. This gives the same facility for specifying an error return as is available with internal procedures.

Multiple Entries to Procedures

Very often a programmer may want to write a subprogram with more than one entry point. Examples of this are routines that may have a considerable amount of code in common, such as sine and cosine, or subprograms that require an initializing entry and a normal entry. To allow for this PL/I provides the ENTRY statement, which specifies an alternate entry point to a procedure. This entry point may have a different list of parameters from that of the main entry. There may be more than one ENTRY statement in a procedure.

Consider, for example, a random number generator. It might require one entry point to set a starting number and multiplier, and another entry point for obtaining a random number — for example:

```
SETRND:  PROCEDURE (I, J);
         DECLARE (MIER, MICAND) STATIC;
         MIER = I;    MICAND = J;
         RETURN;
GETRND:  ENTRY;
         MICAND = MIER * MICAND;
         RETURN (MICAND);
         END;
```

This subprogram has an initializing entry and a normal entry; it would be invoked by statements such as:

```
CALL SETRND (M, N);
        :
X = GETRND;
```

In one case it is used as a function; in the other as a subroutine.

Input/Output

The input statements in FORTRAN were a compromise between ease of learning and flexibility of input/output format control. This compromise satisfied a very large proportion of potential users, but a number of open-shop users complained of the complication of the FORMAT statement, and an equally large number of more experienced users found that FORMAT did not provide the facilities they needed.

PL/I deals with this problem by providing several types of I/O facilities. These vary in level of complexity, and in the degree of control that the programmer has over the external format of the data records. The capabilities are grouped into two main categories, representing two basically different approaches to I/O.

In *Stream-Oriented* I/O, the data is considered as a continuous sequence of characters, which are separated into data fields. Logical and physical record boundaries are ignored. The external representation of the data need not be related to the internal form, and conversions are performed as needed, when each data field is transferred. At the simpler levels, the format need not be specified, and for input, the data names themselves can be obtained from the data stream. At the highest level, external format can be specified to an even greater degree than in FORTRAN.

In *Record-Oriented* I/O, a logical record is transmitted as an entity, there is no scanning of the record

for data fields. The external form is identical with the internal; no conversions are performed. Skipping of records or positioning at a specific record on a direct access device can be accomplished.

Stream I/O

The PL/I statements to transmit data fields in Stream I/O are GET and PUT. There are three modes of transmission which can be specified:

1. Data-directed – Self-identifying data, no format control similar to NAMELIST in FORTRAN
2. List-Directed – Data list in I/O statement, no format control
3. Edit-directed – Data and format list in I/O statement; similar to READ/WRITE with format in FORTRAN.

Data-Directed Input/Output

Data used with data-directed input/output statements appears in a form very similar to a series of assignment statements. For example, an input card might contain:

```
A = 17.7, B = 13.2, C = 9.1;
```

The statement to read this could be:

```
GET DATA;
```

This statement would assume that the data is on the system input file, since no file is specified. The names appearing on the input medium must have been used in the procedure containing the GET, if this form is used. This is needed in order to make the name known to the compiler, so that a data type and storage location can be noted for use at object time.

The same effect is achieved by writing the statement:

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

This differs from a FORTRAN list in that there need be no correspondence between the order of the input data and the order of the list.
Thus, the statement could also have been written:

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

Neither is it necessary that all variables on the list appear on the input medium. A GET statement will process the assignment indications which are separated by commas until one, terminated by semicolon, appears. That ends the GET statement, and any variable names in the list which were not encountered, will leave the data value unchanged.

On output, the PUT statement must contain a list, and the output produced will be a series of assignment statements appearing in the order specified by the list. The statements will be in a form that could be read using a data-directed GET statement with the same list. For example, a program using the procedure AGMEAN, which was used as an example in "The Procedure Statement", could be:

```
ON ENDFILE (SYSIN) STOP;
IN: GET DATA (A,B);
X = AGMEAN (A,B);
PUT DATA (A,B,X);
GO TO IN;
```

The input data for this could be:

```
A = 1, B = 2; B = 3; B = 4;
```

Each GET statement would read up to a semicolon, so this program would find the arithmetic mean for the three cases:

```
A = 1,B = 2; A = 1,B = 3; A = 1,B = 4;
```

and then end the program when an end-of-file is reached. The results would be written on the standard output file as a series of assignment statements.
The output format of numbers in data-directed output is similar to the G format of some systems (for example, 1620 FORTRAN without FORMAT). The number of digits output is equal to the precision of the internal data. The decimal point is adjusted within the field to give the correct numeric value without an exponent, if possible. Otherwise, an E with two digits and a sign is attached to give a valid floating-point constant.

On input, any valid constant is allowed to the right of the equal sign. If the variables on the left-hand side are arrays, the names may have constant subscripts. For example, in a program with the declaration:

```
DECLARE A (12,12),B(6,6),C(6,6);
```

the statement:

```
GET DATA;
```

would be very convenient way of reading either a few alterations to matrices already in core, or even whole matrices if most elements are zero.

Data-directed output is a very simple way of producing a few results with annotation. It is also a very straightforward way of producing legible debugging output for testing flow or arithmetic.

List-Directed Input/Output

With list-directed input and output, the programmer specifies the sequence and the names of the data items to be read from or written onto the external file. He does not, however, have to specify the format, nor

is the input format rigidly controlled. Data appears on the file as a sequence of constants, separated by one or more blanks or a comma. As with constants in the source program, imbedded blanks are not allowed. Contrary to data-directed I/O, the number of data fields processed is exactly the same as the number of scalar items in the list. If array or structure names appear in the list, they are considered as merely a shorthand for all of their components, and thus the number of data fields must equal the total of all the parts of the aggregates.

