IBM

Student Text

A PL/I Primer

Preface

The purpose of this publication is to provide tutorial material not only for the person with some knowledge of computer programming, but also for the novice who knows little or nothing about data processing.

The first chapter is written solely for the novice. A reader who is familiar with basic programming techniques should skip the first chapter, and begin reading at Chapter 2, "Basic Elements of PL/I."

Chapter 1, "Communicating with a Computer," touches on machine language and introduces the concept of symbolic programming. The basic techniques of programming are illustrated by using symbolic instructions, rather than PL/I, because certain PL/I statements can generate so much single-instruction coding that the details of some of the techniques are hidden.

Further information concerning PL/I can be found in the following publications:

IBM Operating System/360, PL/I: Language Specifications, Form C28-6571

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

Copies of this and other IBM publications can be obtained through IBM branch offices. Address comments concerning the contents of the publication to IBM, Technical Publications Department, 112 East Post Road, White Plains, N. Y. 10601

Contents

Introduction

In the past, throughout the data processing field, certain computers generally were identified with a particular field of activity, either scientific or commercial. Programming languages were specialized in the same way. FORTRAN was developed for scientific programming and COBOL for commercial programming.

Today, computing systems are designed for a broader range of activity. The new computers are faster and more powerful. They serve the scientific and commercial programmer equally well, and they provide facilities for many new programming techniques. None of the older languages can take advantage of all the power of the new computers, and more and more computer installations are handling both scientific and commercial programming.

These are the reasons for PL/I. It is a multipurpose programming language that can be used by both commercial and scientific programmers to handle all of their programming work and to give them the widest range of control over the computer.

PL/I has been designed so that any programmer, no matter how brief or extensive his experience, can use it easily at his own level. It is simple for the beginning programmer; it is powerful for the experienced one.

A programmer need not know everything about PL/I to be able to use it. An experienced programmer can use PL/I to specify almost every detail of every step of a highly complicated program. A beginner can take advantage of the many automatic features of the language to do much of his work for him.

The language provides many options in statements, in descriptions of data or files. Wherever there are alternatives, the compiler makes an assumption if no choice is stated by the programmer. In each case, the assumption, called a *default*, is the alternative that would be required in the majority of situations. The default concept is an important part of the simplicity of PL/I. In many cases, a beginning programmer need not even know that alternatives exist.

PL/I was developed by IBM in conjunction with representatives of two customer groups: SHARE, the scientific users' organization and GUIDE, the commercial users' organization.

In the few years since the first use of computers, the field of data processing has been explosive in its growth.

The computer began as a machine to aid the mathematician. Today, computers are used in almost every science and industry. New computer applications are being discovered; new programming developments will provide simpler ways of solving problems.

A computer, however, has no mystic or magical power. It is merely a tool. Granted, a computer is a tool that operates swiftly and accurately; yet as a hammer cannot drive a nail by itself, a computer, undirected, cannot function.

A computer is not an electronic "brain." It has no "memory," merely areas that can be magnetized in a way to represent information.

A computer has no discrimination. It will add apples and oranges without hesitation. In most computers, all numbers, letters, and characters are represented internally as expressions of binary digits (1 and 0). (See the section "Binary and Decimal Notation" at the end of this chapter.) Although various combinations of binary digits, or bits, may arbitrarily have been given meanings, the computer can consider any piece of data only one bit position at a time.

A computer has no judgment. It can make no decisions except to choose one of two alternatives presented to it; even then, the choice is dependent upon the value of one or more binary digits the computer has been directed to examine.

A computer can do only what it is told to do and only in the way it is told to do it. Specific electronic stimuli cause specific electronic reactions. In effect, a computer responds to an *instruction.*

Each instruction causes a certain action to be taken by the computer. A series of instructions can be written to direct a computer to take all the necessary action, step by step, to solve a problem—to receive data, to process it, and to return the results. Such a series of instructions is called a *program.* The notation in which a program is written is called a *programming language.*

Programming Languages

A programming language is not used to "talk with" a computer. The only programming language a computer can respond to—*machine language*—is hardly a language at all. Any programming language that resembles a human language first must be translated into machine language before it can have any effect upon the operation of a computer.

Machine Language

Each different kind of computer has its own machine language, which has been built in by its designers. Figure 1 illustrates a hypothetical, but representative, machine-language instruction as it would appear when punched into a card.

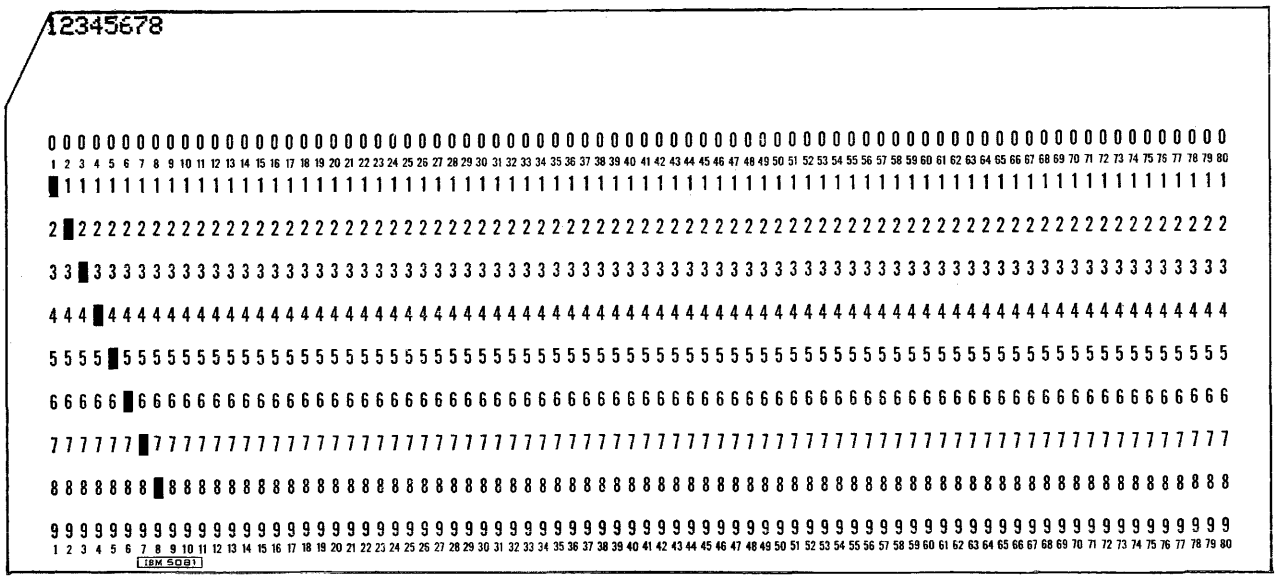


Figure 1. A Typical Machine-Language Instruction.

Assume that a computer has been engineered to recognize this instruction as indicating the arithmetic function of addition, and that the instruction actually consists of three elements, or *fields*. The fields are 12, 345, and 678. The first field, 12, is the *operation code* that instructs the computer to add. The other two fields, 345 and 678, represent specific locations in the main storage area of the computer. They are *addresses* of locations where data is recorded, or stored.

When the computer executes the instruction, the data item stored in storage location 345 is added to the data item stored in location 678. In this case, the sum replaces the item that originally had been stored in location 678.[1]

Obviously, programming in machine language is difficult. A programmer would have to memorize the complete list of operation codes or continually consult a reference list.

More difficult than handling the operation codes in machine language would be the problem of keeping track of the location of different data items. Even the smallest computers have thousands of data locations, each represented by a separate address. A machine-language programmer would have to construct a complete address list, noting the location of each data item; the list would change each time an item is moved from one location to another.

A computer can be programmed to consult an operation code table and keep track of data addresses. Programs can be written using recognizable symbols, and the computer can be instructed to translate this symbolic language into machine language.

Assembly Language

A program that instructs a computer to translate a language, instruction by instruction, into machine code is called an *assembler*. The program to be translated, or *assembled,* is written in *assembly language.*

An assembly language is a programming language that has a different, recognizable symbol to represent each machine operation code, symbols like ADD, MOVE, or READ. The assembler instructs the computer to substitute the equivalent machine-language operation code for each assembly-language operation code. Using the hypothetical computer discussed previously, the assembler would instruct the computer to substitute the machine-language operation code 12 whenever it encounters the word ADD.

The handling of data location addresses by an assembler demonstrates even greater ingenuity. There is no set list of names for the different locations. The programmer chooses any name he wishes. Each name, of course, must be different from all other names in any one program, and each must be used consistently for the same reference.

The machine-language instruction 12345678 might, in assembly language, be written ADD FIRST TO SECOND, although more commonly the word TO is replaced by a comma. Thus, the instruction might be ADD FIRST, SECOND. In this case, FIRST and SECOND actually are names that represent *addresses* of locations where data is stored in the computer. In concept, however, they can be considered to be names of the data items themselves. So a programmer usually chooses such descriptive names as PAYRATE or COEFFICIENT.[2]

The assembler causes the computer to construct a table of names much like the table of operation codes, except that a machine-language address is associated with each name. Consequently, a programmer need not be concerned with where data actually is located; each time a data name appears, the computer substitutes the machine-language address of the location where the data item is to be stored.

Figure 2 illustrates a program as it might be written using an assembly language. Although the purpose of this book is to explain the use of PL/I as a programming language, the details of some basic programming concepts can be more easily illustrated in assembly language coding, since PL/I handles so much programming detail automatically. An understanding of these basic concepts will make it easier to learn PL/I.

Note that there are names, or *labels* written to the left of some of the instructions. Their purpose is to identify the particular instruction. When the assembled program is loaded into the computer for execution, the machine-language instructions actually occupy storage locations within the computer, just as data does. Thus, the label of an instruction is translated by the assembler into the *address* of its associated instruction.

The program is designed to compute the monthly sales of a company and to print the result, listing total sales for each salesman, as well as total company sales. Each card that is read represents a single sale. It lists the salesman's number and amount of the sale. The cards are sorted according to salesman number; consequently, all sales cards for each salesman are together.

Unless specified otherwise, a program is executed sequentially, instruction by instruction. If the sequence of

[1] This type of operation, called *storage-to-storage operation,* often makes it necessary for a programmer to establish a particular storage location for use as a work area; otherwise, the data item that is replaced by the result of a computation might be irretrievably lost. Some computers are programmed using special registers where computations take place, thus avoiding this problem. Some computers use both arithmetic register operations and storage-to-storage operations, but illustrations in this section assume a computer with no such registers.

[2] With most assembly languages, the operation codes are abbreviated to one or two letters. Thus, A might be the operation code for "add," M for "multiply," and MV for "move." In the same way, there are usually restrictions on the number of letters allowed in names, often six or seven. However, to add clarity to this discussion, operation codes and names are completely spelled out.

LABEL	OPERATION CODE	OPERANDS	COMMENTS	INSTRUCTION NUMBER
START	READ	CARD, READAREA	READ FIRST CARD.	1
	MOVE	SALES, MONTHLYTOTAL	MOVE SALES FIGURE TO STORAGE AREA FOR MONTHLY TOTAL.	2
LOOP2	MOVE	SALESNO, SALESNUMBER	SET SALESMAN'S NUMBER.	3
	MOVE	SALES, MANTOTAL	MOVE SALES AMOUNT TO SALESMAN TOTAL.	4
LOOP1	BRANCH LAST CARD	PRINT2	IF LAST CARD HAS BEEN READ BRANCH TO PRINT2;	5
	READ	CARD, READAREA	OTHERWISE, READ NEXT SALESCARD.	6
	COMPARE	SALESNO, SALESNUMBER	COMPARE SALESMAN NUMBER ON CARD WITH SALESMAN NUMBER FROM LAST RECORD.	7
	BRANCH UNEQUAL	PRINT1	IF NOT THE SAME, BRANCH TO PRINT1;	8
	ADD	SALES, MANTOTAL	OTHERWISE, ADD SALES TO SALESMAN TOTAL.	9
	ADD	SALES, MONTHLYTOTAL	ADD SALES TO MONTHLY TOTAL.	10
	BRANCH	LOOP1	GO BACK TO READ NEXT CARD.	11
PRINT1	WRITE	SALESNUMBER, MANTOTAL	WRITE SALESMAN NUMBER AND HIS TOTAL SALES.	12
	ADD	SALES, MONTHLYTOTAL	ADD SALES TO MONTHLY TOTAL.	13
	BRANCH	LOOP2	GO BACK TO SET NEW SALESMAN NUMBER.	14
PRINT2	WRITE	SALESNUMBER, MANTOTAL	WRITE REPORT FOR LAST SALESMAN.	15
	WRITE	MONTHLYTOTAL	WRITE COMPANY TOTAL.	16
	END			

Figure 2. A Hypothetical Assembly Language Program

execution is to be changed, the programmer can specify that execution branch to a label; the instruction having that label becomes the next instruction executed. In Figure 2, for example, when instruction 11 is executed, control is transferred back to instruction 5.

The comments to the right of the instructions are not part of the program. They are a part of the programmer's own documentation to indicate the reason for each instruction. Likewise, the numbers to the right are merely for the programmer's convenience in numbering the instructions.

This program, called a *source* program, shows what the programmer might write and what would be punched into cards, one instruction per card. The assembler instructs the computer to read the source program, to translate each instruction into machine language, and to punch the *assembled* program into another deck of cards or to write it on magnetic tape or on some other form of storage. The translated program is called the *object program;* it can be loaded into the computer and executed.

At the start of the program shown on Figure 2, the first sales card is read and recorded inside the computer in READAREA. Since a card has 80 columns and each column represents one character, READAREA takes up 80 locations in storage (unpunched columns are recorded as blanks).

The second instruction moves the sales amount to the area where the monthly total is to be computed. In a computer, a move instruction might more accurately be described as a "copy" instruction, since the data item is not actually removed from its original location. Consequently, instruction 4 also can "move" the sales figure to the area where the individual salesman's total is to be computed. A programmer must always remember that data used in a previously executed program may still be

recorded within the computer; it must not be assumed that any location in storage is blank or zero unless the computer is instructed to make it so. A move instruction causes replacement of any data left in the storage area; data moved into the location replaces data that had been there before.

Instruction 3 moves the salesman number from READAREA to the locations where it will be stored. All of the information on the card that is necessary to this program has now been stored, and the next card is read and recorded in READAREA, completely replacing the data that had been recorded there when the first card was read.

Instructions 5 through 11 are the heart of the program. They all will be executed repeatedly, once for each sales card except the first sales card for each salesman. Such repetitive execution is called *looping*.

Instruction 5 is one that reveals much about computer programming. It means "test to see if the last card has been processed; if so, branch to PRINT2." Although the condition being tested can occur only once during the entire execution of the program, the test is made before each READ instruction, except the first. The location of this instruction in the sequence of execution is critical. It must be executed only when computations are complete for all data that has been read up to that point.

After each card is read in LOOP1, instruction 7 calls for a test to see if the salesman's total sales have been figured. It directs the computer to compare the salesman number on the card just read with the salesman number saved in storage. If they agree, the other instructions in the loop are executed, and instruction 11 causes a branch back to instruction 5 to repeat the loop.

If, however, the salesman number on the card is different from the salesman number recorded at SALESNUMBER, it indicates that the total sales have been computed for the salesman whose number is recorded at SALESNUMBER. In that case, instruction 8 causes control to branch to instruction 12, labeled PRINT1.

The salesman's number is written, along with the salesman's total sales. "Writing," in this case, could mean printing the actual report, or it could mean writing magnetically, on tape, for later printing. The sales figure is added to the monthly total, and control branches back to LOOP2 to initialize SALESNUMBER and MANTOTAL for the next salesman's records. When the last card is read and tabulated, totals are written for the last salesman and the monthly sales total is written.

During assembly of the program shown in Figure 2, each instruction is translated directly into a single machine-language instruction that is executed by the computer when the object program is run. Instructions of this type are called *executable* instructions.

Another kind of instruction – not shown in Figure 2 but necessary to every assembly-language program—is the *non-executable* instruction. Non-executable instructions are those that give information needed by the assembler in the construction of the table of names and the assignment of addresses.

For example, the READ instructions indicate that data read from cards is to be recorded in READAREA. When the program is executed, READAREA might be at location zero or at location 10, 22, or anywhere else in storage; it is of no concern to the programmer. But he must make certain that READAREA, wherever it may be, is a field of 80 locations in length, one location for each card column. The non-executable instruction describing READAREA might be written as follows:

```
READAREA    DS    CHAR80
```

The operation code DS means "define storage"; CHAR80 indicates that the area defined is to be large enough—but no larger than needed—to store 80 characters of data. When it encounters this instruction, the assembler assigns a machine-language address to READAREA and skips the addresses of the next 79 character locations before making another address assignment. If READAREA is assigned to location 100, for example, and the above instruction is followed by SALESNUMBER DS CHAR8, the address assigned to SALESNUMBER would be 180. The next storage assignment after that would be 188.

In the illustrated program, certain positions of the area into which cards are to be read must be redefined so that individual items of data may be handled separately. The salesman's number and the sales figure must be described as they will appear on the card and as they will be recorded in READAREA. Different computers provide different methods of doing this. One method is illustrated by the following instructions:

```
READAREA    DS    0CHAR80
SALESNO     DS    CHAR8
SALES       DS    CHAR6
```

In the first instruction, the zero preceding CHAR80 indicates that READAREA, which is a total of 80 characters in length, will be redefined and that different portions of it will be given different names. When using this method of redefining a storage area, a programmer must account for the entire area. Since SALESNO and SALES require a total of only 14 locations, the remaining 66 locations in READAREA must also be redefined. This could be done as follows:

```
            DS    CHAR66
```

This redefined area need not be given a name if it is not to be used in the program.[1]

[1] This portion of the card also may contain data; in this case it might be a description of the merchandise sold or the name and address of the buyer. Although this information is not necessary in figuring total sales, it might be applicable to some other program in which the same input cards could be used. The entire card is read, and all data on it is recorded in storage. Unused data is merely ignored by the program.

```
READAREA        DS      0CHAR80   READ AREA
SALESNO         DS      CHAR8     SALESMAN NUMBER ON CARD
SALES           DS      CHAR6     SALES LISTED ON CARD
                DS      CHAR66    REMAINDER OF READ AREA
SALESNUMBER     DS      CHAR8     SALESMAN NUMBER IN STORAGE
MANTOTAL        DS      CHAR10    SALESMAN TOTAL SALES
MONTHLYTOTAL    DS      CHAR14    MONTHLY TOTAL SALES
```

Figure 3. Non-Executable Instructions

All of the data names used in a program must be described. The non-executable instructions are usually kept together, either preceding or following the list of executable instructions.

Figure 3 illustrates all of the non-executable statements required to complete the program shown in Figure 2.

Macro Instructions

Another kind of instruction that gives information to the assembler is the *macro instruction.* A macro instruction instructs the assembler to provide a predetermined *sequence* of instructions instead of a single machine instruction. It might be a commonly used sequence that is "built in" as a part of the assembly program (for example, a sequence to round a currency value to the nearest penny), or it might be a particular sequence of instructions that the programmer could specify.

Assume that a programmer writes a program that requires several simple multiplication operations using numbers that represent dollars and cents. Also assume that the product of multiplication replaces the number being multiplied; thus, it is first necessary to move this number to a work area, compute the product, and then move the answer to the desired location.[1] The following instructions are needed each time the operation is used:

```
MOVE        MULTIPLICAND, WORKAREA
MULTIPLY    WORKAREA, MULTIPLIER
MOVE        WORKAREA, PRODUCT
```

A macro instruction can simplify program writing and card punching. At the beginning of the program a *macro definition* such as the following is written:

```
PRODUCT     MACRO       A, B, C
            MOVE        A, WORKAREA
            MULTIPLY    WORKAREA, B
            MOVE        WORKAREA, C
            END         PRODUCT
```

The letters A, B, and C, which could be any names or symbols a programmer chooses, are called *parameters.* As they appear in the coding, they represent positions in the instructions where other names will be substituted. If a programmer wants to compute gross pay based on hourly rate of pay and hours worked, the following macro instruction could be written:

```
PRODUCT     PAYRATE, HOURS, GROSSPAY
```

[1] This method is used here to provide a clearer explanation of a macro instruction. The product could be computed in place; the multiplicand could first be moved to the location where the product is to be stored. During multiplication, the product would replace the multiplicand and would be properly located.

The assembler would supply instructions as written in the macro definition of PRODUCT and would substitute—in each instruction—PAYRATE for the letter A, HOURS for the letter B, and GROSSPAY for the letter C.

The preceding macro instruction might be followed by:

```
PRODUCT     GROSSPAY, TAXRATE, TAX
PRODUCT     GROSSPAY, SOCIALSECURITY, FICA
```

In other words, the same macro instruction can be used to figure taxes and social security payments to be withheld. In each case, one written instruction in the source program provides three machine-language instructions in the object program.

An engineer might use the same macro instruction to compute wattage and voltage, writing the instructions as follows:

```
PRODUCT     AMPS, OHMS, VOLTS
PRODUCT     VOLTS, AMPS, WATTS
```

Note that in each macro statement the names to be substituted, called *arguments,* are listed in the same order as the corresponding parameters are listed in the first line of the original macro definition.

A programmer working with currency values probably would want the product of multiplication accurate to the nearest cent. Assume that the assembler has a built-in macro instruction that supplies two machine-language instructions to handle this operation. The programmer might write:

```
ROUND       GROSSPAY
ROUND       TAX
ROUND       FICA
```

Or, the built-in macro instruction might be inserted into the programmer's macro definition. The macro definition could be:

```
PRODUCT     MACRO       A, B, C
            MOVE        A, WORKAREA
            MULTIPLY    WORKAREA, B
            ROUND       WORKAREA
            MOVE        WORKAREA, C
            END         PRODUCT
```

If the macro instruction PRODUCT were defined this way, each use of PRODUCT as a macro instruction would include the use of ROUND as a macro instruction, and the computed answer would always be rounded to the nearest cent. A single instruction written in the program would then supply five machine-language instructions—three included in the programmer's definition and two supplied by the ROUND macro instruction.

High-Level Languages

As seen with the use of macro instructions, it is possible to write one instruction that can direct a computer to do a number of operations; one *statement* can represent several instructions. In this manner, a high-level or machine-independent language resembles a series of macro instructions.

An assembly-language programmer must always consider the specific computer being programmed. Macro instructions can free the programmer from some of the machine detail; yet he must be aware of the instructions that are supplied by each macro instruction. Idiosyncracies of the computer can get in the way of programming; details of writing the program can interfere with the programmer's concentration on the solution of the problem.

A high-level language can free a programmer from the detailed job of writing each individual instruction. A high-level language, although it may require strict conformance to syntactic rules, is a programming language that reads much like English or like mathematical notation. Each statement in a high-level language describes the action of the step to be taken.

The macro instruction just discussed would be written in PL/I as the following statements:

```
GROSSPAY=PAYRATE * HOURS;
TAX=GROSSPAY * TAXRATE;
VOLTS=AMPS * OHMS;
WATTS=VOLTS * AMPS;
```

NOTE: The convention in most high-level languages concerning arithmetic operators is to use the plus sign and the minus sign in their conventional mathematical sense. An asterisk (*) denotes multiplication; a slash (/) denotes division; and a double asterisk (**) denotes exponentiation, that is, 2**4 indicates two raised to the power of four. The mathematical equal sign (=), however, may have either of two meanings, depending upon how it is used. When used as in the preceding statements, it is called an *assignment symbol* (the statements are *assignment statements*), and it means "assign the value of the expression on the right of the (=) symbol to the name on the left of the (=) symbol," or "set the value of the variable on the left equal to the value of the expression on the right." In other uses, such as IF A=B..., the equal sign means the same as the mathematical equal sign.

A high-level language, like assembly language, requires a special program to direct the computer to examine each statement and to expand it and convert it into the necessary machine instructions. Such a program, called a *compiler*, is much larger and more detailed than an assembler. A source program translated by a compiler is said to be compiled. A compiler frees the programmer from the details of assembly-language coding as an assembler frees him from the details of machine-language coding. The programmer need have little concern for specific conventions of a particular computer. A high-level-language program written for use with one computer may, without revision or, with minor revision, be compiled for an entirely different computer, even when the actual machine-language instructions and internal representation of the two computers differ completely.

A high-level language allows a programmer to concentrate on the problem at hand; each statement becomes a broad step toward its solution.

Binary and Decimal Notation

The binary digit—also called *a bit*—is the basic unit in a digital computer. From an electronic standpoint, it can be considered to be a switch; it is either "on" (1) or "off" (0). Different computers may recognize certain combinations of binary digits differently, but most of them are concerned with binary notation, which uses a number system based on the value 2.

In decimal notation, the positions, moving left from the decimal point, are units, tens, hundreds, thousands, etc.; moving right from the decimal point, they are tenths, hundredths, thousandths, etc. In binary notation, moving left from the binary point, the positions are units, twos, fours, eights, sixteens, etc., indicating multiplication by two. Moving right from the binary point, the positions are halves, quarters, eighths, sixteenths, etc., indicating division by two.

For example, the decimal number 2304.301 might be analyzed as follows:

2 times 10^3 (1000)	2000
3 times 10^2 (100)	300
0 times 10^1 (10)	00
4 times 10^0 (1)	4
3 times 10^{-1} (0.1)	0.3
0 times 10^{-2} (0.01)	0.00
1 times 10^{-3} (0.001)	0.001
Total	2304.301

In the same way, the binary number 11011.011 could be analyzed as follows:

1 times 2^4 (16)	16
1 times 2^3 (8)	8
0 times 2^2 (4)	0
1 times 2^1 (2)	2
1 times 2^0 (1)	1
0 times 2^{-1} (½)	0
1 times 2^{-2} (¼)	¼
1 times 2^{-3} (⅛)	⅛
Total	27⅜

In most programming languages, the length of an individual instruction or statement depends upon the span of a single punched card. If a statement exceeds the limit of one card, a notation must be made, usually with a punch in some particular card column, to indicate that the statement is continued on the following card.

