File Number GENL-24
Form F28-8053-2

IBM **Systems Reference Library**

Common Business Oriented Language (COBOL) General Information

The COBOL language is designed primarily for commercial data processing. The name COBOL is derived from the words Common Business Oriented Language. The COBOL language was developed by the Conference of Data Systems Languages (CODASYL). CODASYL is a voluntary effort by a number of users and manufacturers of data processing systems.

The COBOL language is similar to English. Programmers with COBOL experience for one data processing system can quickly learn to write in COBOL for other systems. Programs written in COBOL for one data processing system are readily adaptable to any other system for which a COBOL processor is available.

This publication consists of three parts. The first part is a COBOL primer; the second and third parts are a detailed description of the COBOL language.

MINOR REVISION

This publication is a minor revision of the publication *COBOL*, Form F28-8053-1. It incorporates the changes published in Technical Newsletter N28-0019. No other changes have been made. Each change is marked by the symbol "•" next to the paragraph affected.

This edition is a reprint of the previous edition. The format has been changed from that of a General Information Manual to that of a Systems Reference Library publication.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Dept. D91, PO Box 390, Poughkeepsie, N. Y.

Preface

The technique of using electronic data processing systems to solve problems of many kinds has developed rapidly in recent years. Because of the great speeds made possible by electronic methods, computers have been of great value in solving problems which involve either a great many calculations or a large mass of data. Originally, they were used primarily in complex scientific analyses. In the last few years, however, they have been used increasingly to handle the great volumes of data which must be processed in the day-to-day operations of business. Beyond this, they have provided management with new techniques for analyzing and forecasting important developments in many areas.

The first electronic data processing systems could be used only by experts. Instructions to the computer had to be given in special codes—in what is generally called "machine language." However, it was soon learned that it is possible to write machine-language programs that enable a computer to recognize instructions written in other languages. This discovery led to the development of programming languages which were simpler to use than the machine languages.

The first such developments permitted the programmer to write convenient equivalents of machine instructions, using mnemonic symbols to represent them. The computer, acting under the control of previously written machine-language programs, would then "translate" these instructions into equivalent machine instructions, which could then be used in solving problems.

Later, programmers developed "macro-instructions"—that is, single instructions which could be used to produce a whole series of machine instructions. This development greatly increased the power of programming languages. Today, the art of programming has progressed to a point at which it is possible to give directions to a computer by writing statements and sentences in a language which is based on, and which can be read in the same way as, English itself.

The COBOL system is a result of efforts to produce an English-like programming language which can be used with many different types of data processing systems. The name itself is derived from the words "COmmon Business Oriented Language." Unlike the first programming languages, COBOL is "problem oriented." That is to say, the language itself, and the techniques for using it, are conceived in terms of the problems to be solved and the results to be obtained, not, for the most part, in terms of the technical features of the computer. Of course, each problem must still be solved by technical means; it is still necessary to produce a machine-language program before a problem can be solved. However, the language written by a COBOL programmer bears little resemblance to machine language, and the programmer has little direct concern with the method by which the COBOL-language program is translated into machine language.

The COBOL language was developed through the joint efforts of computer manufacturers and users, in cooperation with the United States Department of Defense. The language was first described in a report to the Conference on Data Systems Languages (CODASYL), issued by the United States Government Printing Office in April, 1960. Further refinements have brought the language to its present stage, which is known as COBOL-1961. It is this language which is described in the present manual.

COBOL-1961 has been defined by CODASYL as consisting of two main portions, REQUIRED COBOL-1961 and ELECTIVE COBOL-1961, as follows:

"'REQUIRED COBOL-1961' consists of that group of features and options, within the

complete COBOL specifications for the year 1961, which have been designated as comprising the minimum subset of the total language which must be implemented (to the extent of hardware capability) by any implementor claiming a 'proper' COBOL-1961 compiler."

" 'ELECTIVE COBOL-1961' consists of those features and options, within the complete COBOL specifications for the year 1961, whose implementation has been designated as optional for the manufacturers for the year 1961. If an implementor chooses to include any of these features or options (either totally or partially) in his compiler for 1961, he is expected to implement them in accordance with the specifications, for the feature or option, which are given in the COBOL-1961 manual."

The COBOL language described in this manual consists of *all* of REQUIRED COBOL-1961 as so defined, together with a large portion of ELECTIVE COBOL-1961. This language is fully supported by IBM and will be available for most of its computer systems.