List-directed input provides a convenient method of specifying free-form input for long lists of data such as matrices. It is also a convenient input specification for input media such as paper tape or remote terminals, which do not use card columns for positioning input data. A list-directed input statement has the form:

```
GET LIST (data list);
```

The word LIST specifies that the I/O is list-directed; it is immediately followed by the data list which is enclosed in parenthesis. Examples of list-directed statements are:

```
GET LIST (I, J, A(I,J));
PUT LIST ('VALUE OF X IS',X);
GET LIST (A_MATRIX);
```

NOTES:

1. As in FORTRAN, data read in may be used in subscripts later in the list.
2. Unlike FORTRAN, character data appears in the list as a character literal and not in a FORMAT statement.
3. Array names may appear in lists, as in FORTRAN.

File Names

In FORTRAN, the usual input/output medium is magnetic tape. The programmer specifies logical unit numbers in his READ and WRITE statements and gives instructions to the operator for mounting and dismounting tapes.

In PL/I the programmer addresses files, not units; the location of these files is left to the operating system or may be specified by means of control cards. The name of the file is specified in an input/output statement by the option:

```
FILE (file name)
```

If no name is specified, the standard input (SYSIN) and output (SYSPRINT) files are assumed, as shown in all the preceding examples. Typical statements referring to other files are:

```
GET LIST (A, B, K) FILE (IN_DATA);
PUT FILE (PRINT) DATA (ITERATION, DELTAX, X);
```

The options may appear in any order.

Data Lists

The data lists used in edit-directed and list-directed input control statements have a function very similar to that of the list in a FORTRAN input/output statement.

For input, a list consists of a list of variable names, separated by commas. As in FORTRAN, an array name may be used in a list. In PL/I, arrays are transmitted in row-major order — that is, with the rightmost subscript varying most rapidly. This does not correspond with most FORTRAN implementations. Each complex variable corresponds to two numbers on the external medium.

As in FORTRAN, lists may contain indexing loops. PL/I allows the full power of its DO specification in input/output lists, and looping within a list may be nested. An example of an indexed list is:

```
GET LIST (ID, N,( ( A(I,J) DO J = 0 TO I-1,
I+1 TO N) DO I = 0 TO N));
```

This statement would read in the two scalar items, ID and N, and then all except the diagonal elements of the matrix, A.

For list-directed and data-directed output, the list may contain expressions as well as single variable names. The expression is evaluated when the PUT statement is executed, and the result is the data that is output. A program could conceivably consist of only one executable statement, for example:

```
PUT LIST ('SQUARE ROOTS OF FIRST 100
INTEGERS', (X,SQRT (X) DO X = 1 TO 100));
```

Edit-Directed Input/Output

The edit-directed input/output statements in PL/I come closest to FORTRAN input/output. The main difference between a PL/I and FORTRAN format specification is that in PL/I the format type refers only to the appearance of the data on the external medium, whereas in FORTRAN a format type specifies conversion between one internal form and one external form. For example, F conversion in FORTRAN specifies conversion between internal floating point and external fixed point, with a printed decimal point. In PL/I, F format may be used with any internal data; externally the data appears in the same form as the FORTRAN F format output in FORTRAN.

A second but less fundamental difference is that in PL/I the format specifications may be either within the GET or PUT statement, or in a separate FORMAT statement. Examples of edit-directed input/output statements are:

```
GET EDIT (I, J, A(I,J))(2 F(2), F(10,4));
PUT EDIT (A, (X (I) DO I = 1 TO 5)) (R (FMT1));
FMT1: FORMAT (SKIP (2), F(11, 3), X(10),
5 E (14, 7));
```

The format specification is enclosed in parentheses and immediately follows the data specification. It may be either a list of format elements, similar to FORTRAN format elements, or it may be the label of a FORMAT statement elsewhere in the program, enclosed in parentheses and preceded by the letter R (for remote). Each element consists of a letter specifying the type of data on the external medium, followed by specifications for size of field, decimal places, etc. Unlike FORTRAN, these need not be integer constants; instead, they may be any expression. Therefore, to avoid ambiguity, these specifications are separated by commas and enclosed in parentheses. Thus, instead of E14.8, in PL/I one would write E(14,8).

Types of Format Items

E and F formats in PL/I correspond to FORTRAN E and F formats. The rules for truncation of fields on output, the use of signs, and the punching or omission of decimal points on input, are the same in PL/I as in FORTRAN. One difference is that in PL/I the F format also fulfills the function of the I format in FORTRAN. If the d specification is omitted, a point is not printed, and the output item appears as an integer.

The following examples illustrate the use of F and E formats:

X = −123.4567

F(10)	−123
F(10,5)	−123.45670
F(10,3,2)	−12345.670
E(13,7)	−.1234567E+03
E(13,5,7)	−12.34567E+01

An F format item without a d specification, corresponds to an integer externally in spite of the value of X. A third specification, a scale factor, can be included to cause the external value to be ten raised to that power times the internal value. In E format, two specifications can be given in addition to the field width, w. One, d, gives the number of digits to the right of the decimal point, and the other, s, if given, is the total number of significant digits. If this third specification is omitted, it is assumed equal to d.

As in FORTRAN, the A format specifies alphabetic data on the external medium, and X format specifies characters to be ignored on input, and blanks on output. PL/I also includes format items for complex data and bit strings. In addition, if the external form does not match any of the possible format types, a PICTURE format specification can be given which indicates the contents of each character position.