Although most input to a computer begins with a punched card, input often is transferred to magnetic tape before it actually is introduced into the computer for processing. In many applications, statements and data are introduced into the computer directly from the keyboard of a typewriter input terminal. Consequently, the limitation of a single card, either actual or symbolic, is artificial.

PL/I has no such artificial limitation. There is no fixed-length format for input although a compiler may reserve some card columns. Within the available card area, PL/I can be written in free form; the computer recognizes a continuous stream of input. Just as a period indicates the end of this sentence, a semicolon indicates the end of each PL/I statement. The next statement may begin immediately in the next available location, whether card column, tape character, or typewriter space, or any number of blanks may intervene.

Character Set

There are 60 characters in the PL/I language. These include:

An extended alphabet of 29 characters; the currency symbol ($), the commercial "at" sign (@), and the number sign (#) precede the 26 letters of the English language alphabet.

The ten digits 0-9.

The following 21 special characters:

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

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

A special 48-character set also is provided for use in place of the 60-character set. This set is shown in Appendix 1. In all but three cases, the characters of the reduced set can be combined to represent missing characters from the larger set. The percent symbol (%) for example, is not included in the 48-character set, but a double slash (//) can be used to represent it. The three characters that are not duplicated are the commercial "at" sign, the number sign, and the break character.

The rules for PL/I sometimes specify that an *alphameric* character must be used in certain coding. The term alphameric refers to any of the 29 alphabetic characters and the 10 digits, but not to the 21 special characters.

Comments

Programmers frequently insert comments into their programs to clarify the action that occurs at a given point. These comments enable someone unfamiliar with the program to follow the programmer's line of thought and are helpful to the programmer when looking back over program sections that were written earlier.

Comments are permitted wherever blanks are allowed in a program. They may be punched into the same cards as statements, be inserted between statements, or appear in the middle of statements without affecting compilation of the program.

The character pair, /*, indicates the beginning of a comment. The same characters reversed, */, indicate its end. No blanks or other characters can separate these two characters; the slash and the asterisk must be immediately adjacent. The comment itself may contain

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

any characters except the */ combination, which would be interpreted as terminating the comment.

```
    /* THIS WHOLE SENTENCE COULD BE
INSERTED AS A COMMENT. */
    /* SO COULD @#$¢%&*()_-THIS. */
```

Any characters recognized by the user's system hardware may be used in comments. This includes characters that are not in the PL/I character set, such as the cent sign in the second example above.

Identifiers

An identifier is a combination of alphameric characters and break characters used in a program as names of data items, files, and special conditions, and as labels for statements. The actual words of the language, such as READ, WRITE, GO TO, etc., also are identifiers (such language words, called *keywords,* when used in proper context, have a specific meaning to the compiler; they specify such things as the action to be taken, the nature of the data, the purpose of a name).

An identifier must begin with one of the 29 alphabetic characters and cannot contain blanks. No identifier can exceed 31 characters in length; some compliers may further restrict the length for certain kinds of identifiers. Following are examples of identifiers:

A	BINARY
LOOP	WRITE
SALESNUMBER	FILE2
XR25	#1200
DECIMAL	PAY_NUMBER

The last of these examples (PAY_NUMBER) illustrates the use of the break character to improve the readability of an identifier, since blanks are not permitted.

Data

Data is generally defined as a representation of information or values. Digits and characters are data; however they are used, they always represent values.

A programmer is concerned with several different levels of data, different representations of the same values:

1. Raw data, the values to be processed and the information that states the problem to be solved
2. The representation of values as the programmer writes it in his program
3. Compiler input data, the representation as it is punched into a card or is entered from a typewriter terminal; this data is translated into machine-language data
4. Internal data, the representation as it is maintained inside the computer

To avoid ambiguity, the unqualified word "data," when used in this publication, refers to the representation of values as written in a PL/I program.

Reference to a data item, numeric or alphabetic, is made by using either a *variable* or a *constant* (the terms are not exactly the same as in general mathematical usage).

A variable is a symbolic name—an identifier—having a value that may change during execution of a program. Since it is a name, a variable is not, in itself, a data item; the value of a variable at any specific time is the data item to which it refers at that time.

A constant, which is not given a symbolic name, is unchanging; the data item is its name. Consider the following statement:

```
AREA = RADIUS**2 * 3.141593;
```

AREA and RADIUS are variables, the numbers 2 and 3.141593 are constants. Thus, 3.141593 is the data item; the characters 3.141593 also are written to refer to the data item.

If the constant 3.141593 is to be used in more than one place in the program, the programmer probably would give it a name, probably PI, and would write the statement as follows:

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

In this statement only the digit 2 is a constant; PI is a variable. Since PI is a data name, the programmer can change the value it represents. The value of a variable can remain constant throughout execution of a program; nevertheless, it is still a variable.

The constant does more than state a value; it demonstrates various characteristics of the data item. For example, 3.141593 shows that the data item is a decimal number of seven digits and that six of these digits are to the right of the decimal point.

The characteristics of a variable are not immediately apparent in the name. Since these characteristics, called attributes, must be known, certain keywords are used to describe the attributes of a variable when it first appears in the program. The method used for such data description is discussed in a later chapter.

Expressions

Any identifier, other than language keywords, written in a PL/I program is called an *expression.* An expression may be a single constant or a name, or it may be a combination of them, including operators and other delimiters.

An *arithmetic expression* combines arithmetic data identifiers and arithmetic operators. Arithmetic expressions may involve addition (A + B), subtraction (A − B), multiplication (A * B), division (A / B), and exponentiation (A ** B, or A raised to the power of B).

A number of arithmetic operations may be included in a single expression. For example:

```
A + B − C * (D / (E − F)) ** G
```

Parentheses within an expression indicate that the pa-

renthesized portion is considered as a single value in relation to its surrounding arithmetic operators. The parenthesized portion of an expression is evaluated first, with innermost parenthesized material taking precedence. In the preceding example, D is to be divided by the value of E minus F; the value obtained by this division is to be raised to the power of G; and C is to be multiplied by the value obtained.

Although an expression may contain more than one data item, it represents the single value obtained after the expression is evaluated.

The following section, "Writing a PL/I Program," illustrates how PL/I statements are written and how each interacts with other statements. A specific problem is stated, and a program is constructed step by step, to direct the computer to solve the problem. The illustrated program is not the only one that would solve the problem; it may not be the best one. It has been chosen to demonstrate the ease with which a PL/I program can be written.

Chapter 3: Writing a PL/I Program

A program consists of all the statements necessary to instruct a computer to solve a problem – to get data, to process it, and to return the results of the processing to the programmer.

In analyzing a problem to be solved, a programmer first must examine the data and determine the manipulations necessary to process that data.

Consider a problem of computing interest on a loan. The basic computation would be to multiply the principal of the loan by the interest rate. It could be written:

```
PRINCIPAL * RATE
```

In the program, the value of the expression PRINCIPAL * RATE could be assigned to a variable written in an *assignment statement.*

```
INTEREST=PRINCIPAL * RATE;
```

Assume that the program is to compute the interest on numerous bank loans of different denominations at varying rates of interest. The interest is compounded monthly, and in most cases, the borrower makes a monthly payment to cover the interest charge and to reduce the principal of the loan.

The programmer need write only two assignment statements to make the necessary computations:

```
INTEREST=PRINCIPAL * RATE/12;
BALANCE=PRINCIPAL+INTEREST-PAYMENT;
```

The first assignment statement computes the monthly interest charge; the second statement compounds the interest and deducts the payment to figure the balance due.

Before the computations can be made, however, the computer must get the data; that is, it must have the values for PRINCIPAL, RATE, and PAYMENT.

Certainly the bank would have a record of each loan. Assume that each record consists of an identifying serial number, the current principal of the loan, and the interest rate. Assume, further, that the records of all the loans are collected as a set of data into a master file, arranged in ascending order according to serial number.

A separate set of data is prepared each month listing the payments made. Each record consists of the serial number and the amount of the payment. Records in this data set are in the same order as those in the master file.

The two GET statements required to obtain the data are:

```
GET FILE (INPUT) LIST (PAY_#, PAYMENT);
GET FILE (MASTER) LIST (LOAN_#,
    PRINCIPAL,RATE);
INTEREST=PRINCIPAL * RATE/12;
BALANCE=PRINCIPAL+INTEREST-PAYMENT;
```

The first GET statement indicates that the data is to be read from the file named INPUT and that the first data item is to be assigned to PAY_#, and the second data item to PAYMENT. The second GET statement indicates that the data is to be read from the file MASTER and that the list of data items is to be assigned to LOAN_#, PRINCIPAL, and RATE. The data might be read from punched cards, from magnetic tape, or from any other medium upon which data is recorded.

After the data is read, the assignment statements are executed, and the results of the processing are ready to be returned.

A PUT statement now is added, to write the results. A second PUT statement directs the creation of an updated master file. The two statements are:

```
GET FILE (INPUT) LIST (PAY_#,PAYMENT);
GET FILE (MASTER) LIST (LOAN_#,
    PRINCIPAL,RATE);
INTEREST=PRINCIPAL * RATE/12;
BALANCE=PRINCIPAL+INTEREST-PAYMENT;
PUT FILE (OUTPUT) LIST
    (LOAN_#,PRINCIPAL,INTEREST,PAYMENT,
    BALANCE);
PUT FILE (NEW_MASTER) LIST
    (LOAN_#,BALANCE,RATE);
```

The first PUT statement directs the computer to write, in the file named OUTPUT, the current value of the following list of variables: LOAN_#, PRINCIPAL, INTEREST, PAYMENT, and BALANCE. Since actual printing is a relatively slow process, the data probably would be written on magnetic tape and actually printed as another program.

By the time all of the records in MASTER have been processed, the second PUT statement will have created a new master file that can be used as MASTER when the program is run the following month (Note that the value of BALANCE will be the value of PRINCIPAL when the new master file is read).

At this point, the statements provide all of the actual instructions necessary to read, compute, and write the information for a single loan record. But, since the objective is to process many loans, the program must

```
NEW_RECORD: GET FILE (INPUT) LIST (PAY_#,PAYMENT);
            GET FILE (MASTER) LIST (LOAN_#,PRINCIPAL,RATE);
            INTEREST=PRINCIPAL * RATE/12;
            BALANCE=PRINCIPAL+INTEREST-PAYMENT;
            PUT FILE (OUTPUT) LIST (LOAN_#,PRINCIPAL,INTEREST,PAYMENT,BALANCE);
            PUT FILE (NEW_MASTER) LIST (LOAN_#,BALANCE,RATE);
            GO TO NEW_RECORD;
```

Figure 4.

function as a loop; control must be transferred from the last of the sequence of statements back to the first.

A transfer of this type can be made only to a statement having a *statement label*, which is a name chosen by the programmer and prefixed to the statement by means of a colon. Any name might be used. In this case, a programmer might choose the name NEW_RECORD, since the first GET statement directs the computer to get a "new record."

Now, a GO TO statement can be used to transfer control. See Figure 4.

As the program now stands, it might seem that all of the statements necessary to read, compute, and write have been included. This presumes, however, that no conditions will vary, that a payment has been made for each loan in the master file, that no loan has been overpaid, even that no loan has been completely repaid.

Since a programmer generally cannot examine intermediate results during execution of a program, he must anticipate situations in which unwanted or meaningless results might be obtained. He must tell the computer what to do when a decision must be made. His program must include instructions to the computer to make certain tests and, based upon the results of each test, to select one of two alternative actions to be taken.

For example, if a loan has been paid in full, the record of that loan should not appear in the updated master file. Before an updated record is written, the balance must be tested, and instructions must be provided to tell the computer what to do, based upon the result of the test.

In PL/I, an IF statement is used to specify a test to be made and to give two alternative actions.

Figure 5 shows how a programmer might write an IF statement as a test for a paid-up loan.

Note that no semicolon is used after the first portion of the IF statement. The IF statement is a *compound statement*; that is, it is a statement that contains other statements. The terminating semicolon of the IF statement is considered to be the semicolon that terminates the alternative statements, the statement in the THEN clause or the statement in the ELSE clause.

In the execution of an IF statement, the stated condition is tested. If the condition is found to be true (in this case, if BALANCE = 0), the statement following THEN is executed. If the condition is found to be false (if BALANCE is not equal to zero), the statement following THEN is skipped, and the next statement in the program (the ELSE clause) is executed. When the first alternative is taken in the example shown, control is transferred back to the GET statement labeled NEW_RECORD.

Many IF statements, however, have a THEN clause that does not cause transfer of control. In those cases, when the first alternative is chosen, that statement is executed, and control then skips the next statement (the statement of the ELSE clause), and execution continues with the next sequential statement after the statement of the ELSE clause.

Often the action to be taken in one alternative may require more than one statement. But since the IF statement is designed to execute *one* statement and skip *one* statement (or skip one statement and execute the *next* statement), it is necessary to indicate that more than one statement is to be skipped or that more than one statement is to be executed before a skip is made. One way this can be accomplished is through the use of a DO group.

A DO group is a sequence of statements whose beginning is defined by a DO statement and whose end is defined by an END statement. When a DO group is used as a THEN clause in an IF statement, all of the

```
NEW_RECORD: GET FILE (INPUT) LIST (PAY_#,PAYMENT);
            GET FILE (MASTER) LIST (LOAN_#,PRINCIPAL,RATE);
            INTEREST=PRINCIPAL * RATE/12;
            BALANCE=PRINCIPAL+INTEREST-PAYMENT;
            PUT FILE (OUTPUT) LIST (LOAN_#,PRINCIPAL,INTEREST,PAYMENT,BALANCE);
            IF BALANCE=0
              THEN GO TO NEW_RECORD;
              ELSE PUT FILE (NEW_MASTER) LIST (LOAN_#,BALANCE,RATE);
            GO TO NEW_RECORD;
```

Figure 5.

statements of the DO group are executed before a skip is made or all of the statements are skipped and the ELSE clause is executed. When a DO group is used as the ELSE clause, all of the statements of that group are skipped after the THEN clause is executed.

Two other IF statements are necessary in the bank loan update program. One is a test to make certain that the current payment record and the current master record refer to the same loan. The other is a test to avoid a negative balance, as would be computed if a payment exceeded the amount due.

Figure 6 illustrates how these IF statements would appear in the program.

Before any of the information read from the INPUT file is used in computations, a check is made to avoid deducting a payment from the wrong loan. If LOAN_# is not equal to PAY_#, the two records do not refer to the same loan. Since both files are ordered in ascending sequence according to serial number, the record from which PAY_# has been read refers to a loan that is listed later in the master file, and no payment has been made during the current month for the loan referred to by the record from which LOAN_# has been read (assume that no record will appear in INPUT that does not have a corresponding record in MASTER).

The interest compounded, and the new master record is updated with the new principal. Control is returned to the GET statement labeled MASTER_FILE, another record is read from the master file, and the LOAN_# from that new record is compared with the PAY_# that had differed from the LOAN_# read from the previous record. This loop is repeated, if necessary, until LOAN_# is equal to PAY_#, in which case the entire first alternative is skipped, and the ELSE clause is executed.

The ELSE clause is another IF statement that is inserted in the program to avoid computation of a balance that is less than zero. If there has been no overpayment – if the payment is less than or equal to the total amount due – the new balance is computed. If there has been an overpayment, the THEN clause is skipped and the DO group is executed. The DO group specifies that BALANCE is to be set to zero and a refund is to be computed. The PUT statement in the DO group instructs the computer to write the serial number of the loan, the actual characters REFUND: and the amount to be refunded (the value of the variable REFUND).

When these has been an overpayment and this DO group is executed, two records concerning the loan will be written in the OUTPUT file. In this case, control passes directly from the end of the DO group to the next sequential statement. The written record stating the amount of the refund will be followed by the record giving the serial number of the loan, the principal before interest had been computed, the monthly interest charge, the payment, and the balance. In the case of a loan that has been overpaid, the balance will, of course, be zero.

Although these "unusual" conditions will arise rarely, if at all, the first condition is tested for every record that is read, and the second condition is tested for every payment record read.

One other condition could arise that would interrupt the orderly execution of the program. After the last record of any file has been read, an *end-of-file condition* exists. Any further attempt to read from that file would be unsuccessful, and execution of the program would be terminated unless the programmer had anticipated the condition. He must provide instructions to the com-

```
NEW_RECORD: GET FILE (INPUT) LIST (PAY_#,PAYMENT);
MASTER_FILE: GET FILE (MASTER) LIST (LOAN_#,PRINCIPAL,RATE);
             INTEREST=PRINCIPAL * RATE/12;
             IF LOAN_# ¬=PAY_#
               THEN DO;
                    PRINCIPAL=PRINCIPAL+INTEREST;
                    PUT FILE (NEW_MASTER) LIST (LOAN_#,PRINCIPAL,RATE);
                    GO TO MASTER_FILE;
                    END;
               ELSE IF PAYMENT<=PRINCIPAL+INTEREST
                      THEN BALANCE=PRINCIPAL+INTEREST-PAYMENT;
                      ELSE DO;
                           BALANCE=0;
                           REFUND=PAYMENT-PRINCIPAL+INTEREST;
                           PUT FILE (OUTPUT) LIST (LOAN_#, 'REFUND: ', REFUND);
                           END;
             PUT FILE (OUTPUT) LIST (LOAN_#,PRINCIPAL,INTEREST,PAYMENT,BALANCE);
             IF BALANCE=0
               THEN GO TO NEW_RECORD;
               ELSE PUT FILE (NEW_MASTER) LIST (LOAN_#, BALANCE,RATE);
             GO TO NEW_RECORD;
```

Figure 6.

puter concerning what action it should take when it encounters an instruction telling it to read from a file for which the end-of-file condition exists.

The INPUT file probably will contain fewer records than the MASTER file because payments may not be made for all the loans. If there is a record in INPUT corresponding to the last record in MASTER there would be no problem; the last loan would be updated, and termination of execution following that would cause no trouble. But if there are more records in the MASTER file to be processed after the last record in INPUT has been processed, termination of execution at that point would leave the remaining records in MASTER unprocessed.

A simple ON statement could prevent such an error:

```
ON ENDFILE (INPUT) GO TO MASTER_FILE;
```

The ON statement, like the IF statement, is a compound statement. The semicolon terminating the contained GO TO statement, in this example, also terminates the ON statement.

The ON statement instructs the computer to go to MASTER_FILE after the last record in the INPUT file has been processed. If there are other records in MASTER, they will be read and processed, and the new master file will be updated for each. After the last record from MASTER has been processed, an attempt to read again from MASTER will be unsuccessful, and the program will be terminated. This will not happen, however, until the objective of the program has been accomplished.

The position of the ENDFILE statement within the program is not critical. It need be executed only once, and it can appear any place within the sequence, so long as it is executed before the end-of-file condition arises. Once executed, it remains in force throughout the rest of the execution of the program, or until it is overridden by another ON ENDFILE statement that specifies the same file.

The end of a program, like the end of a DO group, is indicated by an END statement. The beginning is indicated by a PROCEDURE statement, and the entire block of statements, from PROCEDURE to END, is called a *procedure block*, or simply a *procedure*.

The PROCEDURE statement for this program could be:

```
UPDATE: PROCEDURE;
```

A PROCEDURE statement must have a label, and the entire procedure may be referred to by that label, or *procedure name*. A program may consist of a single procedure or of several procedures. When there is more than one procedure in a program, each procedure name is called an *entry name* and may be used somewhat similarly to the way an ordinary statement label is used in a GO TO statement.

Figure 7 shows the entire UPDATE procedure as it might be written. The pattern of indention of statements is not obligatory. But it demonstrates an advantage of PL/I in allowing a programmer to plan his own format of statement sequences. In this case, the format is designed to improve readability of the written program. So far as the PL/I compiler is concerned, the entire procedure could have been written as one continuous string of statements separated by semicolons.

An important element of a procedure is the DECLARE statement. The DECLARE statement supplies necessary information to the compiler so that storage areas can be reserved for the data represented by the names used in the procedure. A DECLARE statement describes the characteristics of the data assigned to each variable; it tells the nature of each file. A name described in a DECLARE statement is said to be *declared* and the words used to describe the characteristics of the data and files are called attributes. Although the attributes describe the characteristics of the *data*, it is the data *name* with which the declared attributes are associated. Consequently, when a data item is assigned to a name whose attributes describe characteristics that are different from the attributes of the data form, the data item will be converted so that it has the characteristics that are described by the attributes of the name to which it is assigned. For example, when a binary data item is assigned to a name that has the DECIMAL attribute, the data item is converted to the decimal representation of its value.

It is apparent, then, that a DECLARE statement specifies whether data assigned to an arithmetic data name is to be binary or decimal. It also specifies the maximum number of digits in any data item, and the number of those digits that represent a fractional portion, that is, the number of digits that should be considered to be to the right of the point. And it specifies whether the value of the data item is to be represented in fixed-point or floating-point notation. (Fixed-point and floating-point notation are described at the end of this chapter.)

For example, in the DECLARE statement shown in Figure 7, PRINCIPAL is declared to be a variable representing decimal, fixed-point numbers, none of which will contain more than eight digits, with the two rightmost digits assumed to represent a decimal fraction. RATE is declared to represent decimal, fixed-point numbers of three digits, with the decimal point assumed to be to the left of the first digit; each number will be a three-place decimal fraction such as would be used to represent a percentage rate during multiplication. The three places provide for percentage rates that might include tenths of a percent, such as 4.5 percent.

In the same DECLARE statement, LOAN_# and PAY_# are declared to be names that will represent

```
UPDATE: PROCEDURE;
        DECLARE PAY_# DECIMAL FIXED (7),
              LOAN_# DECIMAL FIXED (7),
              PRINCIPAL DECIMAL FIXED (8,2),
              BALANCE DECIMAL FIXED (8,2),
              PAYMENT DECIMAL FIXED (6,2),
              REFUND DECIMAL FIXED (6,2),
              INTEREST DECIMAL FIXED (5,2),
              RATE DECIMAL FIXED (3,3),
              MASTER FILE INPUT,
              NEW_MASTER FILE OUTPUT,
              INPUT FILE INPUT,
              OUTPUT FILE OUTPUT;
        ON ENDFILE (INPUT) GO TO MASTER FILE;
NEW_RECORD: GET FILE (INPUT) LIST (PAY_#,PAYMENT);
MASTER_FILE: GET FILE (MASTER) LIST (LOAN_#,PRINCIPAL, RATE);
        INTEREST=PRINCIPAL * RATE/12;
        IF LOAN_# ¬=PAY_#
          THEN DO;
              PRINCIPAL=PRINCIPAL+INTEREST;
              PUT FILE (NEW_MASTER) LIST (LOAN_#,PRINCIPAL,RATE);
              GO TO MASTER_FILE;
              END;
          ELSE IF PAYMENT<=PRINCIPAL+INTEREST
              THEN BALANCE=PRINCIPAL+INTEREST-PAYMENT:
              ELSE DO;
                  BALANCE=0;
                  REFUND=PAYMENT-PRINCIPAL+INTEREST;
                  PUT FILE (OUTPUT) LIST (LOAN_#, 'REFUND: ', REFUND);
                  END;
        PUT FILE (OUTPUT) LIST (LOAN_#,PRINCIPAL, INTEREST,PAYMENT,BALANCE);
        IF BALANCE=0
          THEN GO TO NEW_RECORD;
          ELSE PUT FILE (NEW_MASTER) LIST (LOAN_#,BALANCE,RATE);
        GO TO NEW_RECORD;
        END UPDATE;
```

Figure 7.

fixed-point decimal numbers of no more than seven digits, with no fractional portion.

The names MASTER and INPUT are declared to be file names that represent files to be read from; NEW_MASTER and OUTPUT, file names that represent files to be written in.

The use of the words INPUT and OUTPUT as names for files is an example of another freedom of PL/I. Since the words OUTPUT and INPUT are also used as attributes, they are keywords in the language. Yet they are not reserved; they may be used as names without causing any ambiguity.

The UPDATE procedure could be an entire program, or it could be only one procedure of a larger program that might, for example, compute all of the bank's monthly transactions.

Table 1 shows some values that might be represented by data to be read from INPUT and MASTER during execution of UPDATE.

Table 1. Examples of Data that Could be Processed by UPDATE.

From INPUT		From MASTER		
Assigned to PAY_#	Assigned to PAYMENT	Assigned to LOAN_#	Assigned to PRINCIPAL	Assigned to RATE
8212345	50.00	8212345	600.00	.050
8212347	40.00	8212346	1200.00	.055
8212348	60.30	8212347	24.00	.050
8212349	1000.00	8212348	60.00	.060
		8212349	24000.00	.055
		8212350	880.00	.060
		8212351	72.00	.050

A reader can get a better understanding of the UPDATE procedure by using these numbers to analyze it, statement by statement, doing the computations, making the tests, and following the alternative to be chosen based upon each test.

Table 2 shows the values of the data that would have been written in OUTPUT and NEWMAS files if the values in Table 1 were processed by UPDATE:

Table 2. Example of Output if Data in Figure 1 Were Processed by UPDATE.

Written in OUTPUT				
Written from LOAN_#	Written from PRINCIPAL	Written from INTEREST	Written from PAYMENT	Written from BALANCE
8212345	600.00	2.50	50.00	552.50
8212347	REFUND: 15.90			
8212347	24.00	.10	40.00	0
8212348	60.00	.30	60.30	0
8212349	24000.00	110.00	1000.00	23110.00

Written in NEW_MASTER		
Written from LOAN_#	Written from PRINCIPAL	Written from RATE
8212345	552.50	.050
8212346	1205.50	.055
8212349	23110.00	.055
8212350	884.40	.060
8212351	72.30	.050

Fixed-Point and Floating Point Notation

Fixed-point notation is the notation found in everyday use. It consists of digits and a single point that specifies the beginning of the fraction (the point usually is omitted if there is no fractional portion). For example, 3.141593 is the fixed-point notation for the value of the mathematical constant pi.