ACKNOWLEDGMENT

In accordance with the requirements of the official government manual describing COBOL-1961, the following extract from that manual is presented for the information and guidance of the user:

"This publication is based on the COBOL System developed in 1959 by a committee composed of government users and computer manufacturers. The organizations participating in the original development were:

Air Materiel Command, United States Air Force
Bureau of Standards, United States Department of Commerce
Burroughs Corporation
David Taylor Model Basin, Bureau of Ships, United States Navy
Electronic Data Processing Division, Minneapolis-Honeywell Regulator Company
International Business Machines Corporation
Radio Corporation of America
Sylvania Electric Products, Inc.
UNIVAC Division of Sperry Rand Corporation

"In addition to the organizations listed above, the following other organizations participated in the work of the Maintenance Group:

Allstate Insurance Company
The Bendix Corporation, Computer Division
Control Data Corporation
E. I. du Pont de Nemours and Company
General Electric Company
General Motors Corporation
Lockheed Aircraft Corporation
The National Cash Register Company
Philco Corporation
Standard Oil Company (New Jersey)
United States Steel Corporation

"This COBOL-61 manual is the result of contributions made by all of the above-mentioned organizations. No warranty, expressed or implied, is made by any contributor or by the committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

"It is reasonable to expect that many improvements and additions will be made to COBOL. Every effort will be made to insure that improvements and corrections will be made in an orderly fashion, with due recognition of existing users' investments in programming. However, this protection can be positively assured only by individual implementors.

"Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures and the methods for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

Contents

Part I:
A COBOL Primer

Chapter 1: A COBOL Primer

This chapter is written for the reader who seeks a general understanding of the COBOL system. It tells him what this system is, and it will give him an idea of how the COBOL language can be used in solving commercial data processing problems. The discussion is general and informal. It does not define the rules for using the COBOL system or for writing the COBOL language. That is done in later chapters. Instead, it attempts to explain the basic concepts of the system by means of relatively simple examples.

If the reader is familiar with basic programming principles, he may wish to begin at once with the detailed specifications of the COBOL system. He will find these in Parts II and III of this manual. Part II is devoted to basic concepts, including the organization of a COBOL program, the elements of the language, the rules for combining these elements into meaningful instructions, and the means of preparing data for use in a COBOL program. Like Part I, Part II is written informally, since many readers will be encountering for the first time some of the concepts on which computer programming is based.

Part III contains the working rules of the COBOL system, stated in detailed, but more condensed, form. Part III is intended to be the main reference portion of the manual, the part to which the reader will most often refer after he has become familiar with the basic concepts of Part II. It specifies in detail the exact forms which the programmer must follow in writing instructions in the COBOL language, together with other necessary information.

The experienced programmer may find Chapter 1 of interest. However, it is written primarily for the reader who has had little, if any, experience with the programming of computers.

Business Problems and Machine Procedures

The electronic computer should be thought of as a tool which man may use to help him solve his problems. It has no powers man has not given to it, and it can solve no problems man himself cannot solve, given enough time and patience. It has, however, three important characteristics. The things it *can* do, it can do at phenomenal speeds. It can be instructed, *in advance*, as to how it is to solve problems to be presented to it later on. And it can be given *alternative courses* of action and can be left to "decide" which course to follow, depending on the circumstances. Once the computer has been properly instructed, it can be left to its own devices, so to speak, and it will carry out its instructions faithfully, accurately, and at incredible speeds. As the reader goes further into this chapter, he will begin to see that these three powers give the electronic data processing system a quite extraordinary capacity to solve problems of immense size and complexity.

What must a programmer know in order to use a computer? First of all, he must know what his problem consists of—he must be able to analyze it and reduce it to a series of steps that can be performed, one after the other. Second, he must know how to describe this sequence of steps in the form of instructions that the machine will understand. This manual is concerned with the second of these phases, and it will show how the COBOL language may be used to instruct a machine.

The COBOL language is actually a kind of shorthand way of giving directions to the computer. In most cases, a single COBOL statement represents a great many separate steps which the machine will have to carry out. But before we discuss the COBOL

language itself, we should consider some of the principles on which it is based.

The COBOL language is derived from English, and it looks like English. Thus, the programmer can work with it easily, without having to learn a long list of special symbols and codes, and the rules for using them. The following are typical COBOL-language sentences:

```
SUBTRACT DEDUCTIONS FROM GROSS GIVING NET.
PERFORM TAX-CALCULATION.
IF STOCK IS LESS THAN ORDER-POINT PERFORM
    REORDER-ROUTINE.
WRITE MONTHLY-STATEMENT.
```

These sentences are meaningful even to the casual reader. However, operations within a computer are controlled by instructions in code—in the internal language of the machine. In order for a computer to be able to interpret a COBOL sentence, the sentence must first be "translated" into the machine's language. As the COBOL system is designed, the COBOL-language program need be translated only once, and the resulting machine-language program can be used and reused indefinitely without further translation.