Since literal data to be printed appears in the data list, rather than the format list, there is no need for the H specification. An A format item, however, must appear in the format list at the position desired for the character string. The width need not be specified; it will be deduced from the literal itself. Carriage control characters are not written as such by the programmer. Format items, PAGE, SKIP, LINE, and COLUMN are placed instead in the format list, and the necessary characters are created at object time.

```
PUT PAGE LINE (3) EDIT
  ('SUMMARY') (COLUMN (13), A);
PUT SKIP (2) EDIT (X,Y) (X(3), F (12,5), X (7),
  E (30,15));
```

This would cause an eject to a new page and SUMMARY to be printed on the third line, starting in printing position 13. Then one blank line would be skipped and X would be printed in F format beginning in the 4th column, seven spaces after X would be left blank, and Y would be printed in E format, all on the fifth line of the page.

Format List

In PL/I, as in FORTRAN, each item in the data list is matched against an item in the format list. A format item may be preceded by an iteration factor; thus:

```
3 A (20)
```

is equivalent to:

```
A(20), A(20), A(20)
```

A difference between PL/I and FORTRAN is that, in PL/I, if a format list is completed before all the items in the data list have been transmitted, the format list starts again from the beginning, and not from the last left-hand parenthesis, as in FORTRAN.

Also, if the data list is completed before the format list is exhausted, the unused portions of the format list are ignored. This is true even if the next items are for control only and are not associated with data list items.

Variable Format

FORTRAN has the facility to vary a format specification by reading in a FORMAT card at object time. In

PL/I, this facility can be obtained differently. One technique would be to write the I/O statement with a format specification for a remote format list. The label used can be a variable and can be set to any one of the possible formats. For example:

```
        GET LIST (TYPE);
        IF TYPE = 1 THEN DUMMY = FMT1;
                    ELSE DUMMY = FMT2;
        GET EDIT (A, B, C) (R(DUMMY));
FMT1:   FORMAT (3 F(10,3));
FMT2:   FORMAT (3 E(14,8));
```

Here the variable DUMMY must have been declared with the attribute LABEL. Then the single I/O statement can be made to operate with either of the FORMAT statements.

A more general solution is to make use of the fact that PL/I allows expressions for the w and d fields of format items, as well as the repetition factor. For example:

```
DECLARE X(N);
PUT EDIT (X) ((N) F (120/N,5));
```

This could be used to space the N elements of array X across the page. The width of the field would be computed when the I/O statement is executed, and would be based on the value of N, which could have been read in or computed.

It is also permissible for the expression indicating repetition to evaluate to zero or a negative number, in which case the format item is skipped. Thus, a single format list can be used with various results by selecting those items which are used. For example:

```
PUT EDIT (A, B) ((KEY) F(10,3), (1–KEY) E(14,8));
```

Here, if KEY is equal to 1, the second expression is zero and the format item is skipped. Thus, both variables would use the F format. If KEY equals zero, E format is used for both.

Internal I/O

In PL/I, it is possible to have an I/O statement refer to an internal character string, rather than an external file. Thus, data already read in, can be moved and converted by GET and PUT statements with the STRING option.

Frequently, it is necessary to read in a record whose format is not known by the program, but is indicated on the record itself. In FORTRAN, this could be accomplished by reading in the record, examining the indicator, writing the record out on a scratch file and then reading it back in, with the proper format. One of the uses of the STRING option in PL/I is to solve this problem. For example:

```
GET EDIT (TEMP) (A(80));
GET STRING (TEMP) EDIT (CODE) (F(1));
IF CODE = 1 THEN GO TO OTHER_TYPE; ELSE
GET STRING (TEMP) EDIT (X, Y, Z) (X(1),
  3 F(10,4));
```

The first GET reads in eighty characters from SYSIN and places them in the character string TEMP without any conversion. TEMP must have been declared to be a string of at least eighty characters in length. The other GET statements use this variable as the source of data, rather than reading in anything else. The data remains in the character string and can be re-used by any number of GET statements, or processed in any other way desired.

Print Files

In stream-oriented I/O an output file may be declared with the attribute PRINT. This enables carriage control information to be included as options on the PUT statement. The options may be used with any mode of output, data-directed, list-directed, or edit-directed, and precede the printing of the data, regardless of the order in the PUT statement. For example:

```
PUT PAGE LIST ('PAGE NO', N);
PUT FILE (RESULTS) DATA (TOTAL) SKIP (3);
PUT LINE (40) EDIT (I, DELTAX) (F(10), F(20,5));
```

The first example would eject to a new page on SYSPRINT, and then print the data on the top line. Two lines would be skipped on file RESULTS and then the representation for TOTAL in the second example would be printed on the third line. The last would cause the data list to be printed on line 40 of the current page.

Record I/O

Each READ and WRITE operation transmits a single logical record. The record is moved between the external medium and the variable specified without any conversions. The variable specified must be a structure and will normally contain several data items or arrays. (See "Structures" in Part 3 for a description of the type of data organization.)

One of the uses for Record I/O would be for temporary storage of intermediate results. This is analogous to unformatted READ/WRITE statements in FORTRAN. For example:

```
WRITE FILE (SCRATCH) FROM (TEMP1);
READ FILE (UT1) INTO (RAW DATA);
READ FILE (DATAIN) IGNORE (7);
```

FROM and INTO are key words which are followed by the single name of a structure. The IGNORE option on READ specifies skipping the number of records given by the expression following it.

Structures

Arrays are a convenient notation for many mathematical problems that are concerned primarily with collections of numbers having the same attributes.

Some programs are concerned with collections of data of different types. For example, a record of a customer file might well contain various items of numeric information of different length and alphabetic information such as a name and address. It may often be desirable to treat the record, or parts of it that contain more than one item, as a whole.