Floating-point notation is used in data processing to specify values much larger (or smaller) than would be practical in fixed-point notation. A floating-point data item in PL/I is written as a field of digits followed by the letter E, followed by an optionally signed decimal exponent. The first field may contain a point and may be preceded by a plus or minus sign.

In either decimal or binary floating point, the effect of the exponent on the value is as if the point were moved the number of digit places indicated by the exponent. Point movement is to the right when a positive exponent is specified, and to the left when a negative exponent is specified.

Examples of decimal floating-point data items:

64E46
64E-46
84.35E24
8435E22
.8435E26

The first example represents a value that could be expressed with the number 64 followed by 46 zeros. It could be written mathematically as 64×10^{46}. The second example represents a value that could be expressed by the number 64 preceded by 44 zeros and a point. In decimal fixed-point notation it would be written:

.00000000000000000000000000000000000000000064

The last three examples represent an identical value, which is 84 septillion 350 sextillion, or 8435 followed by 22 zeros.

It is important to remember that the sign of the exponent merely controls the direction in which the point is assumed to be moved; it does not indicate a positive or negative value for the data item. If no sign (or a plus sign) precedes the first field, the indicated value is not less than zero.

For most arithmetic computations using floating-point data items, the first field is converted, if necessary, to a fraction; that is, the point is assumed to be immediately to the left of the first non-zero digit. The exponent is adjusted accordingly. The fractional portion and the exponent are treated separately during computations.

Each digit position shift of the point in a decimal floating-point number multiplies or divides the number by ten. The floating-point number 40E2 represents the value 4000. The number 40E3 represents 40,000, while 40E-1 represents 4.

Binary floating-point data is comparable to decimal floating-point data except that each digit position shift of the binary point multiplies or divides the number by two.

Chapter 4: Data Types

There are three types of data commonly used in a PL/I program: arithmetic, string, and statement label.[1] *Arithmetic data* items are binary and decimal fixed-point or floating-point numbers. *String data* items are combinations of alphameric and special characters or combinations of binary digits. *Statement-label data* items are character strings used as label prefixes.

The following discussion covers each of these types of data, showing how data constants are written in a program and how data variables are defined and used.

Arithmetic Data

An item of arithmetic data is one with numeric value, that is, a number. It may be written in a program as a decimal or binary constant, or it may be the value of a variable.

Any number has certain characteristics. In the case of a constant, these characteristics are apparent. The notation in which a constant is written tells whether the data item has a decimal or binary *base* and whether it has a fixed-point or floating-point *scale,* and it states the value of the item. The number 6.5, used in an arithmetic expression, is a decimal, fixed-point number with the value of six and one half. This number is expressed by using only two digits; it is not written 06.5 or 6.50. The computer need reserve storage space for only digits. Although the decimal point is not stored in the computer, the result of an arithmetic computation is as if the point actually appeared; point alignment is maintained.

A data variable, since it is a name, does not demonstrate the characteristics of the data items it will represent. A reader would infer that the name WAGES would represent currency values. But a computer recognizes words only in the way it has been instructed to recognize them, or not at all; it assumes only what it has been instructed to assume.

Attributes must be declared for each variable so that the compiler can reserve sufficient storage space and use the proper internal notation for data items assigned to the variable. (Different kinds of data use different internal notation; for example, decimal notation and binary notation for the same arithmetic value are stored in different bit configurations.)

An arithmetic variable must have a DECIMAL or a BINARY attribute, a FIXED or a FLOAT attribute, and a *precision* attribute.

For example, the DECLARE statement for WAGES might be:

```
DECLARE WAGES DECIMAL FIXED (5,2);
```

Note that blanks must always be used to separate the keyword DECLARE, the name, and the attributes that are keywords. The precision specification must be enclosed in parentheses and must immediately follow (with or without an intervening blank) the base attribute (DECIMAL or BINARY) or the scale attribute (FIXED or FLOAT).

The precision attribute specifies the *maximum* number of digits of any data item to be assigned to the variable, and it specifies the location of the assumed decimal point. In the above example the precision attribute (5, 2) declared for WAGES specifies that no data item assigned to WAGES should contain more than five digits and that each data item is assumed to have a decimal point immediately preceding the last two digits; that is, each number will have two fractional digits (in this case, to express a value in cents).

Thus, precision implies a value *range.* For example, WAGES could represent any value from −999.99 to 999.99. If any number assigned to WAGES has fewer than three integer and two fractional digits, zeros will be inserted to express the value as a five-digit number. If a number has too many digits, the excess will be lost
– on the right if too many fractional digits are specified
– on the left if too many integer digits are specified.

If TOTAL is declared to be a decimal fixed-point variable with precision of (10, 2) any data item validly assigned to TOTAL, that is, any item representing numeric value, will be expressed as a decimal fixed-point number of ten digits, including two fractional positions. Assume X represents a decimal floating-point data item and Y represents a binary fixed-point data item. In execution of the statement TOTAL = X + Y, the data represented by X and Y would be converted to a common base and scale, the two numbers would be added, and the computed value would be converted to decimal fixed-point notation before assignment, that is, before it becomes the current value of TOTAL.

An arithmetic constant may be preceded by a plus sign or a minus sign. If unsigned, the constant is assumed to have a positive value. The sign of an arithmetic data item, unlike the point, *is* recorded in storage along with the data item. Consequently, the precision

[1] Certain other types of variables, used in program control, are also classed as separate data types. They include task, pointer, event, and area variables. For a discussion of these variables as data types, see "Data Types," Chapter 2 of *IBM Operating System/360, PL/I: Language Specifications,* Form C28-6571.

attribute (10, 2) instructs the compiler to reserve sufficient storage for ten digits *plus the sign.*

Fixed-Point Decimal Data

A fixed-point decimal data item consists of one or more decimal digits. A decimal point may be included. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit.

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

```
3.141593
−5280
455.3
.00003
234.
```

The keywords for decimal fixed-point variables are DECIMAL and FIXED. Precision is stated by means of two decimal integers, separated by a comma and enclosed in parentheses. If only one number is specified, the items are assumed to be integers.

To define PI (3.141593) in this way, the following statement could be used:

```
DECLARE PI FIXED DECIMAL (7,6);
```

This defines the identifier PI as a fixed-point decimal item of not more than seven digits, six of which are to the right of the decimal point. This declaration, of course, just authorizes use of the identifier PI in the program. No value has been assigned. This could be done later with an assignment statement, such as:

```
PI = 3.141593;
```

The value could also be assigned in the DECLARE statement, specifying the INITIAL attribute, as follows:

```
DECLARE PI FIXED DECIMAL (7,6)
  INITIAL (3.141593);
```

This not only defines the identifier PI but gives it an *initial* value of 3.141593. The value may be retained throughout the program, as is probable in this case, or it may be changed during execution by an assignment statement.

The statement of precision is critical in the definition of a variable. A programmer must know the size of the largest number that will be assigned to the variable during execution of the program, and define the variable accordingly. Consider this statement:

```
SUM = X + Y
```

If the value of X is 456.3 and the value of Y is .387, the result of this calculation would be 456.687. The following list shows how this result would be assigned to SUM under several possible precision conditions:

VARIABLE DECLARATION	ASSIGNED VALUE
SUM FIXED DECIMAL (6,3)	456.687
SUM FIXED DECIMAL (8,4)	0456.6870
SUM FIXED DECIMAL (5,3)	56.687
SUM FIXED DECIMAL (6,2)	0456.68
SUM FIXED DECIMAL (6,4)	56.6870
SUM FIXED DECIMAL (6)	000456

Note that alignment is on the decimal point, and zeros are added or digits are deleted to make the item fit the precision specification. While a programmer must guard against unwanted truncation, he can also use this characteristic as a means of dropping undesired fractional digits at the time an assignment is made.

The maximum number of digits allowed in a decimal fixed-point data item is defined for each PL/I compiler. If precision is not specified when a variable is declared, the number of digits assumed depends upon the compiler in use. For example, the maximum might be 15 with the assumed number—*default precision*—five. With default precision, the assumed location of the decimal point is immediately to the right of the rightmost digit.

Sterling Fixed-Point Data

PL/I has a facility for handling data stated in terms of British sterling currency value. Although the data may be written in a program with pounds, shillings, and pence fields, each separated by a decimal point, this data is converted and represented internally as a decimal fixed-point number representing the equivalent value in pence. A sterling data constant ends with the letter L, representing the pounds symbol, for example:

```
2.4.6L
```

This sterling constant represents two pounds, four shillings, six pence. It will be converted and stored internally as 534 (pence).

A sterling variable is declared with the same attribute keywords as fixed-point decimal data, FIXED and DECIMAL. Precision is stated as the number of digits required to represent the amount in pence.

Binary Fixed-Point Data

A binary fixed-point data item expresses an arithmetic value using binary notation. It is written as one or more binary digits with an optional binary point, followed by the letter B.

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

CONSTANT	DECIMAL EQUIVALENT
10110B	22
11111B	31
−101B	−5
111.01	7¼
−1011.111	−11⅞

A variable is declared to represent binary fixed-point data by specifying the BINARY, FIXED, and precision attributes.

Precision of a binary fixed-point variable is specified by two decimal integers, enclosed in parentheses, to represent the maximum number of binary digits and the number of digits to the right of the binary point.

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

```
DECLARE FACTOR BINARY FIXED (20,2);
```

FACTOR is declared to be a variable that can represent an arithmetic data item as large as 20 binary digits, in addition to a plus or minus sign. The decimal equivalent of that value range is from −262,144.25 through +262,143.25[1]

The maximum precision and default precision for binary fixed-point data are specified separately for each different PL/I compiler.

Decimal Floating-Point Data

A decimal floating-point data item is designated in a program by a field of decimal digits followed by the letter E followed by a decimal exponent that may be signed. The first field of digits may contain a decimal point and may be preceded by a plus or minus sign.

Examples of decimal floating-point data items:

```
15E23
15E-23
−4835E48
3141593E-6
.003141593E3
```

The last two examples represent an identical value.

The attributes that describe decimal floating-point variables are DECIMAL and FLOAT. Precision is stated with a decimal integer enclosed in parentheses. It specifies the number of digits to be maintained preceding the E. With most computers, floating-point data items are stored internally with the point immediately to the left of the first non-zero digit. If an item is assigned to a variable with a declared precision that is smaller than the field width of the assigned item, truncation will occur on the right. The least significant digit is the first that is lost.

Example of declaration of a decimal floating-point variable:

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

If the data item .814235E14 were assigned to LIGHT_YEARS, the rightmost digit of the fraction would be lost. The item still would represent a value of more than 81 trillion, but the single-digit truncation would decrease the value of the data item by 500,000,000. If the data item were .814E14, two zeros would be inserted to maintain the declared precision, and it would be as if the item assigned were .81400E14.

Binary Floating-Point Data

A binary floating-point data item, when written in a program, consists of a field of binary digits followed by the letter E, followed by a decimal integer exponent followed by the letter B. The field of binary digits may contain a binary point and, of course, a plus or minus sign. The exponent may be signed. As with decimal data, the exponent indicates displacement of the binary point.

Examples of binary floating-point data:

```
101101E5B
101.101E2B
11101E-28B
```

The attribute keywords for binary floating-point data are BINARY and FLOAT. Precision is specified by stating the number of digits preceding the E, with precision expressed as a decimal integer.

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

With most computers, binary floating-point data is represented internally with the assumed binary point immediately preceding the first 1 digit.

String Data

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

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

Character-String Data

A character-string can include any digit, letter, or special character recognized as a character by the user's computer system. Any blank included in a character string is considered an integral character of the data item and is included in the count of the length. Comments cannot be inserted within a character string.

Character-string constants, when written in a program, must be enclosed in single quotation marks. If an apostrophe or a single quotation mark is a character in a string, it must be written as two single quotation marks with no intervening blank.

Examples of character-string constants:

```
'LOGARITHM TABLE'
'PAGE 5'
'SHAKESPEARE''S ''''HAMLET'''''
'23842'
(2) 'WALLA '
```

In the last example, the parenthesized number indicates repetition of the characters and specifies the character string 'WALLA WALLA' (the blank is included as one of the characters to be repeated). The repetition factor must be an unsigned decimal integer enclosed in parentheses to indicate the number of times the characters are to appear in the actual character string.

Although a character string may be entirely numeric, such a string cannot be used efficiently in arithmetic operations with computers in which the internal representation of numeric characters is different from the internal representation of arithmetic data.

Character-string data is declared to have the CHAR-

[1] The value range is as it would be in an IBM System/360 computer. In some computers, the lowest value that can be represented by 18 binary integer digits might be −262,143.

ACTER and length attributes. Length is expressed by a decimal integer, enclosed in parentheses, which specifies the number of characters in the string. For example:

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

As with other data items declared in this way, the value of the variable, NAME, is to be assigned during execution of the program. Most data items, however, can also be given an initial value by declaring the name with the INITIAL attribute and listing the initial value. For example:

```
DECLARE NAME CHARACTER (15)
  INITIAL ('JOHN DOE');
```

Although the declared length is 15, the length of the string assigned by the INITIAL attribute contains only 8 characters. Blanks are added automatically on the right to fill out the length. The first character assigned is always left adjusted, with padding supplied on the right. In this case, the string would be stored as the characters JOHN DOE, followed by 7 blanks.

A character string is assigned from left to right. If the actual string is longer than the declared length, the excess characters are truncated on the right.

Bit-String Data

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

Bit strings are valuable for general use as logical switches that can be set to 1 or 0 as indicators that may be necessary later in the program for decision making.

Bit strings are increasingly used in information retrieval. Many "yes" or "no" answers can be recorded as a bit string in a relatively small area. For example, computer assistance in medical diagnosis makes wide use of bit strings.

Examples of bit-string constants:

```
'1'B
'111110101100001'B
(64) '0'B
```

The parenthesized number preceding the last example is a repetition factor which specifies that the following digit (or digits) is to be repeated the specified number of times. The example shown would result in a string of 64 binary zeros.

A bit-string variable is declared to have the BIT and length attributes. Length represents the number of binary digits in the string and is indicated by a decimal integer enclosed in parentheses. The letter B is not an actual part of the string and is not considered in the length specification.

Following is an example of declaration of a bit-string variable:

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

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

Label Data

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

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

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

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

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

LBL_A and LBL_B are statement-label constants because they are prefixed to statements. LBL_X is a statement-label variable. By assigning LBL_A to LBL_X, the statement GO TO LBL_X causes a transfer to the LBL_A statement. Elsewhere, the program may contain a statement assigning LBL_B to LBL_X. Then, any reference to LBL_X would be the same as a reference to LBL_B. This "value" of LBL_X is retained until another value is assigned to it.

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

```
DECLARE LBL_X LABEL;
```

The basic unit of a PL/I program is the statement. It tells the computer what to do, how to do it, and, by its relationship to other statements, when to do it. As has been shown, a statement may be considered singly, as part of a group, or as part of a block, or procedure.

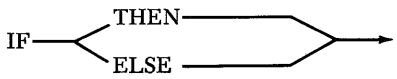
The more commonly used statements have been discussed as they appeared in the procedure UPDATE: the assignment statement for making computations; the IF, ON, and GO TO statements for program control; the GET and PUT statements for input and output; the DECLARE statement for data description; and the PROCEDURE, DO, and END statements for program structure.

This section will deal with a more detailed discussion of the IF and DO statements.

The IF Statement

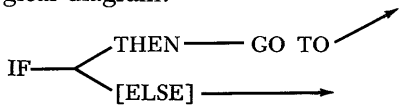
Three different IF statements were used in the UPDATE procedure in Chapter 3, as shown in Figure 8. The location of an IF statement within a program is important. The IF statement is inserted in a program at the point where an exclusive decision between alternates must be made.

A logical diagram of the IF statement could look like this:



The second IF statement fits this diagram. The two paths diverge for the execution of one statement (or group) and merge again into a single path of execution. Either the THEN clause or the ELSE clause is executed, and the other is skipped. No matter which alternative is chosen as the result of the test, execution continues with the next sequential statement that appears in the program immediately following the ELSE clause.

The other two IF statements in UPDATE fit a different logical diagram:



In this type of IF statement, one alternative (the THEN clause in both examples in UPDATE) causes a transfer of control to some other point in the program. Sequential execution does not continue. The above illustrates the kind of IF statement in which the keyword ELSE need not appear. For example, the last IF statement in UPDATE might have been written:

```
IF BALANCE = 0
   THEN GO TO NEW_RECORD;
   PUT FILE (NEW_MASTER) LIST
      (LOAN_#, PRINCIPAL, RATE);
```

If BALANCE is equal to zero, control is transferred to the READ statement labeled NEW_RECORD; execution of the THEN clause will not be immediately followed by execution of the next sequential statement. If

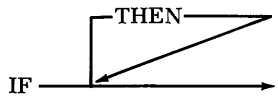
```
1  IF LOAN_# ¬= PAY_#
             THEN DO;
                PRINCIPAL = PRINCIPAL + INTEREST;
                PUT FILE (NEW_MASTER) LIST (LOAN_#, PRINCIPAL, RATE);
                GO TO MASTER_FILE;
                END;

2  ELSE IF PAYMENT <= PRINCIPAL + INTEREST
             THEN BALANCE = PRINCIPAL + INTEREST - PAYMENT;
             ELSE DO;
                BALANCE = 0;
                REFUND = PAYMENT - PRINCIPAL + INTEREST;
                PUT FILE (OUTPUT) LIST (LOAN_#, 'REFUND: ', REFUND);
                END;

   3  IF BALANCE = 0
             THEN GO TO NEW_RECORD;
             ELSE PUT FILE (NEW_MASTER) LIST (LOAN_#, BALANCE, RATE);
```

Figure 8. Examples of IF statements in UPDATE Procedure.

BALANCE is not equal to zero, the THEN clause is skipped, and the next sequential statement is executed. Exactly the same thing applies to the first IF statement in UPDATE. If LOAN_# is not equal to PAY_#, control eventually is transferred back to MASTER_FILE. If LOAN_# and PAY_# are equal, the THEN clause is skipped, and the next sequential statement is executed whether or not it is preceded by the word ELSE. In UPDATE, the second IF statement need not have been written as an ELSE clause.

Both examples 1 and 3 show the THEN clause as being the one that contains the control-transferring statement. Such a statement might appear in either clause.

One other kind of IF statement can be represented by a third diagram:



A statement representing the above kind of IF statement could be as follows:

```
          IF INDEX = 5
            THEN COUNTER = 0;
LOOP1:    GET FILE (INPUT) LIST
            (SALESNUMBER, SALES);
          COUNTER = COUNTER + 1;
              .
              .
              .
          GO TO LOOP1;
```

In the kind of IF statement illustrated above, the alternatives are "execute the THEN clause" or "do not execute the THEN clause." In either case, the next sequential statement is executed. If the expression tested is not true, control continues through the logical flow of execution. If the expression is true, the THEN clause is executed, and control returns to exactly the same point where it would have been had the expression not been true. In this kind of IF statement the word ELSE *must not* appear since the ELSE clause would be skipped whenever the THEN clause is executed.

In the example above, a count is kept of every sales record that is read for each salesman. The counter is reset to zero when the value of INDEX reaches five. The THEN clause is executed only if INDEX equals five before the first sales record is read. If INDEX does not equal five, the THEN clause is ignored.

Although the word ELSE may sometimes be omitted, the word THEN must appear in every IF statement.

IF statements can be nested, one within another, as in the following example:

```
IF A = B
   THEN IF A = C
        THEN D = E;
        ELSE F = G;
   ELSE F = A;
   GO TO LABEL1;
```

In the example, D is made equal to E only if A is equal to both B and C. If A is equal to B, but not to C, then F is made equal to G. If A is not equal to B, then F is made equal to A. If either the innermost THEN clause (D = E); or the innermost ELSE clause (F = G); is executed, control skips to the GO TO statement following the final ELSE clause.

In a series of nested IF statements, each ELSE clause is paired with an IF, starting at the innermost level. The computer makes the IF condition tests in the order that they are written. As soon as it reaches a test that is not true, the checking stops, and the matching ELSE clause is executed. Control then is transferred out of the entire series of IF statements.

In the nesting of IF statements, an associated ELSE clause may or may not appear for the outermost IF. But every *nested* IF must have an associated ELSE clause when any IF statement at a higher level requires an associated ELSE clause.

Assume that a programmer writing the above nested IF statements does not want to provide a second alternative for the innermost IF statement; if A is equal to B but not equal to C, he wants control to go to the statement labeled LABEL1. He *cannot* write the statements as follows:

```
IF A = C
   THEN IF A = C
        THEN D = E;
   ELSE F = A;
GO TO LABEL1;
```

In this case, the clause ELSE F = A would automatically be associated with the innermost IF; it would not be associated with the first IF, as had been intended. To avoid such an error, the programmer must insert a *null* ELSE, as follows:

```
IF A = C
   THEN IF A = C
        THEN D = E;
        ELSE;
   ELSE F = A;
GO TO LABEL1;
```

A null ELSE is an ELSE with a *null statement* as its clause. A null statement, as its name implies, is an empty statement; the only portion of a statement that appears is the terminating semicolon. It gives no direction to the computer. In the above example, the null statement has no effect other than to supply the necessary ELSE clause to be associated with the innermost IF. In this example, if A is equal to B but is not equal to C, the second alternative of the innermost IF is chosen. Since it is a null ELSE, control is transferred out of the entire nest to the next statement, which is GO TO LABEL1.

Suppose the programmer wanted control at this point to be transferred to the LABEL1 statement with the stipulation that D should be made equal to E before the transfer if A is equal to both B and C. He would omit

both ELSE clauses, as follows:

```
IF A = C
   THEN IF A = C
        THEN D = E;
GO TO LABEL1;
```

This series would cause immediate transfer to LABEL1 if A is not equal to B or if A is equal to B but not equal to C. Otherwise it would cause transfer to LABEL1 after D has been made equal to E.

The examples have illustrated the nesting of IF statements only to the second level. Much deeper nesting is allowed; the maximum varies from compiler to compiler.

Any IF statement, at any level, may have a DO group as either or both of its alternatives.

The DO Statement

In the procedure UPDATE, the DO statement was used only because DO groups were necessary as a THEN clause or an ELSE clause. The DO statement can also be used to define and specify control for a group of statements to be used as a loop, that is, a series of statements to be executed and re-executed one or more times before control moves on to the next statement after the group. Every DO statement must have an associated END statement to define the end of the group.

Consider the following example:

```
DO COUNTER = 1 TO 10;
statement-1
statement-2
statement-3
END;
statement-4
```

The DO and END statements specify that statements 1, 2, and 3, whatever they may be, are a DO group. The DO statement further specifies that these statements are to be executed, as a group, ten times before control is transferred to statement 4. The variable COUNTER is used to control the number of times the group is executed. When the DO statement is executed for the first time, COUNTER is assigned the value 1. Statements 1, 2, and 3 are then executed. When the END statement is reached, COUNTER is incremented by one, and control is transferred back to the beginning of the group where COUNTER is tested to see that it is no larger than 10. This looping continues until the value of COUNTER exceeds 10, when control passes on to statement 4. The above example is exactly equivalent to the following:

```
        COUNTER = 1;
LOOP:   IF COUNTER > 10
          THEN GO TO NEXT;
        statement-1
        statement-2
        statement-3
        COUNTER = COUNTER + 1;
        GO TO LOOP;
NEXT:   statement-4
```

NOTE: Since the test is made *after* COUNTER is incremented, its value at the end of the loop will be one incrementation larger than the number of times the loop is executed. In this case, the value of COUNTER will be 11 when execution of the loop is complete.

The variable COUNTER, either as used in the DO statement or as used above, would have to be declared to represent values as great as 10 (DECIMAL FIXED (2) or BINARY FIXED (4)).

A PL/I programmer may, however, use certain variables as counters without having to declare them explicitly in a DECLARE statement. Information that is necessary for storage assignment can be assumed. This facility in PL/I is called *implicit declaration.* Any identifier whose first letter is I, J, K, L, M, or N is assumed to be a variable having BINARY and FIXED attributes, unless the same identifier is declared elsewhere to have different attributes. These assumptions, or defaults, are the same for all PL/I compilers; the precision default may vary from compiler to compiler.

The DO group in the previous example might have been written:

```
DO I = 1 TO 10;
statement-1
statement-2
statement-3
END;
statement-4
```

Unless the single letter I has been declared otherwise, it is, through its appearance in the statement, implicitly declared, by default, to be a variable with BINARY and FIXED attributes and default precision.

A variable used in this way need not be a single letter; NUMBER, for example, could be used to take advantage of implicit declaration, as could I2, I3, I4, etc. Most programmers, however, find that a single letter is more convenient.

The same letter may be used as a counter in a number of different DO loops, so long as the loops are completely separate. For example, statement 4 might be a DO statement specifying the beginning of another DO loop. The letter I could be used in that second DO statement. Although its value is 11 when the first DO loop is completed, the value is reset when the second DO statement is executed.

In the preceding DO statement, the value of I is increased by one each time the DO statement is executed. Since the variable usually serves as a counter, an increment of one is assumed. However, any increment can be stipulated. For example:

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

This DO statement causes the initial value of I to be set to two. Each time the DO statement is executed thereafter, the value is increased by two. Thus, the statements of the DO group would be executed five times, and the final value of I would be 12.

This kind of control expression might be specified when the counter is also used as a variable in an expression within the DO group. For example, the following DO group could be used to compute the volume of each of a series of circular wading pools. Assume that every pool is 12 inches deep and that the diameters range from 18 inches to 10 feet, increasing by six inches from size to size.

```
DO I = 9 TO 60 BY 3;
VOL = (PI * I**2) * 12;
PUT FILE (OUTPUT) LIST (I * 2, VOL);
END;
```

The initial value assigned to I is nine, which is equal to the radius of the smallest wading pool. Each increment of three makes I equal to the radius of the next larger size. The volume is computed for each size, and the result is written, along with the diameter of the pool. Since I has a value equal to the radius, it must be doubled before it represents the value of the diameter. PL/I permits expressions such as I * 2 to be specified in a PUT statement. The expression is evaluated exactly as if it appeared in an assignment statement, and the computed value is written as indicated in the data list.