This translation is accomplished within the computer itself, employing many of the same techniques used in processing ordinary business data. A special program, known as a *processor* and supplied by IBM for the particular computer, is first entered into the machine. The COBOL-language program (often referred to as the *source program*) is then read into the machine, where the processor reads it and analyzes it. The computer acts on it in accordance with instructions built into the processor, and as a result of this process it creates a new program in machine language. This program is known as an *object program*. Once the object program has been produced, it may be used to process data whenever it is required. It may be used at once, or it may be recorded in some external medium and stored for future use. It may be used over and over again as long as it is needed.

A simple example will illustrate the basic principles.† Suppose we wish to increase the value of an item called INCOME by the value of an item called DIVIDENDS. The COBOL language allows us to specify this addition by writing the following sentence:

```
ADD DIVIDENDS TO INCOME.
```

Before the processor can interpret this sentence, however, it must be given certain information. For example, the programmer will have to write the names DIVIDENDS and INCOME in a special part of the program known as the Data Division. Here he will state certain facts about the data represented by those names, such as the maximum size of the individual items of data, the fact that the data consists of numerals, and so on. There are technical reasons for this, which we need not consider here. It is sufficient at this point to say that the Data Division is used to describe data so that the computer can recognize it, obtain it when needed, and know how to treat it in accordance with its special characteristics.

When the processor encounters the sentence ADD DIVIDENDS TO INCOME, therefore, it will have access to certain information that will aid it in translating the sentence. In addition, it will be able to obtain certain information "built into" the processor itself. Let us take this sample sentence and see some of the things that might

†For the convenience of the reader in following the various examples and formats of this manual, words which may be used directly in a COBOL program are printed entirely in capital letters. Some of these words are reserved for special use in the COBOL system. Others are typical of names which may be given by the programmer to some element of the program—such as an item of data—in accordance with rules given later in this manual.

happen to it. (The reader should note, however, that the exact procedure would vary from machine to machine, and that, in any case, the programmer is not directly concerned with the details.)

First, the processor will have to examine the word ADD. It will then consult a special list of words that have clearly defined meanings in the COBOL language. This list is part of the processor. As it happens, ADD is one of these words, and the processor will interpret it to mean that it must insert into the object program the machine instruction (or instructions) necessary to perform an addition.

The processor must also examine the word DIVIDENDS. Since it can obtain, from the Data Division, certain information about DIVIDENDS, it will know where and how DIVIDENDS information is to be stored in the computer, and it will insert into the object program the instructions the computer will need in order to locate and obtain the data.

When the processor encounters the word TO, it will again consult its list of special words. In this case, it will find that this word directs that it is the value of INCOME which is to be increased as a result of the addition. (It could have been DIVIDENDS, or some other value, if the sentence had been written differently.) Actually, it makes no difference to the processor in this case whether the programmer has written ADD DIVIDENDS TO INCOME or, simply, ADD DIVIDENDS INCOME. The former, of course, is more like conventional English, and this is the reason why the programmer is allowed to use TO in this sentence instead of omitting it. The meaning is clear to the processor in either case, even though the second form of the sentence might be ambiguous to a casual reader. The COBOL language contains a number of words which can be used in this way to improve the readability of COBOL sentences.

The processor must now examine the word INCOME. Again, it will have access to certain information about this word, and, as a result, it will be able to place in the object program the instructions necessary in locating and using INCOME data.

We have indicated that the programmer placed a period after the word INCOME, just as he would in terminating an English-language sentence. The effect of the period on the COBOL processor is quite similar. In this case, it tells the processor that it has reached the last word to which the verb ADD applies.

The steps we have described are performed by the processor in creating the object program. They might not be performed in exactly this way or in precisely the same sequence, since machines vary and since each processor must be adapted to a particular machine. However, regardless of the machine, the same COBOL-language sentence would produce machine instructions that will cause the object program to add together the values of DIVIDENDS and INCOME. Thus, the programmer can use the COBOL language to describe a procedure he wishes the computer to follow, and the computer, acting under the direction of its processor, will generate the program necessary to accomplish the desired result.

All of these steps are preparatory in nature, and they are required only in creating the object program used to process the actual data. They need be taken only once. Once the object program has been completed, the source program may be disregarded, and the object program is used for the actual processing. The source program is not required further, unless the programmer wishes to make a change in it; in that case, it must be reprocessed to create a new object program.

In the example we have been describing, the object program could perform the addition of the two specified items very simply. It would require very few machine instructions to do so—in some cases, only one. However, we have been talking about a very simple example. In actual practice, a single COBOL instruction may produce

dozens of machine instructions, and in many cases a few COBOL sentences may cause hundreds, or even thousands, of machine operations. In fact, when the programmer knows how to make the computer repeat a procedure as long as data is available to be worked on, he will see that with a few COBOL sentences he can start the computer on an operation that can go on indefinitely. It is not an exaggeration to say that a computer can be left to perform millions—even billions—of procedures rapidly and accurately and without supervision.