PL/I uses the word "structure" to describe a collection of variables of different types, organized in a hierarchy. Consider the collection of variables:

```
EMP_NO,NAME,SALARY,INSUR,LOAN,SAVE.
```

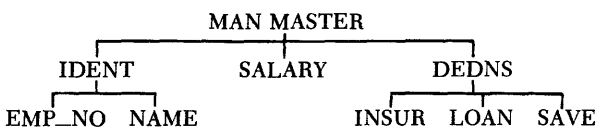
A programmer might wish to move the whole collection, or only a part of it. To do this he must be able to name the group to which he intends to refer. The following declaration accomplishes this:

```
DECLARE 1 MAN_MASTER, 2 IDENT,
3 EMP_NO, 3 NAME, 2 SALARY, 2 DEDNS,
3 INSUR, 3 LOAN, 3 SAVE;
```

The number preceding each name indicates the level of the name. The name MAN_MASTER has level 1, the highest level. This name refers to a structure that includes all the other names in this declaration.

```
IDENT includes EMP_NO and NAME
DEDNS includes INSUR, LOAN, and SAVE.
```

The collection of items could be represented pictorially as:



A structure could be used to group together data which does not have a hierarchial relationship, but which is convenient to refer to by a single name. Sending a collection of arguments to a procedure could be simplified by using a structure as follows:

```
DECLARE  1   ARGLIST,
             2 X COMPLEX,
             2 I   ,
             2 Y (10,10),
             2 Z BIT  (6),
             2 (A, B, C) FLOAT (16)
CALL SUB_1 (ARGLIST);
```

The CALL statement with the single argument would then transmit the entire collection of data to SUB_1. The parameter in SUB_1 would need to be declared as a structure with the same attributes as those being sent, but not necessarily the same names.

To simplify the transmission of data even further, the entire structure could be declared EXTERNAL in both procedures. This would give the analogy of named COMMON in FORTRAN IV in that the external names used, and the order and types of data, must match in all procedures but the names of individual elements could be different. If variables are declared EXTERNAL individually, then the name of each item is known to each procedure making the declaration. In this case, the names must match exactly, but the order of declaration may be different.

Name Qualification

In order to refer to an individual item in the structure, names can be used that do not contain lower levels. In the above example, these are EMP_NO, NAME, SALARY, INSUR, LOAN, and SAVE. However, these names need not be unique. It is permissible to use some or all of the same lower-level names in another structure declared in the same part of the program. For example, another structure in the same program as the previous example might be: DECLARE 1 PAYSLIP, 2 IDENT, 3 EMP_NO, 3 NAME, 2 GROSS_PAY, 2 DEDNS, 3 INSUR, 3 LOAN, 3 SAVE, 3 TAX, 2 NET_PAY;.

To distinguish between the two different variables both called INSUR, it is necessary to specify some additional information. This is done by qualifying them with other, higher-level names to make the identification unique. One INSUR is contained in DEDNS, which is contained in PAYSLIP; the other INSUR is contained in DEDNS, which is in turn contained in MAN_MASTER. To differentiate between the two, the names PAYSLIP.DEDNS.INSUR and MAN_MASTER.DEDNS.INSUR can be used. More simply, PAYSLIP.INSUR and MAN_MASTER.INSUR could be used, since DEDNS does not help in distinguishing between the two. The process is known as qualification. The rules for qualification are that a name used in a structure may be qualified by prefixing it with the names of structures in which it is contained. The names are separated by a period and must be in order of level number, the most inclusive level

appearing first. The names used must be sufficient to positively identify the variable.

Assignment BY NAME

The ability to use, in a structure, names that are not unique does help the programmer in maintaining the mnemonic significance of the words he is using. The most important use of this facility is to indicate that the variables have something in common, and to make use of that fact in his program. The following example shows how PL/I allows this to be done:

```
PAYSLIP = MAN_MASTER, BY NAME;
```

Each of the names in each structure is compared with all the names in the other structure. The comparison includes higher-level qualifications up to, but not including, the names that actually appear in the assignment statement. The assignment is then executed wherever names match. With the examples used previously, the single assignment statement above would be equivalent to:

```
PAYSLIP.EMP_NO =MAN_MASTER.EMP_NO;
PAYSLIP.NAME=MAN_MASTER.NAME;
PAYSLIP.INSUR=MAN_MASTER.INSUR;
PAYSLIP.LOAN=MAN_MASTER.LOAN;
PAYSLIP.SAVE=MAN_MASTER.SAVE;
```

or

```
PAYSLIP.IDENT=MAN_MASTER.IDENT;
PAYSLIP.DEDNS=MAN_MASTER.DEDNS,BY NAME;
```

In the case of the structure IDENT the BY NAME option is not required, since the two structures have the same composition. If two structures have the same organization (that is, if they are divided into levels in the same way, and if each level contains the same number of items), the names of the items do not need to match.

Structure Expressions

Structures may also be used in expressions. If the BY NAME option is used, all the names are compared and the expression is evaluated for all the cases where the names match. If structures appear in an expression without the BY NAME option, the structures must be identically constructed. For example, given the declarations above, and:

```
DECLARE 1 HASH, 2A, 3D, 3E, 2B, 2C,
3F, 3G, 3H;
```

then:

```
HASH = HASH + MAN_MASTER;
```

would cause each element of MAN_MASTER to be added into the corresponding element of HASH.

There is no restriction on the types of variables in the two structures; corresponding elements of the two structures may be different types. If conversion is required, it is performed according to the rules for arithmetic expressions.

Structures, like arrays, provide a convenient way of referring to a group of items, using a single name. Unlike arrays, every item must have its own name, and identification to the lowest level of the structure is by name. Also unlike arrays, the types of the elements differ.