Varying values can be specified as follows:

```
DO I = 1, 8, 9, 15, 17, 25;
```

I is assigned each value in turn. After the DO group has been executed with I equal to 25, control passes to the next statement following the group. The values need not be in ascending order; they might have been written 8, 1, 15, 17, 9, 25.

The values of variables may be assigned successively to the counter in a DO statement:

```
DO X = A, B, C, D, E, F, G, H;
DIST = X*.588E13;
PUT FILE (OUTPUT) LIST (X, DIST);
END;
```

Assume that the variables A through H represent the light-year distance to certain stars expressed as decimal floating-point numbers. The first time the DO statement is executed, the value of A is assigned to X, and the first statement of the group, the assignment statement, is executed. The value of X is multiplied by .588E13, which is an approximation of the distance, in miles, that light travels in one year. The computed value of the expression, which is a floating-point number representing the distance to the star in miles, is assigned to DIST. Then the distance of A in light-years and the distance in miles are written as floating-point numbers. Control returns to the DO statement, and the value of B is assigned to X for the second execution of the group. Repetition continues until the distance in miles has been computed and written for the star represented by H.

Note that the letter X is used for the counter in the DO statement. As noted before, an identifier whose initial letter is I, J, K, L, M, or N can be implicitly declared to represent binary fixed-point data items. Likewise, any identifier beginning with a letter *other than* I through N can be implicitly declared to be a variable representing decimal floating-point numbers. Thus, both X and DIST can be implicitly declared merely through their appearance in this DO group (any of the variables A through H might also have been implicitly declared, although they obviously have appeared elsewhere in the program when their value was assigned).

Another method of loop control causes looping to continue as long as a certain condition exists, as follows:

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

Here, the values of A and B are compared each time control reaches the DO statement. The computer continues executing the statements in the DO group until the value of A becomes equal to or greater than the value of B. Obviously some computation within the DO group must alter the value of A or B, or the loop will be unending. The WHILE clause—any expression following WHILE—must be enclosed in parentheses.

Now, combining the features of both these examples, there is this form of the DO statement:

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

This causes repeated execution of the group either until the tenth execution is completed or until A no longer is less than B. As soon as either condition is satisfied, execution ceases, no matter what the status of the other.

If the programmer wants to continue execution until each of two or more conditions is met, he can write:

```
DO I = 1 TO 10, 11 BY 0 WHILE (A < B);
```

The group is executed ten times. Then the new condition takes control, and execution continues until A no longer is less than B. Use of the comma to separate the control expressions has the effect of setting up successive DO statements for the same DO group (note that if the TO clause is omitted but the BY clause is included, execution could continue indefinitely).

The following statement illustrates another variation:

```
DO I = 1 TO 4, 6 TO 10;
```

The statements of the group are executed nine times. When the group is executed, the value of I will be, successively, 1, 2, 3, 4, 6, 7, 8, 9, 10. The group is not executed when I has the value 5. It is important to note that only one variable can be used as a *counter* in a single DO statement. More than one variable may appear (DO I = K TO M might be written), but only one variable may appear to the left of an equal sign. Thus, the following statement would be in error:

```
DO I = 1 TO 4, J = 6 TO 10;
```

The above would have to be written as separate DO statements, each having its own END statement.

DO groups, like IF statements, may be nested. Consider this example:

```
DO I = 1 TO 10:
statement-1
statement-2
statement-3
            DO J = 1 TO 10;
            statement-1a
            statement-2a
            statement-3a
            END;
statement-4
statement-5
statement-6
END;
```

The statements of the outer DO group—the outer DO-END and statements 1 through 6—would be executed ten times. The statements of the inner DO group—the inner DO-END and statements 1a through 3a—would be executed 100 times, ten times for each execution of the outer DO group. When the first DO statement is executed the first time, I is assigned the value 1. Then statements 1 through 3 are executed. When control reaches the second DO statement, J is assigned the value 1, and the inner loop is executed until the value of J exceeds 10. Control then passes on to statements 4, 5, and 6. When the final END statement is reached, control returns to the first DO statement. The counter I is incremented to 2, and execution proceeds through statements 1 through 3. When the second DO statement is reached for the second time, J is reset to 1, and the inner DO group again is executed ten times before control passes to statement 4 for its second execution. The process is repeated until the outer DO group has been executed ten times. The inner DO group goes through its entire looping process immediately following each execution of statement 3.

When DO groups are nested, a *different* variable must be used for each counter. Had the same counter been used for both groups in the above example, it would reach the value of eleven when the first looping of the nested group was completed. Statements 4, 5, and 6 would be executed for the *first* time, and control would go to the next statement after the END statement of the first DO group. Although the outer loop would be executed only once, the counter would indicate that it had been executed ten times.

The example shows nesting only to the second level. Much deeper nesting is allowed; the maximum varies from compiler to compiler. In general, the maximum will be the same as that for nested IF statements.

Whatever the number of nested groups, each *contained* group will be executed to *completion* for every *single* execution of its *containing* group, just as the second group, in the above example, is executed to completion for every single execution of the outer group.

A DO statement can have a label prefix, and control can be passed to it through the use of a GO TO statement. Control can be passed to a statement within a DO group only if the group is headed by a simple DO statement, that is, a DO without any repetitive specification. Control can, however, be transferred out of a DO loop; for example, a GO TO statement might be a THEN or ELSE clause of an IF statement within the loop.

This discussion has covered only a few of the many ways DO groups can be used. Virtually the only limitation is the programmer's ingenuity.

Chapter 6: Procedures

A program may consist of a single procedure or of several procedures. During execution of the program, control can go from one procedure to another and can return to a previously executed or partly executed procedure.

A procedure is headed by a PROCEDURE statement and ended by an END statement, as follows (the dots represent the statements in the procedure):

```
UPDATE: PROCEDURE;
          .
          .
          .
        END;
```

Each procedure must have a name, that is, each PROCEDURE statement must be labeled. A procedure name denotes an entry point through which control can be transferred to the procedure.

The division of a program into several procedures is a feature of PL/I that provides a special convenience to programmers. The procedures can be written separately, even compiled separately, and executed as a single program. A long program can be divided into logical blocks; special procedures can be written for special purposes. A discussion in the next chapter shows how this feature also provides great economy in the use of internal storage space.

Control does not pass automatically from one procedure to the next. Each procedure, except the first, must be invoked, or called separately from some other procedure. This usually occurs with the execution of a CALL statement, for example:

```
CALL UPDATE;
```

Execution of this statement in another procedure would transfer control to the entry point of the procedure named UPDATE.

The first procedure of a program must have the OPTIONS (MAIN) attribute specified for it in its PROCEDURE statement. At execution time, this procedure is called automatically to begin execution of the program.

The different procedures in a program may be entirely separate from one another, or some may be nested within other procedures. Consider the following two examples:

Example 1:

```
FIRST:   PROCEDURE OPTIONS (MAIN);
         statement-1
         statement-2
         statement-3
         statement-4
         statement-5
         statement-6
         END;
UPDATE:  PROCEDURE;
         statement-a
         statement-b
         statement-c
         END;
```

The two procedures shown are separate from one another; they are *external* procedures. All of the text of a procedure, except its entry name, is said to be contained in that procedure.

Example 2:

```
FIRST:  PROCEDURE
        statement-1
        statement-2
        statement-3
        UPDATE:  PROCEDURE;
                 statement-a
                 statement-b
                 statement-c
                 END;
        statement-4
        CALL UPDATE;
        statement-6
        END;
```

In this example, UPDATE is nested within—or contained in—FIRST. FIRST is an external procedure; UPDATE is an *internal* procedure.

Execution starts with the FIRST: PROCEDURE statement. Statements 1, 2, and 3 are executed. When control reaches the UPDATE: PROCEDURE statement, that statement is ignored, and execution continues with statement 4. Upon execution of the fifth statement, CALL UPDATE, control is transferred to UPDATE. Statements a, b, and c are executed. When the END statement in UPDATE is executed, control is transferred back to the procedure, FIRST, to execute the statement immediately following the CALL UPDATE statement; that is, control is transferred to statement 6.

When UPDATE is called, it is the *invoked procedure* and FIRST is the *invoking procedure;* the CALL UPDATE statement is the *point of invocation.*

Any procedure, external or internal, can invoke another external procedure, but it cannot directly invoke an internal procedure that is contained in some other procedure. Assume that FIRST is a control procedure as follows:

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

```
UPDATE: PROCEDURE;
            statement-a
            statement-b
            statement-c
            END;
CALL D;
CALL UPDATE;
CALL E;
END;
```

The CALL A statement invokes the procedure named A, and control is transferred to A. When it returns to FIRST, after completion of A, the statement CALL B is immediately executed, and control is transferred to the procedure B, later to return to the CALL C statement in FIRST. Execution of FIRST continues in this way.

Any of the procedures invoked by FIRST might invoke other procedures. For example, a statement in procedure A might invoke a procedure Z. But control eventually returns to the statement in the invoking procedure that immediately follows the point of invocation (from Z to A and from A to FIRST).

Since UPDATE is an internal procedure contained in FIRST, FIRST is the only external procedure that can invoke it. If another procedure, say procedure F, were contained in UPDATE, only UPDATE could invoke it. No statement in FIRST could call F.

The following example demonstrates how an internal procedure can be made available to other external procedures:

```
PROC_1: PROCEDURE;
        statement
            .
            .
            .
BETA:   ENTRY;
        CALL PROC_A;
        PROC_A: PROCEDURE;
                    statement
                        .
                        .
                        .
                    END;
        END;
```

In this example, PROC_1 has two entry names, PROC_1 and BETA. The statement labeled BETA is an ENTRY *statement*. An ENTRY statement specifies that its label can be used in a CALL statement to invoke the procedure at the point where the ENTRY statement appears. An ENTRY statement declares its label to have the ENTRY attribute in the same way that a PROCEDURE statement declares its label to have the ENTRY attribute. PROC_1 can be invoked at its beginning with a CALL PROC_1 statement, or at the ENTRY statement with a CALL BETA statement (an ENTRY statement can be given any valid label to be used in an invoking call).

When PROC_1 is invoked at BETA, the first statement executed invokes PROC_A; the invoking procedure has, in effect, called an internal procedure outside its normal scope. When execution of PROC_A is completed, control returns to the statement in PROC_1 that follows the point of invocation, in the preceding example, the END statement of PROC_1. Consequently, control returns to the procedure that invoked PROC_1 at the entry point, BETA.

When PROC_1 is invoked at its primary entry point, at PROC_1, the statements are executed in the order indicated by the programmer. When control encounters the ENTRY statement, it skips to the next statement, in this case, the CALL PROC_A statement.

More than one procedure may be contained in a single procedure, either as separate internal procedures or as nested internal procedures:

```
PROC_1: PROCEDURE;
        statement-1
        statement-2
        CALL PROC_A;
        PROC_A: PROCEDURE;
                    statement-1a
                    statement-2a
                    CALL PROC_B;
                    PROC_B: PROCEDURE;
                                statement-1b
                                statement-2b
                                statement-3b
                                END;
                    statement-4a
                    statement-5a
                    statement-6a
                    END;
        statement-4
        statement-5
        CALL PROC_C;
        PROC_C: PROCEDURE;
                    statement-1c
                    statement-2c
                    statement-3c
                    END;
        statement-7
        statement-8
        statement-9
END_1:  END;
```

PROC_A, PROC_B, and PROC_C are all contained in PROC_1. PROC_B also is contained in PROC_A. PROC_1 can invoke either PROC_A or PROC_C; either PROC_A or PROC_C might invoke one another; but only PROC_A can invoke PROC_B.

A contained procedure must be wholly within its containing procedure; all of the statements of a contained procedure must appear between the PROCEDURE and END statements of the procedure in which it is contained.

A GO TO statement may be used in an internal procedure to transfer control from that procedure to any labeled statement (except entry points) in any containing procedure. A GO TO statement, generally, cannot be used to transfer control between separate procedures or from an outer procedure to a statement in any of its contained procedures.

In the previous example, a GO TO statement in PROC_B could transfer control to a statement in

PROC_A or PROC_1 but not in PROC_C, because PROC_B is not contained in PROC_C. A GO TO statement could also be used to transfer control from PROC_A or from PROC_C to a statement in PROC_1. A GO TO statement cannot be used to transfer control to a statement in PROC_B, or PROC_C; PROC_1 can use a GO TO statement only to transfer control from one to another of its nine internal statements or to the labeled END statement.

Control returns to an invoking procedure when the END statement of an invoked procedure is reached. Often, there are reasons why a programmer wants control to return *before* the END statement is reached. The procedure UPDATE, explained in Chapter 3, illustrates such a situation.

The execution of UPDATE will end after the end-of-file condition arises for MASTER file. With UPDATE used as one of several procedures in a program, another ON statement would be required. The END statement of the UPDATE procedure could be given a label, say ENDUP, and the following statement could be inserted somewhere in the program:

```
ON ENDFILE (MASTER) GO TO ENDUP;
```

After the last record in MASTER file is read and processed, control is transferred to the END statement and is returned to the invoking procedure.

Another method of returning control from an invoked procedure to its invoking procedure is handled through the use of the RETURN statement. When a RETURN statement is executed, the result is the same as if the procedure had been completely executed; control returns immediately to the first statement following the point of invocation. A RETURN statement could not be used in the preceding ON ENDFILE statement because a RETURN statement cannot be the contained statement of an ON compound statement.

A RETURN statement might appear in a procedure as follows:

```
IF CNTR = 100
  THEN RETURN;
GO TO LOOP;
```

Presumably, some operation in the procedure is to be done 100 times and a counter is used, rather than a DO loop, to control the number of repetitions.

In general, each procedure should be written in such a way that it is as independent as possible. It usually is preferable to return control to an invoking procedure with the use of the END statement or a RETURN statement, even if a GO TO statement might be used. It also is preferable, so far as is possible, to keep all references to a single variable within the same procedure (or procedures contained in it). The reasons for this will be made apparent in the following discussions of "Program Execution" and "Recognition of Names."

Program Execution

Any name that represents a data item actually represents the location in storage where that data item is recorded. The compiler analyzes the attributes of a variable to determine the amount of storage area that is needed and *when* it will have to be available.

The fact that certain variables are used in one part of a program and not used in others makes it possible to set aside—or *allocate*—the same physical storage location, at different times, to different variables.

If during execution of a single procedure, but nowhere else in the program, a 100-character field is required for the variable TABLE, the space need not actually be allocated until execution of that procedure begins. If TABLE is not referred to again, there is no need to keep the space reserved after execution of the procedure is completed. The storage area can be used for other purposes.

Such dynamic control of storage is called *automatic* allocation. Variables declared in a procedure are assumed to have the AUTOMATIC storage class attribute; they are allocated when the procedure is invoked and are freed when it is completed.

Some variables may be declared to have the STATIC storage class attribute; they are allocated when the *program* execution begins, and they remain allocated throughout execution of all the procedures in the program.

A third storage class is the controlled storage class. Variables declared to have the CONTROLLED attribute are allocated upon execution of an ALLOCATE statement referring to the variable name. The storage allocated for a controlled variable is released upon execution of a FREE statement referring to the variable. Consider the following example:

```
DECLARE A (50, 50) CHARACTER (15)
  CONTROLLED;
      .
      .
      .
ALLOCATE A;
      .
      .
      .
FREE A;
      .
      .
      .
```

In the DECLARE statement, A is declared to be a two-dimensional array of character strings having the CONTROLLED attribute. Storage is not assigned for A until execution of the ALLOCATE A statement. Upon execution of the FREE A statement, storage no longer is allocated for A; the storage can be used for other purposes.

As long as a procedure is active, its automatic variables remain allocated. They are not released until the

procedure is terminated. The first, or MAIN, procedure remains active throughout the entire program.

A procedure is activated whenever it is invoked through any of its entry points—the procedure name or another entry name. A procedure can be terminated in any of several ways: when its END statement is executed, when a RETURN statement in the procedure is executed, when a STOP statement is executed in *any* procedure, or when a GO TO statement transfers control to an outer procedure.

Figure 9 shows a portion of a program that consists of at least three external procedures, FIRST, THIRD, and SIXTH (SIXTH is not illustrated). One procedure, SECOND, is contained in FIRST; and two procedures, FOURTH and FIFTH, are contained in THIRD, with FIFTH also contained in FOURTH. The numbers at the left are merely for reference; they are not part of the program. FIRST is activated automatically. When statement 1 is executed, all automatic variables declared in FIRST are allocated. Although SECOND is contained in FIRST, it is not activated until it actually is invoked; only its *entry name* is made available when FIRST is activated.

Statement 2 invokes SECOND, and allocations for it are made. Then its statements are executed. When the END statement in SECOND (statement 7) is executed, SECOND is terminated, its automatic allocations are released, and control returns to FIRST. *SECOND is terminated through execution of its END statement.* Storage locations that were automatically allocated for SECOND are now available for use by other procedures.

Statement 8 invokes THIRD. FIRST remains active (as it will throughout execution of the entire program), and THIRD is activated. Again, the internal procedures are not activated, except that the entry name of FOURTH is made available to THIRD.

Figure 9 shows why a GO TO statement cannot be used to transfer control to a labeled statement in an inner procedure. Since the inner procedures are not active, statement labels within them are not available. Even the entry name FIFTH is not available to THIRD. When an inner procedure *is* active, however, its *outer* procedures also are active; consequently, control can go to a labeled statement in an outer procedure.

Statement 13 causes the activation of FOURTH (THIRD remains active, because it has not been terminated). If the THEN clause of statement 19 is executed, control returns to statement 14; FOURTH is terminated before it activates FIFTH. In such a case, *FOURTH is terminated through execution of a RETURN statement.*

If the comparison in statement 19 reveals that M is not equal to N, FIFTH is activated. At this point, all but one of the procedures illustrated are active. Only

```
 1   FIRST:  PROCEDURE OPTIONS (MAIN);
 2           CALL SECOND;
 3           SECOND:  PROCEDURE;
 4                    statement
 5                    statement
 6                    statement
 7                    END;
 8           CALL THIRD:
 9           CALL SIXTH;
10           END;
11   THIRD:  PROCEDURE;
12           statement
13           CALL FOURTH;
14           statement
15           statement
16           FOURTH:  PROCEDURE;
17                    statement
18                    statement
19                    IF M = N THEN RETURN;
20                    CALL FIFTH;
21                    FIFTH:    PROCEDURE;
22                              statement
23                              IF M = N THEN GO TO A;
24                              statement
25                              END;
26                      IF M = N THEN STOP;
27                      END;
28          A:  statement
29           END;
```

Figure 9. Termination of blocks.

SECOND has been terminated (SIXTH, of course, and any procedures it might call have not yet been activated).

If the THEN clause in statement 23 is executed, both FIFTH and FOURTH are terminated, and control is transferred to statement 28, the statement in THIRD that is labeled A. *FOURTH and FIFTH are terminated when the GO TO statement transfers control to an outer procedure.*

However, if M still is not equal to N, statements 24 and 25 are executed. FIFTH is terminated normally, and control returns to FOURTH.

If the STOP statement – the THEN clause of statement 26–is executed, program execution ceases. *All of the procedures are terminated when the STOP statement is executed.*

Otherwise, FOURTH is terminated normally, execution returns to statement 14, and the remaining statements of THIRD are executed. When THIRD is terminated by execution of its END statement, FIRST is the only procedure that remains active. It immediately activates SIXTH. FIRST remains active until control returns to execute its END statement. Execution of the END statement of the MAIN procedure has the same effect as execution of a STOP statement; program execution ceases.

Before control returns from SIXTH, however, other procedures could be activated; SIXTH might invoke internal procedures contained in it, or it might invoke other external procedures. *Even THIRD might be reactivated.*

It is important to note that if THIRD were to be reactivated, its variables would not represent the same values that they represented before the previous termination. When allocation for a variable is released, its value no longer can be determined.

Chapter 7: Recognition of Names

An inference might be drawn from the preceding discussion that no name can be referred to in a procedure unless the name has been explicitly declared within that procedure. The inference would be only partly true.

Some names can be referred to in any procedure, regardless of the procedure in which they appear in a DECLARE statement. These names are *known* throughout the program; their *scope* is the entire program. Other names have a more limited scope; they are known only in parts of the program. A name is *known* when a reference to it can be valid. The *scope* of a name is the range of the program throughout which the name is known.

The way a name is declared, the attributes declared for it, and the procedure in which it is declared can all have an effect upon the scope of the name.

In this discussion, the word "identifier" refers merely to the character or characters that constitute a name; the word "name" refers to an identifier that has been declared to have specific attributes.

The declaration of a name takes place at the time the source program is compiled. The compiler must have full information about every name that appears in a procedure before it can translate the statements into executable machine-language code. In some cases, all or part of the information may be based on assumptions.

A PL/I compiler is written in such a way that any identifier—except language keywords appearing in their proper context—is assumed to be a variable representing arithmetic data that is to be recorded either in decimal floating-point form or in binary fixed-point form, depending upon which letter of the extended alphabet appears as its initial (or only) character. Thus, any identifier could be *implicitly* declared. An implicitly declared identifier is given BINARY and FIXED attributes if its initial character is I, J, K, L, M, or N; otherwise, it is given DECIMAL and FLOAT attributes. An implicitly declared variable is given *default precision*, that is, the precision that is assumed by the particular compiler in use.

These primary assumptions, however, can be overridden, either through the context in which the identifier is used or through the appearance of the identifier in a DECLARE statement in which at least one attribute is specified for it.

An identifier is *contextually* declared to be a statement label if it appears with a statement as a label prefix; it is further contextually declared to have the ENTRY attribute if the statement is a PROCEDURE or ENTRY statement. An identifier that is referred to in a CALL statement also is assumed to be an entry name. (No assumptions are made about an identifier that appears in a GO TO statement; thus, a label variable must be given the LABEL attribute in a DECLARE statement.) An identifier that appears in an input or output statement in a context where only a file name can appear is assumed to have the FILE attribute.

An identifier is *explicitly* declared to have the attributes specified for it in a DECLARE statement.

In an implicit or contextual declaration, *all* of the information needed to classify a name is assumed. If only a part of the information is explicitly declared for an identifier, other attributes are assumed, by default, based upon the attributes that are stated. The name still is said to be declared explicitly, even though some of its attributes are not stated explicitly.

The external procedure shown in Figure 10 illustrates each kind of declaration.

In statement 2, the variables A and B and the file names INFILE and OUTFILE are explicitly declared.

A and B are explicitly declared to have the FIXED attribute and a precision attribute (6). Since they are not completely declared, the DECIMAL attribute is given to them, by default (default attributes for partly

```
1    EXAMPLE:    PROCEDURE;
2                DECLARE (A,B) FIXED (6), (INFILE INPUT, OUTFILE OUTPUT) FILE;
3                ON ENDFILE (INFILE) GO TO PRINT;
4                I = 0;
5    LOOP:       GET FILE (INFILE) LIST (A,B);
6                C = A * B;
7                I = I + 1;
8                PUT FILE (OUTFILE) LIST (C);
9                GO TO LOOP;
10   PRINT:      PUT FILE (OUTFILE) LIST (I);
11               CALL NEXT;
12               END;
```

Figure 10.

declared arithmetic data names are DECIMAL, FLOAT, and the default precision specified for the particular compiler).

Both INFILE and OUTFILE are explicitly declared to be file names, INFILE with the INPUT attribute and OUTFILE with the OUTPUT attribute. Since they are explicitly declared, there is no contextual declaration as a result of their appearance in the GET and PUT statements.

Attribute specifications can be *factored* in a DECLARE statement; that is, a common attribute can be specified for more than one name by enclosing the names in parentheses and specifying the common attribute following the closing parenthesis. The FIXED attribute specification is factored for A and B, and the FILE attribute specification is factored for both file names.

In any DECLARE statement, one or more blanks must separate the attribute specification for a name, and each name must be separated from the preceding attribute list (or name) by a comma.

Four names are contextually declared in the above procedure. The procedure name, EXAMPLE, is declared to have the ENTRY attribute by its appearance in the PROCEDURE statement. LOOP and PRINT are declared to be labels by their appearances as labels in statements 5 and 10. And NEXT is declared to have the ENTRY attribute by the context in which it appears in statement 11. The assumption is made that NEXT appears somewhere else in the program as the label of a PROCEDURE statement. It is given the ENTRY attribute during compilation of this procedure and entered in the table of ENTRY names so linkage can be established when the NEXT procedure is compiled.

Two identifiers, C and I, are implicitly declared in the procedure EXAMPLE. The variable I, used to count the number of records processed, is assumed to have BINARY, FIXED, and default precision attributes; C is assumed to have DECIMAL, FLOAT, and default precision attributes. (Note that the values of A and B will be fixed-point data items. When the expression in statement 6 is evaluated, the value will be converted to floating-point scale before it is assigned to C.)

The names I and C are implicitly declared in the procedure EXAMPLE but not in any particular statement in the procedure. Rather, they are implicitly declared because they do *not* appear in any statement that could supply information about them other than the letters of their names.

All of the variable names declared in the above procedure are assumed to have the AUTOMATIC storage class attribute; they will be allocated only while EXAMPLE is active. They are assumed to have the INTERNAL scope attribute; that is, they will be known only within EXAMPLE. The statement labels, LOOP and PRINT, also have the INTERNAL attribute. All of these names are *internal names.*

EXAMPLE and NEXT, since they are entry names of external procedures, and the file names INFILE and OUTFILE, are contextually declared to have the EXTERNAL scope attribute. They are external names, and they can be used throughout the entire program. File names can be given the INTERNAL scope attribute by explicit declaration only.

Within a single procedure that has no contained blocks, an identifier cannot be declared more than one time. The same identifier cannot appear in more than one DECLARE statement within that procedure, even if the explicitly declared attributes are not conflicting.

The same identifier can, however, be declared more than once in *separate* blocks. For example, A is declared in EXAMPLE. It could also have been declared in UPDATE. But a value assigned to A in EXAMPLE could not be retrieved in UPDATE. Although the identifiers would be identical, the *names* would be separate names. The two declarations for A might give A the same attributes or completely different attributes.

However, a name given the EXTERNAL attribute, either explicitly or by default, is available to other procedures that also declare the same identifier EXTERNAL. For example, A might have been declared:

```
DECLARE A FIXED(6) EXTERNAL;
```

This DECLARE statement makes A available to any procedure in the program that also declares A to have the EXTERNAL attribute. Any other procedure in which a reference is made to this particular name must declare A with all of the same attributes, *including the EXTERNAL attribute.* The identifier A can still be declared internally in some other procedure. In that case, it is considered a separate declaration—or *redeclaration*—of the identifier, and the name has the INTERNAL scope attribute; it is a different name, representing a different storage area.

Some external names need not have the EXTERNAL attribute specified for them in each procedure. A name that can be contextually declared EXTERNAL in a procedure—an entry name or a file name—need not be explicitly declared EXTERNAL.

The nesting of procedures has a profound effect upon the scope of names appearing in the procedures. In general, a name is known throughout the procedure in which it is declared and throughout most procedures that are contained in the procedure in which the name is declared. The exception is any contained procedure in which the same identifier is redeclared. Most names declared within an internal block are *not* known in any outer block.

Consider the example in Figure 11; it represents the first external procedure of a program with a single internal procedure contained in it. Each name appear-

```
1    FIRST:    PROCEDURE OPTIONS (MAIN);
2              DECLARE (M, N) FLOAT(6) EXTERNAL,
                       ALPHA FIXED (6, 2), INFILE
                       FILE INPUT;
3              CALL SECOND;
4              SECOND:   PROCEDURE;
5                        DECLARE M FIXED (4, 2), TITLE
                           CHARACTER (8);
                           .
                           .
                           .
6                        END;
7              CALL THIRD;
8              CALL SIXTH;
                    .
                    .
                    .
9              END;
```

Figure 11.

ing in the example has a distinctive scope within the procedure or within the program.

When execution begins, all STATIC names in the program are allocated, regardless of what procedure they are declared in. Names declared AUTOMATIC (explicitly or by default) in statements *internal to* FIRST are allocated when FIRST is invoked.

(The phrase *internal to* refers to all of the text contained in a block that is not contained in another block nested within it. Note the difference between "internal" and "internal to." In Figure 11, SECOND is an *internal* procedure contained in FIRST, but only its entry name (the name SECOND) is internal to FIRST. The nesting of DO groups or IF statements is not a consideration. Neither a DO or IF statement nor any nesting of them has any effect on the scope of names.)

Since the procedure SECOND is not internal to FIRST, the contained procedure is not activated when FIRST is invoked. The *name* SECOND, however, as declared in statement 4, is internal to FIRST; the entry name of an internal procedure is considered to be contained in (and internal to) the immediately containing procedure.

Table 3 gives the scope of each name that appears in Figure 11.

The declaration of M in statement 5 is a redeclaration of the identifier. The M declared in statement 5 is a different variable from the M declared in statement 2. It is not known in procedure FIRST because it is an internal name, but not internal to FIRST. For the same reason, TITLE is not known in FIRST.

THIRD and SIXTH are declared contextually to have the ENTRY and EXTERNAL attributes in statements 7 and 8 because they appear in CALL statements and are not otherwise declared in the block. SECOND is *not* contextually declared in statement 3, because it is contextually declared in another statement (number 4) that gives more specific information about it, that is, that it is an *internal* entry name.

Table 3. Scope of Names Declared in Procedure FIRST (see Figure 11).

STATEMENT REFERENCE NUMBER	NAME	USE	SCOPE
1	FIRST	external entry name	entire program[1]
2	M	external floating-point variable	FIRST, and any other procedure in which it is declared EXTERNAL; not SECOND because identifier is redeclared in SECOND
2	N	external floating-point variable	all of FIRST and SECOND, and any other procedure in which it is declared EXTERNAL
2	ALPHA	internal fixed-point variable	all of FIRST and SECOND
2	INFILE	file name	entire program[2]
4	SECOND	internal entry name	all of FIRST
5	M	internal fixed-point variable	SECOND
5	TITLE	internal character string	SECOND
7	THIRD	external entry name	entire program
8	SIXTH	external entry name	entire program

[1] Although an entry name is known throughout the declaring block (and most contained blocks), it cannot, in most cases, be referred to within itself. The exception is within a procedure having the RECURSIVE attribute, which allows the block to be reinvoked while it is still active.
[2] File names are always assumed to have the EXTERNAL attribute unless explicitly declared otherwise.

```
1   THIRD:          PROCEDURE;
2                   DECLARE M FIXED (4, 2), SWITCH BIT (1);
3                   CALL FOURTH;
4                   FOURTH:     PROCEDURE;
5                               DECLARE ALPHA CHARACTER (8),
                                (M, N) EXTERNAL;
6                               IF M = N THEN RETURN;
7                               CALL FIFTH;
8                               FIFTH:      PROCEDURE;
9                                           DECLARE SWITCH BIT (1);
10                                          GET FILE (INFILE) LIST (I);
11                                          M = N + I;
12                                          IF M = N THEN GO TO A;
13                                          END;
14                              IF M = N THEN STOP;
15                              END;
16              A: statement
17                  END;
```

Figure 12.

Figure 12 shows some statements of another external procedure of the same program.

Some of the names that appeared in FIRST are also known in THIRD. Table 4 gives the scope of each name that appears in the example.

One of the important reasons for dividing a program into separate procedures and for nesting some procedures within others is for the purpose of limiting or extending the scope of some names. Such scope limitations can also be made through the use of the BEGIN statement.

A BEGIN statement, like a PROCEDURE statement, heads a block of statements whose end is indicated by an associated END statement. A block of statements that is delimited by a BEGIN statement and an END statement is called a *begin block.*

A begin block resembles a procedure block in that it can delimit the scope of names. Names can be declared in a begin block, either contextually or explicitly. A begin block, however, cannot be an external block; it must be contained in a procedure. It may contain other blocks, either begin blocks or procedures, which can be invoked only from within the containing begin block.

A major difference between a procedure and a begin block is the way in which they are activated. As noted previously, a procedure is activated remotely, for example, by a CALL statement; a begin block is activated through normal sequential execution. Control automatically passes into a begin block after the statement preceding it has been executed; control passes through the END statement of a begin block to the next sequential statement. A begin block can be either the THEN clause or the ELSE clause of an IF statement, in which case it functions in much the same way as does a DO group.

A BEGIN statement need not be labeled, but if it is given a label, control can be passed to the begin block by a reference to that label in a GO TO statement. The label of a BEGIN statement cannot be referred to in a

Table 4 Scope of Names Declared in Procedure THIRD (see Figure 12).

STATEMENT REFERENCE NUMBER	NAME	USE	SCOPE
1	THIRD	external entry name	entire program
2	M	internal variable	THIRD not including FOURTH and FIFTH[1]
2	SWITCH	internal variable	THIRD and FOURTH[2]
4	FOURTH	internal entry name	THIRD, FOURTH, and FIFTH
5	ALPHA	internal variable	FOURTH and FIFTH
5	M	external variable	FIRST, FOURTH, FIFTH, and any other procedure in which M is declared EXTERNAL
5	N	external variable	Same as M in 5, plus SECOND
8	FIFTH	internal entry name	FOURTH, FIFTH
9	SWITCH	internal variable	FIFTH[2]
10	INFILE	file name	entire program[3]
10	I	internal variable	all of THIRD[4]
16	A	statement label	all of THIRD

[1] Although M, as explicitly declared in THIRD, has exactly the same attributes as the M that is explicitly declared in SECOND, the two are completely different names.

[2] The two declarations of SWITCH are different; there is no actual connection between the two names.

[3] INFILE is contextually declared here. It is the *same* INFILE that is explicitly declared in FIRST. All file names are assumed to have the EXTERNAL attribute unless specified otherwise.

[4] An implicitly declared name is assumed to be declared in the *external* procedure regardless of where the reference is made to the name within the procedure.

CALL statement.

The primary use of the BEGIN statement and a begin block is to limit the scope of a name, or names, through a sequentially executed block of statements.

Summary

In a program that consists of a single procedure, the scope of any name is the entire program. The ability to limit or extend the scope of a name is a valuable programming tool; so is the ability to control the allocation of storage. Consequently, most programs will consist of more than one procedure.

It is valuable to note the relationships between the scope of a name and the activation and termination of blocks in which the name is used.

An external name can be known throughout all the procedures of a program. All external variables are assumed, by default, to have the STATIC storage class attribute; they remain allocated throughout execution of the entire program.

An internal name is known only within a single external procedure. All internal variables are assumed, by default, to have the AUTOMATIC storage class attribute; they remain allocated only while the block to which their declaration is internal—*the declaring block* —remains active.

The scope of a name includes all blocks contained in the declaring block except for any block (and its containing blocks) in which the identifier is redeclared.

The scope of an external name is through all blocks in which the identifier is declared, either explicitly or contextually, to have the EXTERNAL attribute.

The scope of an implicitly declared variable is through the entire external procedure. The scope of any other internal name is through that portion of the external procedure that could be executed while its declaring block is active. This implies that the scope of an internal variable, with the AUTOMATIC attribute, is that portion of the external procedure in which it would be expected to be allocated.

An explicit declaration of the STATIC attribute for an internal variable does *not* enlarge its scope.

So far in this publication, each data name in discussions and examples has been shown to refer to a single item of data.

In PL/I, data can be named so that an entire collection of data items can be referred to by a single name; certain subcollections can be referred to by another name, or individual elements can be referred to separately.

There are two kinds of data collections to which collective names can be given, *structures* and *arrays*. The major difference between the two is in the kind of data an array name or a structure name represents. An array is a table of homogeneous data items; each item has the same attributes as have all the other items. In an array of arithmetic items, for example, all of the items have the same base, scale, and precision; in a character-string array, all of the items have the same length. The elementary data items of a structure, on the other hand, need not be of the same data type nor have the same attributes. Some names in a structure might refer to string data while other names in the same structure could refer to arithmetic data.

Collective naming does not alter the data in any way. Data in a structure or an array is no different from what it would be if it were referred to by a single variable. Collective naming merely gives a programmer more convenience in referring to and manipulating data.

Structures

A structure is a graded system of names that refers to an area of internal storage. At the first level, a single name —the *major structure name*—refers to all of the data items stored in the entire area that is allocated to that structure. At the second level, certain portions of the area are renamed. This renaming continues until, at the deepest level, a single name, at any particular point in time, refers to a single data item; at the deepest level, in fact, each name is a variable, as previously defined.

Consider a program to figure a weekly payroll. One employee, John J. Doe, whose pay number is 68584, works 40 hours of regular time and 5 hours of overtime. He is paid $4.00 per hour for regular time and $6.00 for overtime.

His weekly pay record, with all of the above information, is read and assigned to a structure named PAYROLL. The information could be ordered:

```
DOE JOHN J  68584  40  05  400  600
```

If this were referred to merely by the name PAYROLL, it might be treated as a character string; but, as a character string, it would be difficult to get to individual items within the string, and arithmetic operations would involve conversion. However, a name can also be given to each element. The names for John Doe's pay record, and the data each name represents might, conceptually, look like this:

	LASTNAME	DOE
	FIRSTNAME	JOHN
	MIDDLENAME	J
PAYROLL	PAY_NO	68584
	REGLRHOURS	40
	OVTIMHRS	05
	STRATE	400
	OVRTIMRATE	600

Thus, a programmer can refer to the entire collection of data items by the name, PAYROLL, or he can refer to an individual item by an individual name.

It often is valuable to be able to refer collectively to more than one, but not all, of the variables in a structure. The names of the variables suggest an intermediate classification:

	NAME	LAST	DOE
		FIRST	JOHN
		MIDDLE	J
	PAY_NO		68584
PAYROLL			
	HRS	REGLR	40
		OVTIM	05
	RATE	STRATE	400
		OVRTIM	600

The major structure, PAYROLL, contains the substructures, NAME, HRS, and RATE. PAY_NO is not a substructure because it represents only a single data item.

During execution of a program in which PAYROLL is used, a number of different pay records would be read and assigned to PAYROLL. The corresponding elements of the records would each be assigned to its proper variable.

A reference to any structure or substructure is a reference to all of the data items referred to by the elementary names in that structure or substructure. It is, perhaps, clearer to say that a reference to any name is really a reference to the *names* at the next deeper level. That is, a reference to the name at the first level (PAYROLL, in the example) is a reference to all of the names at the second level (NAME, PAY_NO, HRS, RATE). A reference to a name at the second level (NAME) is a reference to all of the names in that sub-

structure at the third level (LAST, FIRST, and MIDDLE).

Since there are no levels deeper than LAST, FIRST, and MIDDLE, they are not substructures. They are elementary names, like PAY_NO.

When a structure is declared, the level of each name is indicated by a *level number*. The major structure name, at the first level, always is given the level number 1. Each name at a deeper level is given a greater number to indicate the level depth. For example:

```
DECLARE 1 PAYROLL, 2 NAME, 3 LAST, 3 FIRST,
  3 MIDDLE, 2 PAY_NO, 2 HRS, 3 REGLR, 3 OVTIM,
  2 RATE, 3 STRATE, 3 OVRTIM;
```

The same DECLARE statement could be written as follows:

```
DECLARE 1 PAYROLL,
  2 NAME,
      3 LAST,
      3 FIRST,
      3 MIDDLE,
  2 PAY_NO,
  2 HRS,
      3 REGLR,
      3 OVTIM,
  2 RATE,
      3 STRATE,
      3 OVRTIM;
```

The order of the appearance of names in a declare statement, along with their level numbers, determines the structuring. Except for the major structure name, no particular numbers must be specified. The only requirement is that deeper level names must have greater numbers. The major structure name must be declared with the level number 1, and it must be the first name listed in the structure declaration. It must be followed by one of the names at the second level. Each name at the second level must have a level number greater than 1, and if it is a substructure, it must be followed by the names within that substructure. Each substructure is completely declared before the next substructure declaration begins, and each substructure must have a level number equal to or less than the level number of the immediately preceding substructure at the same level. For example, if RATE were declared with the level number 3, it would be considered to be part of the substructure HRS.

The level numbers of the above structure might have been declared in this way:

```
DECLARE 1 PAYROLL,
        8 NAME,
            20 LAST,
            20 FIRST,
            20 MIDDLE,
        6 PAY_NO,
        2 HRS,
            3 REGLR,
            3 OVTIM,
        2 RATE,
            8 STRATE,
            8 OVRTIM;
```

Exactly the same structuring would result.

The number of levels allowed within a structure will vary from compiler to compiler.

When a structure is declared, attributes may be specified for the elementary item names; otherwise, default attributes would apply. Attributes can be factored in a structure declaration. For example:

```
DECLARE 1 PAYROLL,
        2 NAME,
            3 LAST CHARACTER (12),
            3 FIRST CHARACTER (8),
            3 MIDDLE CHARACTER (1),
        2 PAY_NO CHARACTER (5),
        2 HRS,
            (3 REGLR,
             3 OVTIM)
                FIXED DECIMAL (2),
        2 RATE,
            (3 STRATE,
             3 OVRTIM)
                FIXED DECIMAL (3,2);
```

Although level numbers *must* appear with a name in the DECLARE statement, they do not appear with the name in any other reference to it.

Qualified Names

All names within a single procedure must be unique. But within structures, it is often convenient to be able to use the same identifier for related names. In the above structure, for example, it would be convenient to refer to the items in HRS and RATE AS "regular hours" and "regular rate" and "overtime hours" and "overtime rate."

In fact, the elements can be given the same names. The last portion of the structure might be declared:

```
2 HRS,
    3 REGLR,
    3 OVRTIM,
2 RATE,
    3 REGLR,
    3 OVRTIM;
```

The use of a *qualified name* in referring to the individual item avoids ambiguity. A qualified name is a substructure or element name that is made unique by qualifying it with one or more names at a higher level. The individual names within a qualified name are separated by a period. The above items could be referred to by the following qualified names:

```
HRS . REGLR
RATE . REGLR
HRS . OVRTIM,
RATE . OVRTIM
```

Any of the names in PAYROLL, except PAYROLL itself, need not be unique within the procedure in which it is declared.

They all could be qualified. For example:

```
PAYROLL . NAME
PAYROLL . NAME . LAST
   or     NAME . LAST
   or     PAYROLL . LAST
```

All of the qualifying names need not appear. A name need be qualified only so far as is necessary to make it unique.

Structure Expressions

Structure and substructure names can be used in arithmetic or string expressions.

For example, consider the following statement:

```
PAY = HRS * RATE;
```

PAY, in this case, must be a structure name referring to two arithmetic variables. The above statement would have the same effect as the following two statements:

```
PAY.REGLR=HRS.REGLR*RATE.REGLR;
PAY.OVRTIM=HRS.OVRTIM*RATE.OVRTIM;
```

A reference to a structure or substructure name in an expression is a reference to all of the elements within that structure or substructure. In the case of an arithmetic expression, the expression is evaluated separately for each set of corresponding elements.

A structure expression need not be limited to structure names. The expression HRS/5 would be valid and it would give the same result as the two expressions, HRS . REGLR / 5 and HRS . OVRTIM / 5.

Summary

A structure is a hierarchy of names built upon a list of individual variables, so that the variables can be referred to singly or in groups or so that the entire list of variables can be referred to by a single name.

Structuring has no effect upon the data itself; it merely provides greater flexibility in referring to the data.

Arrays

An array is a named table of data items all of which have identical attributes. Only the array, itself, is given a name. An individual item of an array is referred to by giving its location within the array. The location is specified by a *subscript* following the array name.

Assume TABLE has been declared to be an array of 12 elements. TABLE (1) refers to the first item in the list, TABLE (2) to the second, TABLE (3) to the third, etc. Each of the numbers, (1), (2), or (3), is a subscript that gives the location, within TABLE, of a particular data item.

An array name is declared in a DECLARE statement by giving its name, the number of elements in the array, and the attributes of the items.

```
DECLARE TABLE (12) DECIMAL FIXED (2);
```

This specifies that the name, TABLE, refers to an array of 12 data items, each of which will have a value that can be represented by two decimal digits. TABLE is declared to have the *dimension attribute* of (12). The *bounds* of the dimension are 1 and 12. In any declaration, the dimension attribute must immediately follow the array name. Assume that the following numbers have been assigned to TABLE:

```
31
43
42
57
64
73
79
79
69
58
49
40
```

Thus, TABLE (1) would refer to the data item 31, TABLE (6) to 73, TABLE (12) to 40. The expression TABLE (7) + TABLE (1) would yield a value of 110.

Assume that the values assigned to TABLE represent the average temperature of the months of a particular year. TABLE (1) is the January average, TABLE (2) is the February average, etc. As TABLE was declared in the previous DECLARE statement, the data items could be referred to singly or as a whole. For various reasons, a programmer might want to consider the year as divided into quarters; it might be convenient to be able to use a single reference to all of the average temperatures of a quarter of a year. He might declare TABLE as follows:

```
DECLARE TABLE (4,3);
```

In this statement, the dimension attribute specifies that TABLE refers to 12 data items, but that TABLE is considered to consist of four lists of three items each. It has two dimensions, one with bounds of 1 and 4, the other with bounds of 1 and 3. The data might be recorded in storage in exactly the same way as with the first declaration, but *conceptually* it is ordered differently.

Following are two different ways in which the arrangement might be conceptually illustrated. The one on the left treats it as four consecutive lists of three items each; the one on the right treats it as a matrix of four rows and three columns.

			TABLE	(n,1)	(n,2)	(n,3)
31	TABLE	(1,1)	(1,m)	31	43	42
43		(1,2)	(2,m)	57	64	73
42		(1,3)	(3,m)	79	79	69
57		(2,1)	(4,m)	58	49	40
64		(2,2)				
73		(2,3)				
79		(3,1)				
79		(3,2)				
69		(3,3)				
58		(4,1)				
49		(4,2)				
40		(4,3)				

With such a two-dimensional array, a programmer can refer to cross sections of the array (that is, to all of the variations within the bounds of one dimension) by

writing an asterisk in place of that dimension in a subscripted name. For example:

```
PUT LIST (TABLE (1,*));
```

The above statement would have exactly the same result as the following:

```
PUT LIST (TABLE (1,1), TABLE (1,2),
  TABLE (1,3));
```

In the example below, the first statement would have exactly the same effect as the *four* statements that follow it.

```
A = TABLE (*,2)*2;
A(1) = TABLE (1,2) * 2;
A(2) = TABLE (2,2) * 2;
A(3) = TABLE (3,2) * 2;
A(4) = TABLE (4,2) * 2;
```

Note that A must be an array of four elements, since the expression TABLE (*,2) * 2, would, using TABLE as declared previously, result in four separate values.

In whatever concept one chooses to visualize an array, it is important to remember that although the dimension attribute in the preceding example specifies a two-dimensional array, the location of each item in storage could be exactly the same as if it were a one-dimensional or three-dimensional array. It could, in fact, actually be declared as a three-dimensional array with the following DECLARE statement:

```
DECLARE TABLE (2,3,2);
```

Note that the number of specifications, separated by commas, is the same as the number of dimensions, and that the product of the numbers is equal to the number of items in the array: (12), (4,3), (2,3,2,).

Using the same data, TABLE (2,3,2) might be envisioned as follows:

31	TABLE	(1,1,1)	TABLE	(1,n,m)	(1,n,1) (1,n,2)
43		(1,1,2)			
42		(1,2,1)		(1,1,m)	31 43
57		(1,2,2)		(1,2,m)	42 57
64		(1,3,1)		(1,3,m)	64 73
73		(1,3,2)			
79		(2,1,1)			
79		(2,1,2)			
69		(2,2,1)			
58		(2,2,2)	TABLE	(2,n,m)	(2,n,1) (2,n,2)
49		(2,3,1)			
40		(2,3,2)		(2,1,m)	79 79
				(2,2,m)	69 58
				(2,3,m)	49 40

The dimension attribute (2,3,2), specifies that TABLE represents a list of 12 data items and that the list will be referred to as if it consists of two sub-lists, each of which is further divided into three sub-lists of two items each.

The same result could be achieved through declaring TABLE as a structure:

```
DECLARE 1 TABLE,
          2 TABLE_1,
            3 TABLE_1_1,
              4 TABLE_1_1_1,    (31)
              4 TABLE_1_1_2,    (43)
            3 TABLE_1_2,
              4 TABLE_1_2_1,    (42)
              4 TABLE_1_2_2,    (57)
            3 TABLE_1_3,
              4 TABLE_1_3_1,    (64)
              4 TABLE_1_3_2,    (73)
          2 TABLE_2,
            3 TABLE_2_1,
              4 TABLE_2_1_1,    (79)
              4 TABLE_2_1_2,    (79)
            3 TABLE_2_2,
              4 TABLE_2_2_1,    (69)
              4 TABLE_2_2_2,    (58)
            3 TABLE_2_3,
              4 TABLE_2_3_1,    (49)
              4 TABLE_2_3_2;    (40)
```

The actual data locations referred to could be the same, no matter how the list is referred to.

Declaring a three-dimensional array for the monthly temperature averages for one year means that the year is considered as two six-month periods, with each six-month period further divided into three two-month periods.

A more practical application of a three-dimensional array, used in connection with monthly temperature averages, would be one that contained all the monthly averages over a period of years, say ten years. The averages for each year probably would be referred to as in TABLE (4,3), and the third dimension of the array would vary through the ten years. It could be declared as follows:

```
DECLARE TABLE (10,4,3);
```

If the list contained temperatures for the years 1951 to 1960, the average temperature for July 1957 would be referred to as TABLE (7,3,1), or the seventh year, the third quarter, the first month.

A four-dimensional array might be specified for a monthly average temperature table that spanned a century. The century could be divided into decades, each decade into years, each year into quarters, and each quarter into months.

```
DECLARE TABLE (10, 10, 4, 3);
```

If the above were declared, the subscripted name, TABLE (6,2,3,1), would represent the first month of the third quarter of the second year of the sixth decade.

The limit to the number of dimensions that can be declared for an array is specified for each PL/I compiler.

If the average temperatures were available over a period of two centuries, TABLE might be declared as follows:

```
DECLARE TABLE (2, 10, 10, 4, 3);
```

There would be 2,400 items in this five-dimensional array (2 x 10 x 10 x 4 x 3).

Think of the rightmost number of any dimension attribute as specifying a one-dimensional array. The next number to the left specifies the number of one-dimensional arrays that make up a two-dimensional array. The

next number to the left specifies the number of two-dimensional arrays that make up a three-dimensional array, etc. In the declaration of TABLE as a five-dimensional array, the leftmost number (2) specifies the number of four-dimensional arrays that make up the five-dimensional array.

Variable Subscripts

Subscripts in a name used to refer to an element of an array need not be numbers. Any variable having an arithmetic integer value can be used in place of any of the numbers of a subscript. For example, if ALPHA has the value 2 and BETA has the value 3, the expression A (ALPHA, BETA), is the same as the expression A(2,3).

The ability to use variables as subscripts is especially valuable in DO groups in which array data is manipulated. For example, the statements below specify that the average temperature be written for the month of February of each year of a century (assume that the average temperatures have been recorded for every month of every year since January 1865; assume further that the variable YEAR has been declared with the attributes DECIMAL FIXED (4) and has been given an initial value of 1865).

```
DO I = 1 TO 10;
    DO J = 1 TO 10;
    PUT LIST (YEAR, TABLE(I,J,1,2));
    YEAR = YEAR + 1;
    END;
END;
```

When the first DO statement is executed, I is set equal to 1, and control passes to the second DO statement, where J is set equal to 1. When the PUT statement is executed the first time, both I and J in the expression, TABLE (I,J,1,2), have the value 1. The year 1865 and the temperature for February 1865 are written on the standard output file. YEAR is made equal to 1866, the END statement is executed, and control returns to the innermost DO statement. J is incremented, and the operation is repeated. When the inner DO group has been executed ten times, control returns to the outer DO statement. I is incremented, and control again passes to the inner DO statement. J is reset to 1, and the loop is executed 10 more times. This time, the value of I is two; the temperatures of the second decade are written. The statements are executed until TABLE (10,10,1,2) has been written.