Arrays of Structures

Some collections of data may be most easily organized as a combination of arrays and structures. The simplest case would be where a table has an argument of a different type from the function. For example, a table of parts might have 100 entries, each entry consisting of PART_NO, DESC, QTY, COST. This could be declared as:

```
DECLARE 1 ITEM (100), 2 PART_NO FIXED
(9,0), 2 DESC CHARACTER VARYING (20),
2 QTY FIXED (4,0), 2 COST FIXED (5,2);
```

The dimension following the name ITEM indicates that it is a vector and that individual items may be selected by subscripting. Thus, ITEM (11) would be the eleventh entry in the table, and the eleventh part number would be:

```
ITEM.PART_NO (11), or PART_NO (11)
```

More than one level may be subscripted, for example:

```
DECLARE 1 PART (100), 2 PRICE,
2 STOCK (10);
```

To refer to a particular element of STOCK, for a particular PART, one could write PART (I) .STOCK (J) or PART.STOCK (I,J) or, if PART is not needed to positively identify STOCK, it could be written STOCK (I,J).

Allocation of Storage

One of a programmer's problems when writing in a high-level language, or even assembly language, is that every time a variable is declared, storage is assigned for it, normally for the entire duration of the program. Usually, if he is writing a program that is divided into subroutines, he wishes, when entering a subroutine, to be able to use names and storage that will not be confused with the names and storage used in the calling program. When he leaves the subroutine, he has

finished with the working storage used in the subroutine, and he does not want to be prevented from using, in the main program, names used in the subroutine.

FORTRAN protects the user from accidental confusion of names between subroutines, while at the same time allowing communication between subprograms by means of the COMMON statement. However, the storage associated with names remains static. If a subroutine uses working storage for an array, that storage remains assigned, whether or not the subroutine is in use.

The EQUIVALENCE statement in FORTRAN helps, by allowing two names to refer to the same location in storage, but once again this a static arrangement and cannot be altered after compilation. Another problem arising from the static allocation of storage is that the amount of storage allocated to an array cannot be altered at object time. If an array is smaller than the DIMENSION statement specifies, the surplus space is wasted. Another difficulty is that the DIMENSION statement specifies the limit on each dimension. If an array may be 100 x 50 or 50 x 100, the dimensions allotted to it must be 100 x 100, and half the space is wasted.

In a multiprogramming environment, where surplus storage might profitably be used by another program, it becomes even more important that a program should not retain storage it is not using. The operating system will therefore provide facilities for assigning and releasing storage. The storage assignment features of PL/I are particularly relevant to this type of environment and operating system.

The Scope of Names

IN FORTRAN, a variable name used in a subprogram cannot be confused with other uses of the same name in other subprograms. If the name is required to refer to the same variable in various subprograms, it is placed in COMMON.

In PL/I, a name is known in the procedure in which it is declared, and in all procedures nested within that procedure. A name can, however, be declared in an inner procedure, and in this case the name refers to a new variable and is not confused with the previous use of the name.

This means of communication cannot work with separately compiled procedures. PL/I, therefore, provides an attribute EXTERNAL for variable names. A name that has been declared EXTERNAL is known to all other procedures in which it has also been declared EXTERNAL. In this way, EXTERNAL is analogous to COMMON; this is illustrated in the following example:

```
A: PROCEDURE; DECLARE (X,Y,Z) COMPLEX;
   .
   .
   .
B: PROCEDURE; DECLARE (P,Q,R,X) FIXED;
   .
   .
   .
   END B;
C: PROCEDURE; DECLARE V EXTERNAL FLOAT
   BINARY;
   END C;
   END A;
D: PROCEDURE; DECLARE V EXTERNAL FLOAT
   BINARY;
   END D;
```

V is common to procedures C and D.
Y and Z are known in procedures A, B, and C, and are complex.
X is complex in procedures A and C, but refers to a different fixed-point variable in procedure B.
P, Q, and R are known only in procedure B.

STATIC and AUTOMATIC Storage

In PL/I there are four ways in which data storage may be assigned: STATIC, which corresponds to FORTRAN usage; AUTOMATIC, which is similar to ALGOL usage; CONTROLLED and BASED, which are assigned by executable statements called ALLOCATE and FREE.

STATIC storage is assigned when a job is loaded, and remains assigned until the end of a job. This is the kind of storage to which the FORTRAN programmer is accustomed.

AUTOMATIC storage is assigned on entry to the block in which it is declared, and is released on exit from that block. A block is either a procedure or a group of statements headed with BEGIN and terminated with END. Either form of block may include DECLARE statements. If it does, the AUTOMATIC storage specified in the DECLARE statement is assigned only when control passes to the block. A procedure may gain control as a result of being called either as a function or as a subroutine, and may be called at either its main entry point or a secondary entry point. In each case, the same assignment is performed. On exit from the procedure, by a RETURN or END statement, the storage allocated on entry is released.

Since the assignment and release of storage, particularly in a multiprogramming environment, will involve calls to a Supervisor, which might be time-consuming, this class of storage will not normally be used for single variables. However, in matrix problems there are significant advantages to be gained. Consider a program in which two matrices are to be multiplied. Once the product has been formed, the original matrices are no longer required; only the product

need be retained. The following BEGIN block would achieve this:

```
STRESS: PROCEDURE; DECLARE C(12,12) BINARY;
PROD:   BEGIN;
        DECLARE (A(12,12), B(12,12)) BINARY;
        GET LIST (A,B);
        C = 0;
        DO I=1 TO 12; DO J=1 TO 12;
        C(I,J) = SUM (A (I,*)*B(*,J));
        END PROD;
        .
        .
        .
END STRESS;
```

If names in a DECLARE statement are not declared EXTERNAL, they are assumed to be INTERNAL; similarly, if they are not declared STATIC, CONTROLLED, or BASED, they are assumed to be AUTOMATIC. Therefore, A and B are assigned storage on entry to the block:

```
PROD: BEGIN;
```

When the block is completed, A and B are discarded. The array C remains and may be used in a succeeding statement.