Array Expressions

A single element of an array of arithmetic data is a single arithmetic data item. It can be manipulated in the same way as any other such item.

Likewise, entire arithmetic arrays (or cross-sections of them) can be used in arithmetic expressions. Such expressions are evaluated element by element, and a value is returned for each element. Expressions containing array names are *array expressions*, and they always return an array value.

NOTE: Although an array may be conceived as being similar to an arithmetic matrix, array expressions are not always expressions of conventional matrix algebra.

Since array expressions are always evaluated on an element-by-element basis, all arrays referred to in a single array expression must have identical dimensions and bounds.

Examples:

If A is the array:	3	5	8
	−2	7	6
	5	−6	4
Then −A is the array:	−3	−5	−8
	2	−7	−6
	−5	6	−4
A * 2 is the array:	6	10	16
	−4	14	12
	10	−12	8
If B is the array:	2	1	3
	4	2	1
	8	−2	−3
Then A + B is the array:	5	6	11
	−2	9	7
	13	−8	1
A * B is the array:	6	5	24
	−8	14	6
	40	12	−12

Summary

An array is a list of homogeneous elements that are referred to by subscripted names. The subscript refers to the location of the element within the list.

The dimensions declared for an array are merely an indication of how elements will be referred to; they do not indicate any particular arrangement of the actual data items in the storage area.

All of the arrays in the discussion and in the examples have been arithmetic arrays. PL/I allows arrays of string data, either character-string or bit-string; all elements must be of equal length. A programmer may even declare a statement-label array, in which each element refers to a statement label. Arrays may be elements of a structure. And, in fact, an array may be an array of structures, in which each element of the array is an entire structure. All of the structures of an array of structures must be identical.

Discussions of these other types of arrays can be found in other PL/I publications.

Chapter 9: A Table Look-Up Procedure

The process of looking up a value in a keyed table is so common an experience that one is apt not to realize the number of steps involved. For instance, finding a number in a telephone directory is done almost automatically.

A telephone directory consists of two lists, a name list arranged alphabetically and a number list arranged in no apparent order. Each telephone number, however, occupies a position in its list corresponding to the position of the appropriate name in the list of names.

A search is made through the list of names. The initial letter of the particular name eliminates all but a specific section of the directory; other subsections are subsequently eliminated until the correct name is found. The value sought, the telephone number, is then secured from the corresponding position in the second list. If the name is not in the first list, the telephone number is not in the second.

A computer could be programmed to look up telephone numbers through the use of arrays and DO loops. A simple example is shown below.

Assume that the elements of a one-dimensional array NAME_LIST are the names of 100 persons listed in a telephone directory. Assume further that the telephone number for each person is the corresponding element of another one-dimensional array NUMBER_LIST. NAME is a variable that represents the name of the person whose telephone number is sought, and NUMBER is a variable to which the number, when found, is to be assigned. The look-up operation might be written:

```
LOOP: DO I = 1 TO 100;
        IF NAME = NAME_LIST (I)
          THEN GO TO FIND;
        END LOOP;
FIND: NUMBER = NUMBER_LIST (I);
```

During execution of the DO loop, NAME is compared with each of the elements in NAME_LIST, beginning with the first. As I is incremented, NAME_LIST (I) refers to successive elements in the array. When a match is found between NAME and an element of NAME_LIST, the telephone number for that name is secured from the corresponding position in the array NUMBER_LIST and is assigned to NUMBER.

Despite the speed of a computer, such step-by-step searching is inefficient. Various techniques are used in programming to allow faster table searching. Array manipulation, as provided in PL/I, is one such technique. For example, both of the arrays shown above could be declared as two-dimensional arrays—NAME_LIST (10,10) and NUMBER_LIST (10,10). Each still would contain 100 elements, but each conceptually would consist of 10 lists of 10 elements each, allowing for the elimination of 10 elements with a single test. The statements directing the search might be written:

```
        J = 10;
        DO I = 1 TO 9 WHILE
             (NAME> = NAME_LIST (I,J));
        IF NAME = NAME_LIST (I,J)
          THEN GO TO FIND;
        END;
        DO J = 1 TO 10;
        IF NAME = NAME_LIST (I,J)
          THEN GO TO FIND;
        END;
FIND:   NUMBER = NUMBER_LIST (I,J);
```

The first DO loop instructs the computer to examine the last name in each of the first nine groups of ten. There is no need to test the tenth group, since after the ninth is eliminated, only the tenth remains.

The DO statement specifies that two tests are to be made at the head of the group; either could cause control to be transferred out of the loop. The first test is of the value of I; when it exceeds 9 (that is, when it is incremented to 10) control passes out of the loop. The second test compares the value of NAME with the current value of NAME_LIST (I,J). If NAME is greater than or equal to the element of the list, the DO group is executed. If NAME is less than the current element of NAME_LIST, then the name being sought must be one of the nine preceding names.

When control enters the second DO loop, the value of I specifies the group of ten that contains the name being sought. If NAME is less than NAME_LIST (4,10), for example, it has already been shown to be greater than NAME_LIST (3,10), and the second DO group makes its first test against NAME_LIST (4,1). If, however, NAME is greater than NAME_LIST (9,10), the value of I will have been incremented to 10 (after the DO group is executed for the ninth time, I is incremented and tested to see if it is greater than 9; since it has been incremented to 10, control passes out of the group with the value of I remaining at 10).

When the second DO group is entered, J is reset to one and is incremented until NAME matches an element in NAME_LIST. Note that J will be incremented to 10 only if the name is the last element in the entire table; the tenth element of each of the first nine groups is tested for equality in the first DO group.

When a match is found between NAME and an ele-

ment of NAME_LIST, control is transferred to FIND. Since each telephone number is in the same position in NUMBER_LIST as the corresponding name in NAME_LIST, the subscripts that specify the name in NAME_LIST also specify the number in NUMBER_LIST. Consequently, NUMBER_LIST (I,J) is assigned to NUMBER. The assumption is that the value of NAME always can be found in NAME_LIST; no provision is made for a case in which a name being sought is not in the table.

Figure 13 is a table look-up procedure that provides for cases in which the value sought might not actually be in the table. Assume that the procedure is used in a program by an insurance company to find the cost of different kinds of insurance policies. The variable X represents a code number that identifies a specific type of policy; the object of the search is the premium for $1,000 of coverage of that type of policy. The array A is a list of 1,000 different code numbers, arranged in ascending order. The array B is a corresponding list of premiums. When X matches a code number in A, the corresponding element of B is assigned to the variable Y.

Assume further that the data stored in the two tables is read during execution of another procedure. All variables are declared with the EXTERNAL attribute so they can be referred to in other procedures of the program. Any procedure that invokes LOOKUP will first assign to X the code number in question. After control returns from LOOKUP to the invoking procedure, the appropriate premium cost will be available at Y.

```
 LOOKUP: PROCEDURE;
         DECLARE ((X,A(10,10,10)) FIXED(4), (Y,
           B(10,10,10)) FIXED(4,2))
           DECIMAL EXTERNAL;
/* IS X IN TABLE A? */
         IF X < A(1,1,1)
            THEN GO TO ZERO;
         IF X > A(10,10,10)
            THEN GO TO ZERO;
/* INITIALIZE J AND K */
         J,K = 10;
/* FIND THE CORRECT GROUP OF 100 */
         DO I = 1 TO 9 WHILE (X >= A(I,J,K));
             IF X = A(I,J,K)
                THEN GO TO TABLE_B;
         END;
/* FIND THE CORRECT GROUP OF 10 */
         DO J = 1 TO 9 WHILE (X >= A(I,J,K));
             IF X = A(I,J,K)
                THEN GO TO TABLE_B;
         END;
/* FIND THE CORRECT ELEMENT */
         DO K = 1 TO 10 WHILE (X >= A(I,J,K));
             IF X = A(I,J,K)
                THEN GO TO TABLE_B;
         END;
/* IF X IS NOT IN TABLE-- */
    ZERO: Y = 0;
         RETURN;
/* ASSIGN THE PROPER VALUE */
TABLE_B: Y = B(I,J,K);
         END LOOKUP;
```

Figure 13. A Table Look-up Procedure.

Both of the arrays are three-dimensional arrays of 1,000 elements; consequently, the first subscript refers to groups of 100, the second subscript to subgroups of 10, and the third subscript to single elements within a subgroup.

The first tests made in LOOKUP are to discover whether or not the value of X lies within the range of the table. If it does not—if it is less than A(1,1,1) or greater than A(10,10,10)—control goes to ZERO, the value of zero is assigned to Y, and control is returned to the invoking procedure where, presumably, Y will be tested for a zero value.

If X does lie within the range of the table, J and K are set to 10 in the *multiple assignment* statement, and control passes into the first DO loop. (With PL/I, the same value can be assigned to a number of different variables in a single statement.) The operation of the DO loop is the same as that described in the previous example except that here, the WHILE clause compares X with the last element in each group of 100. The first comparison is with A(1,10,10), or the 100th element in the array. If X is greater than or equal to A(1,10,10), the DO loop is executed. If X is shown to be equal to A(1,10,10), control is transferred to the statement labeled TABLE_B and the correct premium cost—the corresponding element of B—is assigned to Y. Then control reaches the END statement of the procedure and returns to the invoking procedure. If X is *not* equal to A(1,10,10), control returns to the heading of the DO group, I is incremented to 2, and the WHILE clause compares X with A(2,10,10).

The looping continues until (1) an exact match is found, (2) the loop has been executed nine times, or (3) the test in the WHILE clause shows that X is less than the current value of A(I,10,10).

In the first case, the correct value from B is assigned to Y. In the second case, control passes out of the first DO loop to the second DO loop with the value of I set at 10 (the variable always is incremented beyond the top limit set in the DO statement). In the third case, control passes to the second DO loop with the value of I specifying the group of 100 whose range contains the value of X (because X is less than the current value of A(I,10,10) but greater than the last previous value of A(I,10,10).

The second DO group functions exactly as the first except that the object is to find the correct group of ten elements. The first and third subscripts remain constant, the first at the current value of I, and the third at 10. As in the first loop, execution continues until one of the three conditions is satisfied: (1) the correct value

is found, (2) the loop is executed nine times, or (3) the WHILE clause test shows that X is less than A(I,J,10).

When the correct group of ten elements is found, control is transferred to the third DO loop, which steps sequentially through the group of ten, with the third subscript incremented from 1 to 10. In most cases, X will be found before K reaches 10, since K is set to 10 through the first two DO loops. If the correct element is the last one in the table, K must be incremented to 10 before it will be found.

The WHILE clause appears in the third DO statement as an error-control expression, because the value of X may not be a valid code number in the table. Although the table of code numbers must be arranged in ascending order, it is not necessary that every number be included; for example, consecutive code numbers might be 8723, 8728, 8843. If the value of X were 8729, it would be tested against 8728, K would be incremented, and the WHILE clause would show that X is less than 8843. In this case, control would pass out of the last DO group to the statement labeled ZERO. The variable Y would be assigned the value zero, and control would return to the invoking procedure.

This procedure has been described to demonstrate how PL/I provides ease in manipulating data when it is organized in arrays. It should not be concluded that this is the best way to handle a table look-up. A more efficient method of handling what is basically the same procedure is explained in the following chapter.

The name of a variable, a file, etc. can be referred to in a procedure only if that procedure lies within the scope of the name. There is, however, a way in which the scope of a name can be extended to a procedure in which the name normally would not be known. This is accomplished by *passing* the name as an *argument of the invocation.* Consider the following example:

```
ALPHA: PROCEDURE;
       DECLARE COST FIXED DECIMAL
                  (4,2),
               TABLE (10,10)
                 FIXED DECIMAL (6),
               OUTFILE FILE OUTPUT;
           .
           .
           .
       CALL BETA (COST,TABLE,OUTFILE);
           .
           .
           .
       END ALPHA;
```

In the CALL statement, COST, TABLE, and OUTFILE are arguments that are passed to BETA as part of the invocation. Note that the list of names, called the *argument list,* is enclosed in parentheses and that the names are separated by commas. Assume that ALPHA is one procedure of a program used to compute a company's inventory of stock on hand, and that part of the task of ALPHA is to compute the number of each of 100 different items, all of which have the same cost. The total number of each item is assigned as one element of the 100-element array TABLE. A separate objective of the program is to compute the total cost of all stock on hand. That job could be handled by procedure BETA:

```
BETA: PROCEDURE (X,Y,FILE_A);
      DECLARE (X FIXED (4,2),
               Y (10,10) FIXED (6),
               Z (10,10) FIXED (10,2),
               TOTAL_COST FIXED (15,2)
                 STATIC INITIAL (0))
                 DECIMAL;
      Z = Y * X;
      TOTAL_COST = TOTAL_COST + SUM (Z);
      PUT FILE (FILE_A)
        LIST (Z,TOTAL_COST);
      END BETA;
```

Procedure BETA is a *subroutine,* a procedure to which arguments are passed in the invoking CALL statement. In the PROCEDURE statement, the names X, Y, and FILE_A are contextually declared to be *parameters.* A parameter is a name used in a procedure to represent another name (or some other expression) that is passed to that procedure as an argument of the invocation. A name is contextually declared to be a parameter through its appearance in the *parameter list* of a PROCEDURE or ENTRY statement.

The CALL BETA statement in ALPHA invokes BETA and specifies that any reference, within BETA, to the names in the parameter list – X, Y, and FILE_A – actually are references to the names in the argument list of the invocation – COST, TABLE, and OUTFILE. Names in an argument list are always associated with names in the parameter list in the order in which they appear; that is, the first argument is associated with the first parameter, the second argument with the second parameter, etc. The parameter list, like the argument list, must be enclosed in parentheses and names within the list must be separated by commas.

In the preceding example, BETA is invoked by a statement in ALPHA, and the arguments passed are names that are known in ALPHA. BETA also could be invoked from other procedures, to compute and write costs of other inventory items that are stored in other 100-element arrays. With each invocation, an argument, known within the invoking procedure, must be passed for each parameter. The question of scope of the names does not arise, since the passing of a name as an argument effectively extends the scope of that name through the procedure to which the name is passed.

The passing of an argument does *not* perform the same operation as an assignment statement. The name, itself, is passed, not the value it represents. Consequently, storage already allocated for a variable before it is passed as an argument need not be duplicated as allocated storage when the subroutine is activated. However, attributes must be declared – explicitly, contextually, or implicitly – for each parameter within the procedure that is headed by the statement in which the parameter is contextually declared.

In BETA, for example, attributes are explicitly declared for both X and Y; the FILE and OUTPUT attributes are contextually declared for FILE_A in the PUT statement. But a reference to X, Y, or FILE_A is actually a reference to another name. When BETA is invoked by the statement shown in ALPHA, any reference to X is a reference to COST, any reference to Y is a reference to TABLE, and any reference to FILE_A is a reference to OUTFILE. Since storage is allocated for COST and TABLE when ALPHA is activated, no further allocation for them is necessary when BETA is activated. Variables that are not parameters will, of course, be allocated for a subroutine. The variable TOTAL_COST, declared in BETA, will be allocated at

the beginning of the program, since TOTAL_COST is declared to have the STATIC storage class attribute. And storage for the array variable Z will be allocated each time BETA is invoked.

In the execution of BETA, the array Z is used as a work area to hold the results of the array expression until they can be written. The variable TOTAL_COST is the cost of the total inventory; it is initially set to zero and is updated and written each time BETA is executed.

The assignment statement

```
TOTAL_COST = TOTAL_COST + SUM(Z);
```

makes reference to another kind of procedure that is similar to a subroutine. The name SUM is the entry name of a *function procedure.* A function procedure (commonly called a *function*) differs from a subroutine in the way it is invoked. A subroutine is invoked by a CALL statement; a function is invoked by a *function reference,* an example of which is contained in the assignment statement. A function reference is the appearance, in an expression, of a function entry name with associated arguments.

In the above example, SUM is the entry name of a procedure that computes the sum of all the elements of an array. The appearance of the name SUM and the argument Z invokes the function SUM and passes the name Z to a parameter listed in the PROCEDURE statement of SUM. The function SUM is executed, and the computed sum of the elements of Z is returned as a value to the point of invocation, that is, to the function reference. The expression then is executed as if the statement had specified the actual value rather than the function reference.

The function procedure SUM is a *built-in function;* that is, it is provided by PL/I as a part of the compiler. There are a number of built-in functions, which a programmer using PL/I need not write as separate procedures; they are a part of the language, and any of them may be invoked merely by a function reference. (A complete list of built-in functions and their descriptions appears in the publication *PL/I: Language Specifications.*)

A programmer may, however, write a procedure to be used as a function. The procedure LOOKUP, discussed in Chapter 9, might be written as the following function procedure:

```
LOOKUP: PROCEDURE (X,A,B);
        DECLARE ((X,A(10,10,10)) FIXED (4),
                  B (10,10,10) FIXED (4,2))
                    DECIMAL;
/* IS X IN TABLE? */
        IF X<A(1,1,1)
           THEN RETURN (0);
        IF X>A(10,10,10)
           THEN RETURN (0);
/* FIND THE CORRECT GROUP OF 100 */
        DO I = 1 TO 9 WHILE (X >= A (I,10,10));
           IF X = A (I,10,10)
              THEN RETURN (B(I,10,10));
        END;
/* FIND THE CORRECT GROUP OF 10 */
        DO J = 1 TO 9 WHILE (X >= A(I,J,10));
           IF X = A(I,J,10)
              THEN RETURN (B (I,J,10));
        END;
/* FIND THE CORRECT ELEMENT */
        DO K = 1 TO 10 WHILE (X >= A(I,J,K));
           IF X = A(I,J,K)
              THEN RETURN (B(I,J,K));
        END;
        RETURN (0);
        END LOOKUP;
```

In the PROCEDURE statement, the names X, A, and B are contextually declared to be parameters; the DECLARE statement specifies attributes for them. Assume, again, that LOOKUP is a part of an insurance company program. It might be invoked by the reference to LOOKUP in the assignment statement shown below.

```
          .
          .
          .
PREMIUM = THOUSANDS * LOOKUP
            (CODE,CODE_LIST,COST_LIST);
IF PREMIUM = 0
   THEN PUT LIST (CODE, 'NOT IN CODE_LIST');
   ELSE ... ;
```

The assignment statement is written to compute the total premium cost of a particular insurance policy. The variable THOUSANDS represents the total value of the policy in units of $1,000. The argument CODE is a code number for the particular kind of policy; CODE_LIST is a table of such code numbers; and COST_LIST is the corresponding table of costs. When LOOKUP is invoked, these arguments are passed to it, so that during execution, X refers to CODE, A refers to CODE_LIST, and B refers to COST_LIST.

The same procedure might be invoked from other points in the program with different arguments being passed. Thus, the same procedure could be used to search through a number of different 1000-element tables.

Execution of the procedure is the same as was described for LOOKUP in Chapter 9 except for the way in which the procedure is terminated and the way in which the value being sought is assigned.

A function procedure is terminated by a RETURN statement of the following form:

```
RETURN (expression);
```

It might be interpreted as meaning "return the following value – along with control – to the point of invocation." For example, if CODE is not included in CODE_LIST, the RETURN (0) statement would return a zero value to the point of invocation, and the invoking as-

signment statement would be evaluated as if it had been written as follows:

```
PREMIUM = THOUSANDS * 0;
```

A function can return only a *single* value; consequently, the expression in a RETURN statement must be an arithmetic or string expression that represents a single value.[1] When CODE is found in CODE_LIST, the RETURN statement returns the corresponding value from COST_LIST.

Summary

Through the specification of arguments and parameters, subroutines and functions can be used throughout a program to perform the same operations upon many different data items whose names may be known only within the invoking procedure.

A subroutine or a function need not lie within the scope of a name that is referred to within the subroutine or function; the passing of the name as an argument of the invocation effectively extends the scope of the name through the procedure that is thus invoked.

[1]Some built-in functions can return array or structure values, but programmer-written functions cannot.

Any kind of name can be passed as an argument. Examples in this chapter have illustrated data arguments and file arguments. Entry names and statement labels also can be passed, so that a CALL statement or a GO TO statement can be written in a procedure to refer to an appropriate name that is passed to it as an argument. An exception is that most built-in function names cannot be passed as arguments.

Arguments may be expressions other than names. A constant can be passed, as can an expression containing arithmetic or string operators.

An argument must be passed for each parameter appearing in the parameter list of the invoked procedure. Arguments in the argument list must appear in the same order as the corresponding parameters in the parameter list. Both the argument list and the parameter list must be enclosed in parentheses; expressions in either list are separated by commas.

Since the name, not the value, is passed, storage need not be allocated for each parameter that represents a data variable. However, attributes must be declared for each parameter – explicitly, contextually, or implicitly – within the subroutine or function.

Chapter 11: Input/Output

The basic function of input and output is data transmission, getting the data to be processed and returning the results of the processing. Using one of the simplest forms of input and output (as has been shown in examples), a programmer need write only the nature of the operation, GET or PUT, and a list of data names that specify where the data is to be stored or where the data to be written can be found. The full range of input and output operations, however, allows a programmer to edit data and insert symbols such as a dollar sign and a decimal point, and to control the format and layout of the printed page.

Data on an external medium is collected in a *data set.* In PL/I, a *file name* is declared for each data set, and the file name is given file attributes that describe the data set and the manner in which it will be handled. For example, the INPUT attribute specifies an existing data set that is to be read; OUTPUT specifies a data set that is to be written.

PL/I deals with data sets in two different kinds of data transmission, *stream oriented* and *record oriented.* With stream-oriented transmission, the data set is considered to be a continuous stream of data items, in character form, to be assigned from the stream to variables, or from variables into the stream. With record-oriented transmission, the data set is considered to consist of a collection of physically separate records, each of which consists of one or more data items in any form; each record is transmitted as an entity directly to or from a variable or directly to or from an addressable buffer.

The nature of the two kinds of transmission is expressed in the keywords of the statements used in each. The basic input/output statements in stream-oriented transmission are GET and PUT, get the next data items from the stream or put the specified data items into the stream. In record-oriented transmission, the comparable statements are READ and WRITE, read the next record directly from the data set or write the specified record directly into the data set. Stream-oriented transmission implies data conversion. All of the items in the stream are in character form. On input, they are converted automatically to conform to the attributes of the variable to which they are assigned; on output, data items are converted, if necessary, to characters. In record-oriented transmission, there is no conversion; data is transmitted exactly as it is recorded, either internally or on the external medium.

File Declaration

File attributes, like data attributes, may be declared implicitly, contextually, or explicitly. For example, any file name is assumed to have the EXTERNAL scope attributes unless the INTERNAL attribute is declared explicitly. The INPUT attribute is assumed, but a WRITE statement or a PUT statement can cause contextual declaration of the OUTPUT attribute.

Unlike data attributes, however, file attributes can be explicitly declared in two different ways, by their appearance in a DECLARE statement or by their appearance in an OPEN statement. A file must be opened before it can be used. The opening of a file associates a particular data set with the specified file name (a data set can have a name that is different from the file name, although usually the two names are the same). The opening also allows for checking or writing of tape labels, and it provides a means for a programmer to specify certain layout specifications if output into the file is to be printed.

A file may be opened explicitly or implicitly. It is opened explicitly through execution of an OPEN statement. It is opened implicitly if one of a group of statements, such as GET or WRITE, is executed prior to execution of an OPEN statement that specifies the same file name.

Standard Files

There are two standard system files available for use by a PL/I programmer. The first is the standard system input file called SYSIN. The second is the standard system output print file called SYSPRINT. A GET statement that specifies no file name is equivalent to a reference to SYSIN. A PUT statement that specifies no file name is equivalent to a reference to SYSPRINT. This type of reference applies only to these two statements; any other reference to either file must be stated explicitly. The standard system files need not be declared or opened explicitly; a standard set of attributes is applied automatically. For SYSIN, these attributes specify that it is a stream-oriented input file from which the data will be obtained in the same sequence as it appears in the file. For SYSPRINT, the attributes specify stream-oriented output that is to be printed. Both file names are assumed to have the EXTERNAL scope attribute. Any of the attributes can be changed by explicit declaration.

Stream-Oriented Transmission

There are three modes of stream-oriented transmission, *list-directed* transmission, *data-directed* transmission, and *edit-directed* transmission. Whichever is used, the following information must be specified explicitly or implicitly for each GET or PUT statement:

1. The name of the data set from which data is to be obtained or to which data is to be assigned; that is, the file name.
2. A list of variables representing storage areas to which data items are to be assigned during input, or from which data items are to be obtained during output. Such a list is known as a *data list.*
3. The format of each data item.

In certain circumstances, all of this required information can be implied; in other cases, only a portion of it need be stated explicitly. If the file name is not specified, either SYSIN or SYSPRINT is assumed; this applies to any of the three modes of stream transmission. In list-directed and data-directed transmission, the format need not be specified. In data-directed input, not even the data list need be specified.

Data items in the stream are written as valid arithmetic or string constants. For an easier understanding of stream-oriented data transmission, it is important to remember that data in the stream always is in character form. Although a bit string is recorded inside the computer in binary digits, the only way it can be printed is as a string of 1 and 0 characters. Except for the enclosing quotation marks, there is no way to see the difference between the printed character string '-38.32' and the printed fixed-point decimal number -38.32. The character string is composed of six characters (the minus sign and the decimal point each require a storage location); it would be truncated if it were assigned to a string variable with a declared length of less than six. Yet, the precision of the fixed-point decimal number is (4,2); only the number of digits is considered.