CONTROLLED and BASED Storage

STATIC and AUTOMATIC storage deal with most cases where a programmer needs to control storage only as control enters or leaves a block. There may, however, be cases where conditions arising during execution may influence the assignment of storage. In a particular case, for example, the need for additional storage may become apparent only when the input data is supplied.

CONTROLLED and BASED storage are provided to handle such situations. Storage that has been specified as CONTROLLED in a DECLARE statement can be assigned and released by the ALLOCATE and FREE statements.

In the previous example, if the DECLARE statement had been:

```
DECLARE (A(12,12), B(12,12))
CONTROLLED, C(12,12) STATIC;
```

the block might have contained statements such as:

```
ALLOCATE A,B;
        .
        .
        .
IF (SPECIAL) THEN FREE A,B;
END;
```

The storage for arrays A and B would be allocated only when the statement to do so is executed. When the FREE statement is executed the storage is released, and any data values for A and B are lost. If a second ALLOCATE were executed before the FREE statement, the previous allocation would be "pushed down" and a new one would be created. Now, a FREE statement would release the most recent allocation, and "pop-up" the one previous to it with its data values preserved.

The BASED storage type also allows the ALLOCATE and FREE statements to control storage assignment. Unlike CONTROLLED, however, BASED does not "stack" multiple allocations for the same variable but permits concurrent reference to each allocated area. Special variables called *POINTER variables* are used to distinguish several allocations for the same BASED variable.

Variable Dimensions

Since the storage for AUTOMATIC, CONTROLLED, and BASED data is allocated at object time, it is possible to specify the dimensions of AUTOMATIC, CONTROLLED, or BASED arrays at object time. The dimension attributes for these three classes of arrays may, therefore, be variables, or even expressions. The values of the variables must be known on entry to the block in which the array is declared, or, in the case of CONTROLLED or BASED data, at the time that the ALLOCATE statement is executed. Examples of this method of declaration are:

```
GET LIST (L,M,N);
BEGIN; DECLARE A(L,M),B(M,N),C(L,N);
```

Where a called procedure needs to declare an array of the same dimensions as a calling procedure, the dimension specifications may be replaced by asterisks. For example:

```
        DECLARE X (100,100);
        CALL INVERT (X);
INVERT: PROCEDURE (X);
        DECLARE X (*,*);
```

The dimension bounds of X are transmitted to procedure INVERT along with the data. X can then be used in array assignment statements without the need for passing or knowing the dimensions. If they are needed for other purposes, a built-in function, DIM, will supply them as needed. Note, however, that the asterisk notation is not permitted for BASED variables.

Asynchronous Procedures

IN FORTRAN, statements are executed one at a time in the order in which they are written, except where this is modified by control statements such as GO TO and DO. A READ statement can, however, take a long time to execute, and during most of this time only the channels will be active, the CPU being idle. With Teleprocessing lines and typewriter displays, the de-

lays can be even more severe than with a conventional magnetic tape system. Buffering systems can help, but limitations on buffer size make it impossible to achieve the same efficiency as a good programmer in making use of the channels and the CPU simultaneously.

In PL/I, a procedure may be called asynchronously. This means that control may pass to the next statement before the called procedure has been executed. The control system treats the called procedure as an independent task and assigns it a priority. Whenever an interrupt occurs, the task with the highest priority that is able to proceed is given control. From the point of view of the programmer, the called procedure may be considered to be executed in parallel with the calling procedure.

The advantage to be gained from this will depend on the amount of computing that can be done by one program while the other is executing channel commands. Where disks are being used for random processing, or Teleprocessing lines are being used, the savings can be considerable.

A newer development is the computing installation with two or more central processing units in direct communication with each other. In this situation, the ability to execute more than one task at a time is fundamental. The provisions for the simultaneous operation of tasks need to allow for more than two tasks, and for more than two levels of task. One task may call another, which may in turn call yet another. In this way, a "tree of tasks" may be built.

The relationship of tasks that call each other is a purely dynamic one. The exact form of the tree of tasks is not normally known at compile time. Tasks that may affect each other's data need a means of testing whether other accesses to that data are complete.

Procedures, whether or not they are called asynchronously, communicate in exactly the same way. External data is known between procedures; automatic data is assigned on entry to a procedure, in the same way as in a single task. The only exception is that allocated controlled data can be freed only in the task that allocated it.

The TASK, EVENT, and PRIORITY Options

A procedure may be called asynchronously by attaching one or more of the three options TASK, PRIORITY, and EVENT to a CALL statement.

The TASK option names the task so that it can later be referred to in PRIORITY pseudo variables or functions. The EVENT option names a variable that can be waited on or tested to determine whether the task is complete. The PRIORITY option specifies the priority of the attached task relative to the task creating it.

A subroutine to assemble and transmit a message could be called by a statement such as:

```
CALL TRANSMIT (A, DATE_TIME, ADDRESS,
TEXT) EVENT (MESSAGE);
```

Control can go to the statement following as soon as the task has been established. The task would be terminated when the procedure TRANSMIT reached a RETURN or END statement that would normally return control to the calling procedure. If an exit statement is executed in a task, the task is abandoned, but the calling procedure is unaffected.

The WAIT Statement

The programmer needs some means of testing whether a task has been completed. (The completion of a task does not interrupt the program that initiated it.) One way of doing so is by the WAIT statement, which suspends the flow of control through the program containing the WAIT statement until the task specified by the WAIT statement is complete. Suppose that the programmer wishes to wait until the message in the CALL statement of the previous example has been transmitted, and the procedure TRANSMIT has been terminated. This could be done by the statement:

```
WAIT (MESSAGE);
```

The name MESSAGE identifies the event to which the WAIT statement refers, namely the completion of the task that was created with MESSAGE as its event variable. The WAIT statement can refer to a number of tasks by specifying a list of event variables. For example:

```
WAIT (MESSAGE1, MESSAGE2, MESSAGE3,
REPLY);
```

If the programmer wishes to wait for only one of these to be completed, he can write:

```
WAIT (MESSAGE1, MESSAGE2, MESSAGE3,
REPLY) (1);
```

The integer following the parentheses specifies the number of events in the list that are to be completed before the calculation proceeds.

The COMPLETION Function

There will often be cases where the programmer will not want to go into a waiting state to test whether a task has been completed. This situation is provided for

by a built-in function COMPLETION. The argument of the function is the EVENT variable associated with the task to be tested. The value of the function is binary 1 if the task has been completed, and binary 0 if it has not. (1 and 0 correspond to TRUE and FALSE.) This function could be used in a statement such as:

```
IF COMPLETION (COMPUTE) THEN GO TO
PROCESS;
```

There may be cases where the programmer wishes to test whether a task has reached a particular point other than completion. An EVENT variable may be set by a statement of the form COMPLETION(P) = 1. This can then be tested in another task by a WAIT statement or by a COMPLETION function. Note that such EVENT variables should be, at the time, unassociated with the completion of a task. Thus they should be declared as EVENT variables.

PL/I also permits the DISPLAY statement and certain RECORD-oriented input/output statements to contain an EVENT option. Such statements, however, cannot be tested for completion by means of the COMPLETION function; the WAIT statement must be used instead.

List Processing

PL/I provides an additional data type known as a POINTER. This is a data item whose value is the location of another data item. This gives the ability to chain together strings of data. Structures might be used where all pertinent data is collected into one aggregate, and one of the fields in the structure is a pointer to the next structure in the chain. Pointers may also point to other pointers, and thus a multilevel organization can be created.

These functions, which are summarized below, are described in greater detail in the publication *IBM Operating System/360: PL/I Language Specifications* (C28-6571).

In the summary below, the following notation is used:

X,Y and Z	Denote obligatory arguments, which may be expressions.
O,P and Q	Denote optional arguments, which may be expressions.
I and J	Denote obligatory arguments, which must be integer constants.
M and N	Denote optional arguments, which must be integer constants. In their absence, a default value is assumed that will depend on either the arguments or the implementation.

Arithmetic Generic Functions

The base, scale, and mode of the arguments of the arithmetic generic functions are used to determine the characteristics of the result, except where these are specified by the function itself (for example, FIXED).

When an argument specifies a fixed-point scale factor, it may be omitted if the result is floating point.

The arithmetic generic functions are listed below:

FUNCTION (ARGUMENTS)	VALUE RETURNED
ABS(X)	Absolute value of X.
MAX(X,Y,O,...)	Value of maximum argument of any number of arguments.
MIN(X,Y,O,...)	Value of minimum argument of any number of arguments.
MOD(X,Y)	Remainder of integer division X/Y.
SIGN(X)	1 if X is positive, 0 if zero, −1 if negative.
FIXED(X,M,N)	X, converted to fixed point, precision (M,N).
FLOAT(X,M)	X, converted to floating point, precision (M).
FLOOR(X)	Largest integer not exceeding X.
CEIL(X)	Smallest integer not exceeded by X.
TRUNC(X)	X, truncated to an integer.
BINARY(X,M,N)	X, converted to binary, precision (M,N).
DECIMAL(X,M,N)	X, converted to decimal, precision (M,N).
PRECISION(X,I,M)	X, precision altered to (I,M).
ADD(X,Y,I,M)	X + Y, precision (I,M).
MULTIPLY(X,Y,I,M)	X*Y, precision (I,M).
DIVIDE(X,Y,I,M)	X/Y, precision (I,M).
COMPLEX(X,Y)	Complex result (X + iY).
REAL(X)	Real part of X.
IMAG(X)	Imaginary part of X.
CONJG(X)	Complex conjugate of X.

Floating-Point Arithmetic Functions

These functions have the conventional mathematic meanings. A terminal D indicates that the argument, or, in the case of ATAND, the result, is in degrees and not radians.

The arguments will be converted to floating point before evaluation; the result will be floating point. Some, but not all, of the functions will accept complex arguments (see Appendix 1 in C28-6571).

The functions are listed below:

EXP(X)	COS(X)
LOG(X)	TAND(X)
LOG10(X)	TAN(X)
LOG2(X)	TANH(X)
ATAND(X)	SINH(X)
ATAN(X)	COSH(X)
ATAN(Y,X)	ATANH(X)
ATAND(Y,X)	ERF(X)
SIN(X)	ERFC(X)
SIND(X)	SQRT(X)
COSD(X)	

String Functions

These functions return bit strings if the arguments are binary, character strings if the arguments are decimal.

FUNCTION (ARGUMENTS)	VALUE RETURNED
BIT(X,M)	X, converted to a bit string of length M.
CHAR(X,M)	X, converted to a character string of length M.
SUBSTR(X,Y,P)	A string, P characters (or bits) long, starting from the Yth character (or bit) of X.
INDEX(X,Y)	If Y is not a substring of X, the value 0 is returned. If Y is a substring of X, the value returned is the position of Y in X.
LENGTH(X)	The length of the string X.
HIGH(I)	A string, I characters long, of the highest data characters.
LOW(I)	A string, I characters long, of the lowest data characters.
REPEAT(X,I)	The string X repeated I times.
UNSPEC(X)	A bit string that is the internal representation of X.
BOOL(X,Y,Z)	Boolean function (see C28-6571 for details).