On stream-oriented input, such characters are converted to conform with the attributes of the variable to which they are assigned. On output, the internal data representation is converted to character representation. For example, the 1 and 0 *bits* of a bit string in storage are converted to the *characters* 1 and 0.

Binary arithmetic constants cannot be written as output. The value of a binary fixed-point data item is converted to decimal fixed-point notation before being written; the value of a binary floating-point data item is converted to decimal floating-point notation.

List-Directed Data Transmission

List-directed data transmission permits the programmer to specify the variables to which data is to be assigned (or from which data is to be acquired) without specifically stating a format for the data. The format is a standard one and is supplied by the compiler. List-directed transmission provides easy input/output operations for programmers who do not require a special format either on input or output, and who are interested only in a list of the results of the processing.

The elementary form of the GET and PUT statements, when used for list-directed input and output, is:

```
GET FILE (file name) LIST (data list);
PUT FILE (file name) LIST (data list);
```

The FILE (file name) is the *file specification;* LIST (data list) is the *data specification.* The file name and the data list must each be enclosed in parentheses. The two specifications need not appear in a particular order. If the file specification is omitted, it is assumed that one of the standard files is to be used. In list-directed transmission, the keyword, LIST, must always head the data specification.

List-Directed Data Lists

The data list in a list-directed GET statement is a list of variables (representing internal storage areas) to which data items in the data stream are to be assigned. The variables in a data list are separated by commas. An example of a list-directed GET statement follows:

```
GET FILE (MASTER) LIST
  (LOAN_#, PRINCIPAL, RATE);
```

The GET statement in the above example causes three data items from MASTER file to be assigned to the variables of the data list in the sequence in which they are listed; that is, the first data item is assigned to LOAN_#, the second to PRINCIPAL, and the third to RATE. Assignment stops at this point because the data list has been exhausted.

The data list in a list-directed PUT statement differs from that of a GET statement only in that a data item may be represented by an expression other than its name, for example, an arithmetic expression whose value is the item to be written. Once evaluated, the value represented by an expression is transmitted in the same way that the value represented by a variable is transmitted. Items in the data list (including expressions, if any) are separated by commas. An example of a list-directed PUT statement follows:

```
PUT FILE (OUT)
  LIST (NAME,6.3*RATE,NUMBER-10);
```

The PUT statement in the above example causes three data items to be written in the file named OUT. The sequence in which the data items are written follows the sequence of the items in the data list; that is, the first data item is the value represented by the variable NAME, the second is the value resulting from the evaluation of the expression 6.3*RATE, the third is the value resulting from the evaluation of the expression

NUMBER-10. Writing stops at this point.

Note that in list-directed input/output or in any form of stream-oriented transmission, it is the data list that determines the amount of data that is obtained from the stream or inserted into the stream.

Format of List-Directed Data

In list-directed input, successive data items on the external medium must be separated either by commas or blanks. On output, blanks are supplied between items automatically.

List-Directed Data Representation

The internal and external representation of a data item in list-directed transmission is determined by the attributes declared for it by the programmer. For example, a data item for which the attributes CHARACTER (10) have been declared would be recorded internally as a character string of length 10. On output, it would be written the same way. To better understand how this applies to list-directed GET and PUT statements, assume that the standard input file contains the following data:

```
'NEW YORK', 'JANUARY', -6.5, 72.6
```

Assume, further, that the following two statements appear in the program:

```
DECLARE CITY CHARACTER (12), MONTH
  CHARACTER (9), MINTEM FIXED DECIMAL
  (4,2), MAXTEM FIXED DECIMAL (5,2);
GET LIST (CITY, MONTH, MINTEM,
  MAXTEM);
```

The GET statement would cause the data items to be assigned as follows:

1. CITY is assigned the character string NEW YORK, left adjusted and padded on the right with four blanks.
2. MONTH is assigned the character string JANUARY, left adjusted and padded on the right with two blanks.
3. MINTEM is assigned the value –06.50.
4. MAXTEM is assigned the value 072.60.

The character strings are padded on the right with blanks to conform with the declared length of the strings; quotation marks are not maintained internally. The decimal fixed-point numbers are aligned on the assumed decimal point, to conform with the declared precision. Consider the result of the following PUT statement:

```
PUT, LIST (CITY, MONTH, MAXTEM,
  MINTEM, 'RANGE:', MAXTEM-MINTEM);
```

The record would be printed in SYSPRINT:

```
NEW YORK JANUARY 72.6  -6.5  RANGE: 79.1
```

Note that if a character string is printed, the single quotation marks are not written, whether the string is specified as the value of a variable (CITY and MONTH) or is specified as a character constant ('RANGE:'). If a character string is written in a file that does not have the PRINT attribute, the enclosing quotation marks are *supplied, if necessary, and are written.*[1]

Data-Directed Data Transmission

The elementary forms of the GET and PUT statements in data-directed transmission are written as follows:

```
GET FILE (file name) DATA;
PUT FILE (file name) DATA (data list);
```

The data list need not appear in the GET statement because data in the stream must be in the form of a series of assignment statements that includes each variable name and the value to be assigned to it.

The data in the input stream might look like this:

```
A=7.3  B='ABCDE'  C(4,2)=9876;
```

The variables A, B, and C must be known within the block in which a GET statement obtains these data items; they would have been declared in the block (or in a containing block). The effect is the same as if there were a data list specifying A, B, and C. Note that the last data item is followed by a semi-colon. It is a character that must appear in the stream to delimit the number of data items to be obtained by a single GET statement.

On output, the data list must appear to specify which data items are to be written into the stream. The PUT statement, referring to the data items could be:

```
PUT FILE (OUT) DATA (A,B,C(4,2));
```

On input, the assignments can be separated by commas or by blanks. On output, blanks are supplied and the semi-colon is written after the last item specified in the data list.

In data-directed transmission, data items on input or output are of the same form as specified for list-directed transmission.

Edit-Directed Data Transmission

Edit-directed data transmission allows a programmer to specify the format of data as it appears in the stream to be read and how it will appear when a data item is written.

The basic format of the GET and PUT statements for edit-directed transmission is as follows:

```
GET FILE (file name) EDIT (data list) (format list);
PUT FILE (file name) EDIT (data list) (format list);
```

If the file specification is omitted one of the standard files is assumed. The data specification consists of two parts, the data list and the format list. Each must be enclosed in parentheses. The data list is the same as that described for list-directed transmission, that is, a list of variables in a GET statement and a list of variables,

[1] When a bit string is written, the single quotation marks *do* appear, as does the letter B, even in a PRINT file. The binary digits are converted to the characters, 1 and 0.

constants, or other expressions in a PUT statement. The format list is a list of format descriptions, each of which describes the format of an individual data item. The first format item in the format list describes the format of the first data item in the data list, the second format item the second data item, etc.

Following is an example of an edit-directed GET statement.

```
GET FILE (INFILE) EDIT (LOAN_#,
    PRINCIPAL,RATE) (A(7), F(8,2),F(3,3));
```

Items, both in the data list and in the format list, are separated by commas. It is not necessary for the file specification to precede the data specification, but the data list must immediately precede the format list.

Matching the first format item with the first data name, the above statement specifies that the next seven characters in the data stream are to be assigned to the variable LOAN_#. The A format item implies a character string. In the procedure UPDATE however, LOAN_# was declared as a decimal fixed-point variable with a precision attribute (7). If the above statement appeared in UPDATE, the first seven characters would be converted to decimal fixed-point representation, upon assignment, to conform with the attributes of LOAN_# (if the first seven characters in the record contained any characters other than digits – and one decimal point and one plus or minus sign in its proper place – conversion could not be accomplished, and it would be in error).

The second format item and the second data name specify that the next eight characters are to be assigned to the variable PRINCIPAL. The F(8,2) format item implies that the characters are digits; it further implies that no actual decimal point appears among the eight characters, but that a point is assumed between the sixth and seventh characters. The external character representation of the data item is converted to internal-fixed-point decimal representation and assigned to PRINCIPAL.

The third format item and the third data name specify that the next three characters are to be converted to decimal fixed-point representation with an assumed decimal point to the left of the first character, and the item is to be assigned to RATE.

If a plus or minus sign or a decimal point appears in the characters in the input stream, it is included in the number of characters read. For example, –28.32 would require a format specification of A(6) or F(6,2), for the entire number to be read. The format specification, F(4,2), would result in the reading of only the first four characters (–28.). The value would not, however, be changed to –.28, even though the format item specifies the point is to be to the left of the last two digits. An actual point in the data that is read overrides the location of an assumed point as specified in the format specification; upon assignment to a fixed-point variable, the actual point would not be stored, but the data item would be aligned so that the location of the point agreed with the location of the point as declared for the variable.

If a format list is shorter than the associated data list, that is, if there are more data items than format items in a GET or PUT statement, the format list is re-used. For example, if only two format items were listed with four data names, the first format item would be associated with the first data item, the second with the second; then the *first* format item would be re-used with the *third* data item, and the *second* format item would be re-used with the *fourth* data item. A single format item could apply to all the items in a data list.

Edit-Directed Data Representation

Edit-directed data transmission allows great saving of space on the external medium used for input. Since each format item specifies the number of characters that make up each data item, there need be no separating commas or blanks. No decimal point need appear, because the format item specifies the location of the assumed decimal point. Quotation marks are not needed to identify strings.

The examples below show how input data might be recorded on a card or on tape, first to be used for list-directed input, then for edit-directed input. Assume the variables to which the data is to be assigned have been declared as in UPDATE:

```
LOAN_# FIXED DECIMAL (7), PRINCIPAL FIXED
  DECIMAL (8,2), RATE FIXED DECIMAL (3,3)
```

For list-directed input, the data and the GET statement would be:

```
8212349,24000.00,.055
GET LIST (LOAN_#, PRINCIPAL, RATE);
```

For edit-directed input:

```
821234902400000055
GET EDIT (LOAN_#, PRINCIPAL, RATE) (F(7),
  (F(8,2), F(3,3);
```

Format Items

There are two types of format items, data format items and control format items. Data format items, such as the ones in the preceding example, describe data items in the data stream. Control format items specify positioning within the stream or on the printed page. In the following discussion, all but the control and printing format items are data format items.

Fixed-Point Format Item: As discussed previously, a fixed-point format item specifies the appearance of a decimal fixed-point data item in the data stream. Its form is F(w,d); w represents the width of the field, or the total number of characters, including sign and point;

and d represents the number of digits to the right of the decimal point. If d is not specified, the point is assumed to be to the right of the rightmost digit.

On output, an actual decimal point is inserted, trailing zeros are supplied, if necessary, and a minus sign is inserted if the value of the data item is less than zero.

The F format item also is used to write fixed-point binary data, which is converted, on output, to decimal notation.

Floating-Point Format Item: The format item to specify the appearance of a decimal floating-point data item takes the form E(w,d). The letter w represents the width of the field, and d represents the number of digits to the right of the point. The field-width specification (w) represents the total number of characters, including decimal point, signs, and the letter E; consequently d must always be specified, since the point always will be somewhere to the left of E.

On output, a decimal point is inserted, blanks are inserted to the left if the actual number of characters is less than w. A minus sign is supplied for the *exponent* if the implied location of the point is to the left of its actual location. A minus sign is inserted to the left of the *first character* if the value of the data item is less than zero.

The E format item is used on output for either decimal or binary floating-point data. Binary data is always converted to decimal notation.

Character-String Format Item: Character strings in the data stream may be described with the A(w) format item. The letter w represents the number of characters in the string. It is always required on input. For output, if w is omitted, the length is taken as the length of the specified string.

Quotation marks should not appear in the input stream because a single quotation mark would be considered to be a single character. Quotation marks are not written on output.

Bit-String Format Item: Bit strings in the data stream may be described with the B(w) format item. The letter w represents the number of bits in the string. It is always required on input. On output, if w is omitted, the length is taken as the length of the specified bit string. Neither quotation marks nor the letter B should appear in the input stream. They are not written on output.

Picture Format Item: The picture format item, on input, is used to describe the type of characters in the stream. On output, it is used to edit the data item and to specify certain characters to be inserted. The form is P 'picture specification'. The picture specification is a string of *picture characters*. It always is enclosed in single quotation marks. For example:

P '999V99'	indicates a field of any five decimal numeric characters with an implied point between the third and fourth characters.
P'AAA9999'	indicates a field of any three alphabetic characters followed by any four decimal characters.
P 'XXXXXX'	indicates a field of any six characters, either alphabetic, numeric, or special.

Consider the following PUT statements:

```
PUT FILE (OUTFIL) EDIT (PAYMENT)
  (P '$ZZ,ZZ9.99');
PUT FILE (OUTFIL) EDIT (PAYMENT)
  (P '$$$,$$9.99');
```

Assume that the first column in the example below represents various values for PAYMENT when the above statements are executed. The second column represents the way they would be written upon execution of the first PUT statement; the third column, the way they would be written upon execution of the second PUT statement.

```
00002532      $    25.32        $25.32
3844267     $38,422.67     $38,422.67
0000001       $     0.01         $0.01
```

Picture characters may also be used to specify internal characteristics of data with the PICTURE attribute. For details about the PICTURE attribute and all of the picture characters that may be used in connection with it and with the picture format item, see the publication *PL/I: Language Specifications.*

Spacing Format Item: The spacing format item X(w) is a control format item that specifies relative horizontal spacing. On input, it specifies the number of characters (w) to be ignored. On output, it specifies that w blanks are to be inserted into the data stream.

For example:

```
GET EDIT (LOAN_#,RATE) (A(7), X(8),F(3,3));
```

This statement specifies that the first seven characters from SYSIN are to be assigned to the character-string variable LOAN_#, the next eight characters are to be ignored, and the following three characters are to be converted to decimal fixed-point notation and assigned to RATE.

```
PUT EDIT (LOAN_#,RATE) (A(7), X(8),F(3,3));
```

This statement specifies that the values of LOAN_# and RATE are to be printed in SYSPRINT. The value of LOAN_# is to be written as a character string of length seven, and eight blanks are to be inserted before the value of RATE is written as a three-digit number, with a decimal point inserted before the first digit.

Printing Format Items: The printing format items are used only with files that have the PRINT attribute. They are PAGE, SKIP (w), LINE (w), and COLUMN (w).

The PAGE format item specifies that the next output is to be written on a new page.

The SKIP (w) format item specifies that w—1 lines are to be skipped and the next data item is to be written on the *w*th line. If w is omitted, it indicates that the next data item is to be written on the next line.

The LINE (w) format item specifies that lines are to be skipped so that the next data item will be written on the wth line of the current page.

The COLUMN (w) format item specifies that blanks are to be inserted so that the first character of the next data item will be the wth character of the current line.

Note that the SKIP format item, like the X format item, specifies relative positioning, while LINE and COLUMN specify absolute spacing.

Example:

```
PUT EDIT ('MONTHLY BANK LOAN REPORT')
  (PAGE,LINE(2),A(24));
PUT EDIT (LOAN_#,PRINCIPAL,INTEREST,
  PAYMENT,BALANCE)
    (SKIP (3),A(7),COLUMN (15),F(8,2)),
      COLUMN (35),F(3,3),COLUMN(45),F(6,2),
      COLUMN (65),F(8,2));
```

The first PUT statement specifies that the heading MONTHLY BANK LOAN REPORT is to be written on line two of a new page. The second statement specifies that three lines are to be skipped and the value of LOAN_# is to be written, beginning at the first character of the line; the value of PRINCIPAL, beginning at the 15th character; the value of INTEREST at the 35th, the value of PAYMENT at the 45th, and the value of BALANCE at the 65th.

The overall layout of a page of a PRINT file is controlled through the use of the PAGESIZE and LINESIZE options of the OPEN statement. For example:

```
OPEN FILE (OUTFILE) OUTPUT STREAM
  PRINT LINESIZE (120) PAGESIZE (50);
```

This statement opens the file OUTFILE as a print file. Lines on the page will be a maximum of 120 characters in length; depth of each page will be a maximum of 50 lines.

An attempt to print on a page after 50 lines have already been printed (or skipped) will raise the ENDPAGE condition and cause an interruption in the same way that the ENDFILE condition can cause an interruption when an attempt is made to read from a file that has reached the end of file. The standard system action for the ENDPAGE condition is to skip to a new page, but a programmer can establish his own action through use of an ON ENDPAGE statement.

Remote Format Item

A programmer often can simplify the writing of his program by use of the remote format item. It has the following form:

```
R(statement label)
```

The statement label is the label of a FORMAT statement written somewhere else in the block. Consider the following two statements, which could appear in UPDATE:

```
                .
                .
                .
        PUT EDIT (LOAN_#,PRINCIPAL,
          INTEREST,PAYMENT,BALANCE)
          (R (LIST_A));
                .
                .
                .
LIST_A: FORMAT (SKIP (3),A(7),COLUMN
          (15),F(8,2),COLUMN (35),F(3,3),
          COLUMN (45),F(6,2), COLUMN
          (65),F(8,2));
                .
                .
                .
```

The remote format item R (LIST_A) indicates that the format list stated in the FORMAT statement labeled LIST_A is to be used as the format list for the PUT statement. The effect is as if the PUT statement actually contained the format list referred to. The remote format item and the FORMAT statement are a convenience in programs in which the same format list is applicable to a number of different GET or PUT statements.

The String Option

One feature of the GET and PUT statements is concerned with *internal* data transmission, rather than input and output. In either statement, the FILE (file name) option can be replaced by the STRING (string name) option. When the string option is specified, the statement has nothing to do with a file. In a GET statement, it indicates that the designated string is to be considered as a stream of input characters; in a PUT statement, it indicates that the designated string is to be considered as the output stream.

Although the string option can be used with any of the three modes of stream-oriented transmission, it is most practical in association with a format list since individual items in the string need not be separated by commas or blanks.

Consider the following example:

```
GET STRING (RECORD) EDIT (NAME,PAY_NO,
  HOURS,RATE) (A(12),A(7),F(2),F(4,2));
```

This statement specifies the following:

- the character string RECORD, which is recorded in the *internal* storage area, is to be scanned
- the first 12 characters of the string are to be assigned to NAME
- the next 7 characters are to be assigned to PAY_NO
- the next 2 characters are to be converted to decimal fixed-point representation and assigned to HOURS
- the last 4 characters specified are to be converted to a fixed-point decimal number with two fractional digits and assigned to RATE
- the remaining characters of the string, if any, are to be ignored

The PUT statement with a string option is the reverse of a GET statement:

```
PUT STRING (RECORD) EDIT (NAME,
  PAY_NO, HOURS*RATE) (A(12),A(7),
  P '$$99.99');
```

This statement specifies the following:

the character value of NAME is to be assigned to the first 12 character positions of the string variable RECORD

the character value of PAY_NO is to be assigned to the next 7 character positions of RECORD

HOURS is to be multiplied by RATE, and the value of the product is to be converted to a character string and assigned to the next 7 character positions of RECORD (this substring comprises the following characters: a dollar sign or a blank, a dollar sign or a decimal digit, a decimal digit, a decimal digit, a decimal point followed by two decimal digits; any of the digits might be zero)

Record-Oriented Transmission

Record-oriented data transmission deals with data sets that are composed of a series of separate records. Each record is read or written as an entity, either into or from an addressable buffer or into or from a specified variable (usually a structure or an array).

The data transmission statements used in record-oriented transmission are READ, WRITE, REWRITE, and LOCATE. Only the READ and WRITE statements are used when the records are accessed in their physical sequence from input and output files and are transmitted directly to and from specified variables.

Consider the example:

```
DECLARE 1 PAYROLL,
            2 NAME,
                3 LAST CHARACTER (12),
                3 FIRST CHARACTER (8),
                3 MIDDLE CHARACTER (1),
            2 PAY_NO CHARACTER (5),
            2 RATE,
               (3 REGULAR,
                3 OVERTIME)
                    FIXED DECIMAL (3,2);
READ FILE(INFILE) INTO (PAYROLL);
```

The READ statement causes the record to be read directly into the structure PAYROLL. There is no conversion of data types to conform to the attributes declared for the names. The data in the record must exactly match the declaration of PAYROLL; that is, the first 12 characters (including any blanks necessary to extend the string to its declared length) must represent the last name, the next 8 characters the first name, etc. And the portion of the record that will be assigned to RATE must be the valid *internal* representation of fixed-point decimal numbers. Since there is no conversion, the data in the record could not be written in character form. A record of this sort must have been written by a previously executed program.

The following statements might also be a part of the same program:

```
DECLARE 1 PAY_RECORD,
            2 NAME,
                3 LAST CHARACTER (12),
                3 FIRST CHARACTER (8),
                3 MIDDLE CHARACTER (1),
            2 HOURS,
               (3 REGULAR,
                3 OVERTIME)
                    FIXED DECIMAL (2),
            2 PAY,
               (3 REGULAR,
                3 OVERTIME)
                    FIXED DECIMAL (5,2);
    GET FILE (TIME_CARD) LIST (PAY_RECORD.
      NAME, PAY_RECORD . HOURS);
TEST: IF PAYROLL . NAME = PAY_RECORD . NAME
    THEN DO;
         PAY = HOURS * RATE;
         WRITE FILE (WAGES) FROM
           (PAY_RECORD);
         END;
    ELSE DO;
         READ FILE (INFILE) INTO (PAYROLL);
         GO TO TEST;
         END;
```

As shown in the example, both record-oriented and stream-oriented statements may appear in the same procedure. Assume that the file TIME_CARD, specified in the GET statement, represents a data set of punched cards being read from a card reader. Each card has the employee's name and the hours worked. The GET statement would cause the data, punched in character form, to be converted to fixed decimal notation for the data assigned to HOURS. The WRITE statement, however, would write the record from PAY_RECORD into the file WAGES exactly as the data appears in internal storage, presumably for some other program, since the data in internal format could not be printed directly.

The files referred to in the READ and WRITE statements would have to be declared to have the attributes RECORD and UNBUFFERED (if neither RECORD nor STREAM is declared, STREAM is assumed; consequently the file used in the GET statement need not be explicitly declared to have the STREAM attribute). The RECORD attribute specifies that the file is to be used with record-oriented statements. The UNBUFFERED attribute specifies that the data need not go into a buffer, but may be assigned directly to the variable specified in the INTO clause of a READ statement or directly from the variable specified in the FROM clause of a WRITE statement. The files are assumed to be SEQUENTIAL, that is, files in which records are accessed in the order of their physical appearance. The opposite of a sequential file is a direct file, a file in which each record has an identifying key so that records may

be read or written in any order by specifying the proper key in the READ or WRITE statement. A direct file generally must be explicitly declared to have the DIRECT and the KEYED attributes. The KEYED attribute declares that each record has a key, and it specifies the number of characters in the key.

The example in Figure 14 further illustrates record-oriented data transmission. It is basically the same procedure UPDATE that is explained in Chapter 3.

Four file names are declared in the example, three of them explicitly, one implicitly.

1. The implicitly declared file is the standard file SYSPRINT; it is both declared and opened as a result of the PUT statement (line 27).

2. MASTER is declared (line 8) to be an update, buffered file. It is opened implicitly as a result of the READ statement (line 14), at which time the SEQUENTIAL and RECORD attributes are implicitly applied. An update file is one that is both read from and written into. In a sequential update file, each record that is read is rewritten into the file, with or without change, before another record is read.

```
1.        UPDATE: PROCEDURE;
2.                DECLARE 1 DETAIL CONTROLLED (N),
                             2 PAY_# CHARACTER (7),
                             2 PAYMENT DECIMAL FIXED (6,2),
3.                         1 LOAN_INFO CONTROLLED (M),
                             2 LOAN_# CHARACTER (7),
                             2 PRINCIPAL DECIMAL FIXED (8,2),
                             2 RATE DECIMAL FIXED (3,3),
4.                         1 STATEMENT CONTROLLED (L),
                             2 LOAN_INFO,
                                3 LOAN_# DECIMAL FIXED (7),
                                3 PRINCIPAL DECIMAL FIXED (8,2),
                                3 RATE DECIMAL FIXED (3,3),
                             2 CHARGE DECIMAL FIXED (5,2),
                             2 PAID DECIMAL FIXED (6,2),
                             2 NEW_BALANCE DECIMAL FIXED (8,2),
5.                         INTEREST DECIMAL FIXED (5,2),
6.                         BALANCE DECIMAL FIXED (8,2),
7.                         REFUND DECIMAL FIXED (6,2),
8.                         MASTER FILE UPDATE BUFFERED,
9.                         INPUT FILE INPUT,
10.                        OUTPUT FILE OUTPUT;
11.               OPEN FILE (INPUT) RECORD SEQUENTIAL BUFFERED;
12.               ON ENDFILE (INPUT) GO TO MASTER_FILE;
13. NEW_RECORD:   READ FILE (INPUT) SET (N);
14. MASTER_FILE:  READ FILE (MASTER) SET (M);
15.               INTEREST = PRINCIPAL * RATE / 12;
16.               IF LOAN_# ¬= PAY_#
17.                  THEN DO;
18.                       PRINCIPAL = PRINCIPAL + INTEREST;
19.                       REWRITE FILE (MASTER);
20.                       GO TO MASTER_FILE;
21.                       END;
22.               IF PAYMENT <= PRINCIPAL + INTEREST
23.                  THEN BALANCE = PRINCIPAL + INTEREST - PAYMENT;
24.                  ELSE DO;
25.                       BALANCE = 0;
26.                       REFUND = PAYMENT - PRINCIPAL + INTEREST;
27.                       PUT LIST (LOAN_#, 'REFUND:',REFUND);
28.                       END;
29.               LOCATE STATEMENT FILE (OUTPUT) SET (L);
30.               STATEMENT . LOAN_INFO = LOAN_INFO;
31.               CHARGE = INTEREST;
32.               PAID = PAYMENT;
33.               NEW_BALANCE = BALANCE;
34.               LOAN_INFO . PRINCIPAL = BALANCE;
35.               REWRITE FILE (MASTER);
36.               GO TO NEW_RECORD;
37.               END UPDATE;
```

Figure 14.