Array Functions

These may have array expressions as arguments. All the functions return a scalar result. In addition, any of the arithmetic or string generic functions may have an array as an argument. The result will be each element of the array operated on individually by the function.

FUNCTION (ARGUMENTS)	VALUE RETURNED
SUM(X)	The sum of all the elements of X.
PROD(X)	The product of all the elements of X.
ALL(X)	The AND sum of the elements of X after the elements have been converted to bit strings.
ANY(X)	The OR sum of the elements of X, after the elements have been converted to bit strings.
POLY(X,Y)	Polynomial (see C28-6571).
LBOUND(X,Y)	The lower bound of the Yth dimension of X.
HBOUND(X,Y)	The upper bound of the Yth dimension of X.
DIM(X,Y)	The extent of the Yth dimension of X.

Condition Functions

These functions may be used only in ON units. They return information on various aspects of the program at the moment when an interrupt occurs.

FUNCTION	VALUE RETURNED
ONCOUNT	The number of unprocessed interrupts produced by the abnormal termination of an input/output event.
ONFILE	The name of the file on last I/O operation.
ONLOC	The entry name of the current procedure.
ONSOURCE	The field being processed at conversion interrupt.
ONCHAR	The character which caused conversion interrupt.
ONKEY	The key of logical record which caused interrupt.
ONCODE	A code specifying type of error.
DATAFIELD	The incorrect NAME on data-directed input.

List-Processing Built-In Functions

ADDR(X)	Current address of data variable X.
EMPTY	A storage area of zero size, containing no allocations.
NULL	Null value for pointer. End of chain.
NULLO	A null offset value.

Other Built-In Functions

FUNCTION (ARGUMENTS)	VALUE RETURNED
DATE	Character string of form YYMMDD, where YY represents the year, MM the month, and DD the date.
TIME	Character string of form HHMMSSTTT, where HH represents the hour, MM the minutes, SS the seconds, and TTT the milliseconds.
ALLOCATION(X)	If storage has been allocated for X, the value returned is 1; if not, the value returned is 0.
LINENO(X)	The current line number of PRINT file X.
COUNT(X)	The number of items transmitted during the last GET or PUT on file (X).
ROUND(X,I)	The expression X, rounded on the Ith digit after the point.
STRING(X)	The structure X, converted to a string.
COMPLETION(X)	0 or 1, depending on the status of EVENT (X).
PRIORITY(X)	The priority of task relative to the task containing the function.
STATUS(X)	Zero or non-zero, depending on whether the status of event X is normal or abnormal.

Appendix 2: ON Conditions

The following conditions must be enabled by a prefix for an interrupt to occur:

CONDITION NAME	DESCRIPTION
CHECK (identifier list)	The listed variable is altered or the labeled statement executed.
SIZE	Loss of significant digits on the left of assignment.
SUBSCRIPTRANGE	Subscript out of range.
STRINGRANGE	Invalid attempt to use SUBSTR built-in function.

The following conditions are normally enabled, but may be disabled by NO prefix:

OVERFLOW	Floating-point result too big.
UNDERFLOW	Floating-point result too small.
ZERODIVIDE	Fixed- or floating-point division by zero.
CONVERSION	Error in converting to a different data type.
FIXEDOVERFLOW	Loss of left-hand significant digits during fixed-point expression -evaluation.

The following conditions are always enabled and cannot appear in prefix lists:

ENDFILE(X)	End-of-file on file X.
ENDPAGE(X)	End-of-page on PRINT file X.
TRANSMIT(X)	Permanent transmission error on file X.
UNDEFINEDFILE(X)	File named is incorrectly specified or cannot be OPENed.
NAME(X)	Unrecognized name on data-directed input on file X.
KEY(X)	Incorrect KEY given on file X.
RECORD(X)	Record on file X is incorrect length.
AREA	Attempts to allocate more storage than exists in an area.
CONDITION(X)	X is programmer-named condition raised by signal statement.
FINISH	Termination of main procedure by STOP, RETURN, END or EXIT.
ERROR	Termination of major task due to error.

Appendix 3: Correspondence of FORTRAN and PL/I Statements

FORTRAN	PL/I
DIMENSION	DECLARE
COMMON	DECLARE with EXTERNAL
EQUIVALENCE	DECLARE with DEFINED or CELL
DATA	DECLARE with INITIAL
DOUBLE PRECISION	DECLARE with Precision
INTEGER	DECLARE with FIXED
REAL	DECLARE with FLOAT or REAL
COMPLEX	DECLARE with COMPLEX
LOGICAL	DECLARE with BIT
Assignment	Assignment
GO TO	GO TO
ASSIGN	Assignment with Label Data
IF	IF
DO	DO
STOP	STOP

FORTRAN	PL/I
PAUSE	DISPLAY
CONTINUE	Not needed
CALL	CALL
RETURN	RETURN
FUNCTION	PROCEDURE
SUBROUTINE	PROCEDURE
ENTRY	ENTRY
READ	GET or READ
WRITE	PUT or WRITE
FORMAT	FORMAT
ENDFILE	CLOSE
REWIND	CLOSE
BACKSPACE	No equivalent
BLOCKDATA	DECLARE with EXTERNAL and INITIAL
EXTERNAL	DECLARE with ENTRY
NAMELIST	GET/PUT with data-directed I/O

Index

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

SC20-1637-3

A Guide to PL/I for FORTRAN Users

Printed in USA SC20-1637-3