3. INPUT is declared as an input file in the DECLARE statement (line 9). When the OPEN statement (line 11) is executed, the additional attributes RECORD, SEQUENTIAL, and BUFFERED are explicitly declared.

4. OUTPUT is declared as an output file (line 10). It is implicitly opened as a result of the LOCATE statement (line 29), and the RECORD, SEQUENTIAL, and BUFFERED attributes are implicitly applied.

All four file names are assumed, by default, to be EXTERNAL names.

The appearance of the CONTROLLED attribute specification followed by a parenthesized identifier (lines 2, 3, and 4) contextually declares the associated structure name to be a *based variable* and the parenthesized identifier to be a *pointer variable.*

A based variable is a name used in record-oriented transmission to describe the attributes of a record in a buffer. A based variable always has the CONTROLLED storage class attribute; no storage is allocated for it automatically. A pointer variable, which has the AUTOMATIC storage class attribute by default, is used to point to the location of the buffer that the based variable describes.

For example, the declaration of DETAIL indicates that N will point to a buffer into which or from which records will be read or written, and that each record will consist of two data items with the same attributes as those declared for PAY and PAYMENT. When the READ statement in line 13 is executed, a record is read from INPUT into a buffer. The second portion of the statement, SET (N), specifies that the pointer variable N points to the beginning of the record (in effect, the value of N is the address of the first storage location of the buffer). It is as if the record were assigned directly to DETAIL; a reference to PAY_# becomes a reference to the first data item in the buffer, a reference to PAYMENT becomes a reference to the second data item in the buffer.

The effect of the declaration of the two other based variables LOAN_INFO and STATEMENT is the same. Each is used to describe data of a record as it appears in a buffer.

The second READ statement causes a record to be read from the update file MASTER. The pointer variable M is set. If no payment is made, the principal is compounded and the REWRITE statement (line 19) causes the record to be written back into MASTER. There is no need to specify a variable since the REWRITE statement refers to the buffer into which the record was read. If there is a payment, the REWRITE statement in line 35 rewrites the updated record in MASTER.

The LOCATE statement (line 29) does not immediately cause any transmission of data. As it is written here, it specifies that a buffer, as described by the based variable STATEMENT, is to be allocated and that L is to be set to point to the buffer. The assignment statements (lines 30-33) create a new record in that buffer from the data in the buffer described by LOAN_INFO, followed by the values of INTEREST, PAYMENT, and BALANCE. A LOCATE statement also specifies that data in the indicated buffer will be written automatically immediately before execution of the next LOCATE or WRITE statement that specifies the same file name or immediately before the specified file is closed.

The assignment statement (line 34):

```
LOAN_INFO . PRINCIPAL = BALANCE;
```

updates the record from MASTER, which is rewritten by the REWRITE statement (line 35).

In this version of UPDATE, records of repaid loans are not deleted from the file. Records cannot be deleted from an update file that is also a sequential file.

For a program of this sort, however, MASTER could be a direct file in which records are accessed in any order. Records can be deleted from or added to a direct update file.

Assume that MASTER is declared as follows:

```
MASTER FILE UPDATE BUFFERED DIRECT
  KEYED (7);
```

LOAN_INFO might be declared:

```
1 LOAN_INFO,
    2 PRINCIPAL DECIMAL FIXED (8,2),
    2 RATE DECIMAL FIXED (3,3),
```

The declaration of MASTER specifies that any reference to MASTER will include a key specification that is a string of seven characters. The pay number, as a character string, could be used as the key. The structure LOAN_INFO is not a based variable; direct files cannot be buffered.

The READ statement could be:

```
READ FILE (MASTER) INTO (LOAN_INFO)
  KEY (PAY_#);
```

After the record is read from INPUT, the PAY_# is used as a key to find the correct record in MASTER. In this case, there is no need for the test shown in line 16 in Figure 14; the two records must refer to the same loan.

NOTE: This would not allow for updating loan records for which no payment had been made. A different program could be written to handle that updating.

The REWRITE statement then would be written as follows:

```
REWRITE FILE (MASTER) FROM
  (LOAN_INFO) KEY (PAY_#);
```

The deletion of repaid loan records could be:

```
IF BALANCE = 0
   THEN DELETE FILE (MASTER) KEY
     (PAY_#);
```

Redefining a Buffer

The use of based variables allows operations with data sets in which format of the records may vary. Consider the following simple example:

```
DECLARE 1 FORMAT_1 CONTROLLED
          (IN_IDENT),
            2 FORMAT CHARACTER (1),
            2 PAY_NUMBER CHARACTER (7),
            2 WAGES FIXED DECIMAL (5,2),
        1 FORMAT_2 CONTROLLED
          (IN_IDENT),
            2 FORMAT CHARACTER (1),
            2 PAY_NUMBER CHARACTER (7),
            2 WAGES,
                3 REGULAR
                  FIXED DECIMAL (5,2),
                3 OVERTIME
                  FIXED DECIMAL (5,2),
                        .
                        .
                        .
READ FILE (INPUT) SET (IN_IDENT);
IF FORMAT = 'A'
   THEN...
```

Assume that the first character of the record indicates the format of the record. If the character is A, it indicates the record matches FORMAT_1; if the character is not A, it indicates the record matches FORMAT_2. When the format of the record is determined, the corresponding based variable is used to refer to the contained data items.

Summary

The handling of input and output in PL/I can be done very simply if only minimum facilities are required. On the other hand, PL/I provides the facilities for the programmer to maintain careful and detailed control of all input and output operations.

This discussion has been primarily concerned with descriptions of specific examples. For complete details and specifications see *PL/I: Language Specifications*, Chapter 7, "Input/Output."

Chapter 12: Expressions and Operations

As discussed previously, any identifier other than a keyword that appears in a PL/I statement is an expression. A single variable is an expression, as is a single constant. Expressions can refer to arrays (array expressions); they can refer to structures (structure expressions); or they can refer to single items of data (called *scalar expressions*).

When one or more single expressions appear in conjunction with operators, the combination is an operational expression, and its type depends upon the nature of the operator. For example, an arithmetic expression is one that involves arithmetic operators.

An operator that precedes the variable or constant (the operand) is a *prefix operator* (as with −A or +A); an operator that appears between operands is an *infix operator* (as in A+B or A−B).

Arithmetic Operations

Most of the examples of operational expressions discussed so far in this book have been arithmetic expressions. The arithmetic operators, as noted before are:

\+ − * / **

Of these, only the plus and minus sign can appear as prefix operators.

If the operands of an arithmetic expression differ in base or scale, they are converted to a common base and scale before evaluation is made. Since the result conforms with the attributes of the variable to which assignment is made, there is no need to discuss here the manner in which different operands are converted. A detailed discussion is included in the publication *PL/I: Language Specifications.*

Data operated upon in an arithmetic expression must have an arithmetic value.

Comparison Operations

There are three kinds of comparison operations: arithmetic, character, and bit.

The comparison operators are:

< <= = ¬= >= >

None of the comparison operators can be a prefix operator.

Arithmetic comparison involves the comparison of signed arithmetic values.

Character comparison involves left-to-right, pair-by-pair comparison, according to collating sequence. For example A>B, B>C, etc. If the operands are of different lengths, the shorter is extended on the right with blanks.

Bit comparison involves left-to-right comparison of binary digits. If the operands are of different lengths, the shorter is extended on the right with binary zeros.

Concatenation Operations

Concatentation operations involve the chaining of characters, with no intervening blanks. The concatenation operator is ||, which is written as two "or" (|) symbols.

If the operands are bit-string, the result is a bit string. In all other cases, the result is a character string. If the operands are not bits or characters, they are converted to characters.

Examples:

If A is 7345 with an implied decimal point between the second and third digits, and if B is 8923, with an implied point before the first digit, A||B would result in the character string '73.45.8923'.

If C is '0111011'B and D is '111'B, the result of A||B is '0111011111'B.

If E is 'ABC' and F is 'DEF', the result of E||F is 'ABCDEF'.

Bit-String Operations

Bit-string operations involve the following Boolean logical operators:

¬ not
& and
| or

The "not" sign always is a prefix operator; the "and" and "or" signs always are infix operators.

Bit-string operations are performed on a bit-by-bit basis, from left to right. If operands are not bit strings, they are converted before the operation is performed; If the operands are of different bit-string length, the shorter will be extended on the right with binary zeros.

The following table shows the result of a bit-to-bit comparison under each possible circumstance:

A	B	NOT A	NOT B	A AND B	A OR B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Consider the following examples:

```
A = '1101'B; B = '1111'B; C = '010'B
        ¬ A = '0010'B
        A|C = '1101'B
        A&B = '1101'B
        A&C = '0100'B
      ¬ A&B = '0010'B
```

Order of Evaluation

Arithmetic expressions are evaluated according to the priority of the operator. Any expression enclosed in parentheses is evaluated before any other part of the expression.

Exponentiation (**), prefix + and prefix − have the highest priority. These operations will be completed first, and if more than one of these operators appears in the same expression, they are evaluated from right to left.

Multiplication (*) and division (/) have the second priority. They are evaluated from left to right.

Addition (+) and subtraction (−) have the lowest priority. They are evaluated from left to right.

If any other order is desired, parentheses must be used to indicate the order. For example:

```
A * B / C ** D  is not equivalent to
  (A * B / C)** D
A * B / C       is equivalent to
  (A * B) / C
A / B + C       is not equivalent to
  A / (B + C)
```

Bit-string operations, like arithmetic operations, are evaluated according to the priority of the operator. The "not" sign (¬) has the highest priority; the "or" sign (|) has the lowest priority. Any expression enclosed in parentheses is evaluated first. For example:

```
  B | C & D is equivalent to B | (C & D)
¬ B | C     is not equivalent to ¬ (B | C)
```

Chapter 13: Error Control and Program Checking

The IF statement has been shown as one way to check upon the execution of a program—to avoid errors and to make certain that the proper action is taken at the proper time. In UPDATE, discussed in Chapter 3, IF statements were included to avoid computation of a negative balance and to ensure that each payment would be applied to the proper loan record. Another IF statement was written to delete, from the new master file, the record of any repaid loans.

Control of many similar conditions of a general nature is supplied by PL/I. The ON ENDFILE statement in UPDATE is an example of one of these.

These conditions are situations that the computer has been *engineered* to recognize and note, or that the computer is *instructed* to recognize and note by coding automatically supplied as part of PL/I.

When one of these conditions arises, normal execution is halted at that point, and an *interruption* occurs. Control is then transferred to a predetermined group or block that instructs the computer what action to take. PL/I supplies a standard system action that is to be taken when any of the conditions arises. In many cases, this standard system action results in the printing of an error message and in complete termination of execution. In some cases, an error message is printed and execution continues from the point where execution was interrupted.

Whenever an interruption occurs, standard system action is taken unless the programmer provides an alternative action with an ON statement.

If an attempt is made to read from a file after the last record in that file already has been read, the ENDFILE condition arises. If an illegal data conversion is attempted on character-string data, the CONVERSION condition arises. If an assignment causes loss of high-order (leftmost) digits or bits, the SIZE condition arises.

These are but a few of the many conditions that can be checked. For all of these conditions except SIZE, PL/I provides constant monitoring to prevent unnoticed errors that would affect proper execution of the program.

The ON Statement

If a programmer specifies action to be taken when an interruption occurs for a specific condition, his specification always overrides the standard system action provided by PL/I.

```
ON ENDFILE (INPUT) GO TO MASTER_FILE;
```

This ON statement appears in UPDATE. Standard system action when any ENDFILE condition arises is to print an error message and then to terminate execution of the program. The ON statement in UPDATE overrides this action, and execution continues until completion of the procedure.

```
ON OVERFLOW GO TO ERROR;
```

The OVERFLOW condition can arise during floating-point calculations when the exponent of a computed floating-point data item exceeds the maximum size allowed, as defined for the particular compiler. ERROR is the label of a statement or the first of several statements that specify what action is to be taken, whether to try to recover from the error or to note the error and continue with other computations.

The ON statement is a compound statement that contains another statement. In the above case, GO TO ERROR is the contained statement, or the *on-unit*. An on-unit can be a single statement or a begin block:

```
ON FIXEDOVERFLOW BEGIN;
                   DECLARE (ERROR,
                     TEMP) FLOAT
                     DECIMAL;
                   TEMP = TABLE (I,J);
                   ERROR = TEMP * 5280;
                   PUT LIST
                     (TABLE(I, J), ERROR);
                   END;
```

The FIXEDOVERFLOW condition arises when a computed fixed-point data item exceeds the maximum precision allowed by the particular compiler. Assume that the programmer suspects that the FIXEDOVERFLOW condition might arise during evaluation of the decimal fixed-point expression, TABLE (I,J) * 5280. If it does, control is transferred to the on-unit of the ON FIXEDOVERFLOW statement. In the begin block, which is the on-unit, two temporary floating-point variables, ERROR and TEMP, are declared. The first assignment statement assigns to TEMP the value of TABLE (I,J) converted to floating-point scale. The second assignment statement specifies another evaluation of the same data that caused the interruption, but this time, the evaluation is made using floating-point data, with the floating-point result assigned to ERROR. Identification of the original data item, TABLE (I,J), is written as a fixed-point number, and the result of the computation is written as a floating-point number.

When an on-unit is a begin block, control returns from its END statement to the statement immediately following the point where the condition arose. Normal

execution continues from there. If the on-unit is a statement that transfers control to some other statement, execution will not automatically recommence following the point where the condition arose.

An on-unit can be a null statement:

```
ON FIXEDOVERFLOW;
```

In this case, if an interruption occurs due to the FIXEDOVERFLOW condition, no action is taken. Control is transferred to the on-unit, but since it is a null statement that specifies no action, execution begins again with the statement immediately following the evaluation that caused the condition to arise.

Scope of the ON Statement

The point of execution of an ON statement in a procedure determines the scope of its effectiveness. If a condition arises before execution of an ON statement that names that condition, standard system action is taken. After execution of an ON statement, its effect holds throughout that block; even if statements that physically precede the ON statement are reexecuted, the ON statement is still effective.

More than one ON statement for a specific condition can appear *internal to* a single block. A respecification also can appear in a *contained* block.

After an ON statement has been executed, its effect continues through all execution, even if control is transferred to another *external* procedure, until one of the following situations changes the effect:

1. Another ON statement for the same condition is executed.

2. Control is returned to a block in which another ON statement or standard system action is in effect.

3. A REVERT statement restores the effectiveness of another ON statement or standard system action.

A REVERT statement specifying a particular condition cancels the effectiveness of any ON statements for that condition that have previously been executed in the block to which the REVERT statement is internal. After a REVERT statement is executed, the action to be taken if an interruption occurs for the specified condition is the same as it was at the point of invocation of the block to which the REVERT statement is internal. A REVERT statement is ignored unless an ON statement, *internal to the same block*, has established an on-unit. Consider Figure 15.

In procedure A, standard system action will be taken if the FIXEDOVERFLOW condition arises before statement 2 is executed. After statement 2, the effective on-unit is CALL AERROR. The ON statement (2) continues effective until the ON statement (5) in procedure B is executed. When the BEGIN statement is reached, control passes into the BEGIN block. Statement 7 immediately establishes a new on-unit (CALL CERROR) until the REVERT statement reestablishes CALL BERROR as the effective on-unit. That on-unit remains effective throughout the external procedure D (the REVERT statement is not effective since no ON FIXEDOVERFLOW statement has previously been executed in D).

When the END D statement is executed, control returns to statement 10 in begin block C, which reestablishes CALL CERROR as the effective on-unit. But when END C is executed, control passes through it to the next statement in procedure B. The on-unit in statement 5 (CALL BERROR) is reestablished, and it continues effective until the END B statement is executed and control returns to procedure A where the first ON statement remains effective, and if the FIXEDOVERFLOW condition arises, the AERROR procedure is called.

```
1.      A: PROCEDURE;
                .
                .
                .
2.         ON FIXEDOVERFLOW CALL AERROR;
                .
                .
                .
3.         CALL B;
4.      B: PROCEDURE;
                .
                .
                .
5.         ON FIXEDOVERFLOW CALL BERROR;
                .
                .
                .
6.      C: BEGIN;
7.         ON FIXEDOVERFLOW CALL CERROR;
                .
                .
                .
8.         REVERT FIXEDOVERFLOW;
9.         CALL D;
10.        ON FIXEDOVERFLOW CALL CERROR;
                .
                .
                .
11.        END C;
                .
                .
                .
12.        END B;
                .
                .
                .
13.        END A;
14.     D: PROCEDURE;
                .
                .
                .
15.        REVERT FIXEDOVERFLOW;
                .
                .
                .
16.        END D;
```

Figure 15.

Condition Prefixes

An interruption for most error conditions of a general type will occur whether or not an ON statement has been executed. These conditions are said to be *enabled.* An ON statement specifying a particular condition merely determines the action to be taken when the condition arises; an ON statement has nothing to do with allowing or not allowing an interruption to occur when the condition does arise. PL/I, however, allows a programmer to control certain interruptions. He can disable some conditions that would normally cause interruptions.

This control is established through the use of *condition prefixes.* An enabling condition prefix is a condition name, enclosed in parentheses, and prefixed to a statement with a colon:

```
(SIZE): statement
```

A disabling condition prefix is the same as an enabling condition prefix, but the characters NO precede the condition name:

```
(NOFIXEDOVERFLOW): statement
```

Blanks are not allowed between the NO and the condition name.

There are only eight condition names that can appear in a prefix. They are: FIXEDOVERFLOW, CONVERSION, SIZE, OVERFLOW, UNDERFLOW, ZERODIVIDE, SUBSCRIPTRANGE, and CHECK (identifier list). The first four condition names have been described previously. The others are described in the following text.

UNDERFLOW. This condition arises when the computed exponent of a floating-point number is smaller than the permitted minimum, as defined for the particular compiler.

ZERODIVIDE. This condition arises when an attempt is made to divide by zero, in either a floating-point or fixed-point computation.

SUBSCRIPTRANGE. This condition arises when a subscripted name appears in a program and the value of a subscript is outside the bounds of that dimension of the array.

CHECK (identifier list). The condition arises when one of the identifiers of the identifier list is involved in a statement that is executed. The identifiers can be entry names, statement labels, or variable names. Standard system action is: an entry name is printed each time the block is invoked; a statement label is printed each time the statement is executed; a variable name and its current value are printed each time it is evaluated.

No condition names but these eight can appear in a condition prefix. All other conditions are always enabled and cannot be disabled.

Of the previously listed condition names, only SIZE, SUBSCRIPTRANGE, and CHECK (identifier list) must be enabled by the programmer. The condition is enabled if the condition name appears as a prefix to a statement. For example:

```
(SIZE): A = B * C;
```

If the product of B * C is greater than can be expressed with the precision declared for A, the SIZE condition will be raised. An interruption will occur, in this case, since the condition prefix has enabled the condition during execution of this statement.

A SIZE condition prefix can be prefixed to any statement. A CHECK (identifier list) prefix can be prefixed only to a PROCEDURE or BEGIN statement.

```
(CHECK (PROC_B, TAX, FICA)): PROC_A:
   PROCEDURE;
```

A condition prefix always precedes any statement labels that are prefixed to the same statement.

Scope of the Condition Prefix

The scope of a condition prefix depends upon the kind of statement to which it is prefixed. If the condition name is prefixed to any statement other than a PROCEDURE or BEGIN statement, the condition is enabled (or disabled) only through the evaluation and execution of that single statement. If it is prefixed to an IF statement, its scope is only through the evaluation of the expression in the IF clause; it affects neither the THEN clause nor the ELSE clause. If a condition name is prefixed to a DO statement, its scope is only through execution of the DO statement itself; the prefix of a DO statement has no effect upon any other statements of the DO group.

If the condition name is prefixed to a PROCEDURE or BEGIN statement, its scope is through the entire block, including all nested blocks except for any statements that lie within the scope of another condition prefix in which the same condition is specified differently.

Unlike the scope of an ON statement, the scope of a condition prefix does *not* extend to a block that is invoked remotely. A condition prefix to a CALL statement has no effect during execution of the procedure that the CALL statement invokes.

A condition prefix to a single statement overrides the scope of a prefix of the statement that heads a block. A condition prefix to the heading statement of any internal block overrides the scope of a prefix to the heading statement of an outer block.

If more than one condition name appears in the same condition prefix, the names must be separated by commas.

Consider the example shown in Figure 16. In statement 1, the condition prefix enables the SIZE condition (it is one of the conditions that is not enabled automatically; other conditions, OVERFLOW, for example, are automatically enabled). If either the SIZE condition or

```
1.                (SIZE): ALPHA:  PROCEDURE;
2.                                ON SIZE CALL AERROR;
3.                                ON OVERFLOW CALL OVERROR;
                                        .
                                        .
                                        .
4.                                CALL BETA;
                                        .
                                        .
                                        .
5.      (NOOVERFLOW, NOSIZE):     BETA; PROCEDURE;
                                              .
                                              .
                                              .
6.                                (SIZE): A = B||C;
                                              .
                                              .
7.                                            .
    (OVERFLOW, SIZE): GAMMA:  BEGIN;
8.                                ON SIZE CALL BERROR;
                                              .
                                              .
                                              .
9.                                            REVERT SIZE;
                                              .
                                              .
                                              .
10.                                           END GAMMA;
                                              .
                                              .
                                              .
11.                                       END BETA;
                                        .
                                        .
                                        .
12.                               CALL DELTA;
13.                               END ALPHA;
```

Figure 16.

the OVERFLOW condition arise during execution of statements in the procedure ALPHA, an interruption will occur. The ON statements (2 and 3) specify action to be taken if these conditions cause an interruption.

The prefix to statement 5 disables both the OVERFLOW condition and the SIZE condition (both normally would have been enabled, since the procedure BETA lies within the scope of the condition prefix in statement 1). The condition prefix in statement 6 overrides the prefix of the heading statement and enables an interruption for the SIZE condition during evaluation and execution of the assignment statement. Note that although the scope of the prefix of the ALPHA heading statement has been overridden, the scope of the ON statement (statement 2) is not affected by the change in the scope of the prefix for the same condition.

The prefix to the GAMMA: BEGIN statement reenables both conditions, cancelling the scope of the prefix of the BETA: PROCEDURE statement. Although the action to be taken changes if an interruption for SIZE occurs during execution of GAMMA, there will be an interruption if the condition arises; the change in scope of the ON statement does not affect the scope of the prefix.

When the END GAMMA statement is executed and control returns to BETA, the OVERFLOW and SIZE conditions are again disabled, to remain so until control returns to ALPHA.

When statement 12 is executed, the DELTA procedure is invoked. None of the effects of a condition prefix is transferred to DELTA. However, the effect of each ON statement (statements 2 and 3) continues into DELTA until another ON statement for each condition is executed.

Summary

A PL/I programmer has, with the ON statement and the condition prefix, two powerful facilities for program checking and for controlling errors that might occur during execution of a program.

The ON statement specifies action to be taken when an interruption occurs—to recover from an error or to continue even though an error has occurred. The condition prefix allows a programmer to decide when an

error would preclude successful completion of his program.

The scope of an ON statement and the scope of a condition prefix follow two different rules.

The scope of an ON statement continues through the *program* until another ON statement for the same condition is executed. The scope of a condition prefix to a heading statement continues through the *external procedure* until another condition prefix is effective.

In some cases (as in procedure BETA), the scope of an ON statement specifying action to be taken in case of an interruption can extend through blocks during which the occurrence of an interruption is precluded.

Appendix 1: The 48-Character Set

Except for certain restrictions, the characters that make up the 48-character set are the same as those that make up the 60-character set. These restrictions are given below.

The following characters are *not* included:

NAME	REPRESENTATION
Percent	%
Colon	:
Not	¬
Or	|
And	&
Greater Than	>
Less Than	<
Break Character	_
Semicolon	;
Number Sign	#
Commercial "At" Sign	@
Question Mark	?

The following three characters are replaced as indicated:

SIXTY-CHARACTER SET	FORTY-EIGHT-CHARACTER SET
:	. .
;	, .
%	//

NOTE: The two periods that replace the colon must be immediately preceded by a blank if the preceding character is a period.

The following operators, as used in the 60-character set, are replaced in the 48-character set by the indicated alphabetic operators:

SIXTY-CHARACTER SET	FORTY-EIGHT-CHARACTER SET
>	GT
>=	GE
¬=	NE
<=	LE
<	LT
¬	NOT
|	OR
&	AND
||	CAT

The above nine alphabetic operators are "reserved" in the 48-character set; that is, they must not be used as programmer-specified identifiers.

In each case, one or more blanks must immediately precede the alphabetic operator if the preceding character would otherwise be alphameric; also, one or more blanks must immediately follow if the following character would otherwise be alphameric. For example, to indicate the comparison of the variables A6 and BQ2Y for inequality, one would write A6 NE BQ2Y, but not A6NEBQ2Y, A6 NEBQ2Y, or A6NE BQ2Y. However, since the equal symbol is usable, the comparison of these two variables for equality may be written A6=BQ2Y.

The break character, commercial "at" sign, and number sign are not used in the 48-character set and consequently may not be employed in identifiers.

Appendix 2: Permissible Keyword Abbreviations

In PL/I, certain keywords can be abbreviated. The abbreviations themselves are keywords and are recognized as synonymous in every respect with the full keywords. The following alphabetical list gives these keywords and their abbreviations:

KEYWORD	ABBREVIATION
AUTOMATIC	AUTO
BINARY	BIN
CHARACTER	CHAR
CONTROLLED	CTL
CONVERSION	CONV
DECIMAL	DEC
DECLARE	DCL
EXTERNAL	EXT
FIXEDOVERFLOW	FOFL
INITIAL	INIT
INTERNAL	INT
OVERFLOW	OFL
PROCEDURE	PROC
SUBSCRIPTRANGE	SUBRG
UNDERFLOW	UFL
ZERODIVIDE	ZDIV

Index of Definitions

Index