Perhaps the best way to understand how the COBOL system works is this: The COBOL language contains a basic list of key words and symbols. Each key word and symbol specifies to the processor a definite set of machine operations. In effect, the programmer thus has at his disposal a whole series of "prefabricated" portions of the object program he wishes the computer to construct. When he writes a COBOL-language program, he is actually directing the computer to bring together, in the proper sequence, the groups of machine instructions necessary to accomplish the desired result. The language in which he does this is not only easy to work with; it saves him from having to specify a great many machine steps in detail. The rules for writing the COBOL language are much simpler than those which govern the machine languages, and the programmer is enabled to write his programs easily, rapidly, and accurately. Using the processor appropriate to his particular machine, he can use English words and conventional arithmetic symbols to direct and control the complicated operations of the computer.

A Few Facts About Electronic Data Processing

If the reader has not had some previous exposure to the basic facts of electronic data processing, it will be worth while to discuss some of them briefly at this point. In particular, we should give some thought to some of the methods used for representing data within a computer and some of the types of operations the computer can perform on data.

The Language of the Computer

Information can be represented in many ways. When we write English, we use printed symbols. If we wish to transmit information by telegraph line, we must usually have it expressed in the form of special codes. Similarly, when we wish to communicate with a computer, we must convert our data into codes which the computer will understand. In most practical operations, this conversion is made automatically, and the programmer will rarely need to think in terms of the actual codes.

The COBOL programmer may first write his program on an ordinary sheet of paper, though he will usually find it more convenient to use a special printed form. (The reader will see examples of such a form later in this manual.) It would be possible to enter the program and the associated data directly into the data processing system by means of a keyboard much like that of a typewriter. However, the computer can accept data at extremely high rates of speed, and no human operator could possibly keep up with it. Thus, it is usually better to convert the data into codes which the computer can use efficiently *before* it is presented to the computer.

A convenient way of doing this is to punch the data into IBM cards, using a conventional card punch. This unit has a keyboard resembling that of a typewriter. When a key is pressed, a pattern of rectangular holes is punched in the card. There is a special pattern for each letter of the alphabet, each numeral, and each special character.

A further conversion is now required—into the special codes used within the computer. In many machine systems, the cards are fed into a special reading unit which makes the required conversion and stores the data in the form of magnetized spots

on one of various magnetic storage media. For example, codes may be stored on magnetic tapes or in a system of magnetic "cores" within the computer. Most high-speed computers use cores for main storage and tapes for external storage. Magnetic tape, which resembles that used on a tape recorder, has a special convenience: A reel of tape may be connected to the computer when it is required, and removed and stored when it is not needed. It has become the standard medium for entering data into the system, for recording data as output from the system, and for storing it when not in use. The usual procedure for entering data into a computer is to place it on a tape first, using a card-to-tape converter, and then to connect the tape to the computer so that its contents can be read. Preparation of the tape can be accomplished by other equipment while the computer is at work on other programs, and this practice saves using valuable machine time for the conversion.

The internal codes of the computer may be used not only to represent data, but to represent instructions to the machine as well. At any one time, the storage unit of the computer will usually contain both data and instructions. Its capacity to store and use both kinds of information, simultaneously and in volume, is an important factor in its ability to handle complex problems.

Basic Types of Machine Instructions

Every computer is designed to perform a basic repertory of operations. These operations may be specified by the use of machine codes reserved for the purpose. For example, one code may cause a simple addition, another may cause an item of data to be moved from one part of storage to another, while a third may cause a numeric value to be rounded.

Each such code, together with information needed to specify the data or the part of the machine affected, may be thought of as a "machine instruction." In this sense, a machine instruction describes the smallest element of procedure which a computer may be directed to perform, and the code or codes used to specify it are "built into" the machine.

A small computer may have a basic repertory of twenty or thirty such machine instructions, while a larger computer may have several hundred. In the latter case, the extra instructions represent refinements and additional features which simplify programming and greatly increase the speed and efficiency of the equipment. However, virtually all of these instructions can be related to just a few basic kinds of operations—movement of data, simple arithmetic, instructions that control the sequence in which operations are performed, and tests or comparisons of data that can be used for what might loosely be called "decision making." Programs of great power and complexity can be built up by combining instructions of a very few basic types.

In the first part of this chapter we said that the COBOL language is a kind of shorthand method of giving directions to a computer. We saw that a simple statement, such as ADD DIVIDENDS TO INCOME, may cause the machine to perform an extended sequence of machine operations. If the reader will review that example, he will see that those operations fall, in one way or another, into a few basic types.

For example, it was stated that the computer would have to examine a list of COBOL words to locate the word ADD. In many current machines, a search of this kind involves matching the word ADD successively against the words in the list. The machine begins by comparing the word ADD against the first word. If the words are not identical, it proceeds to the next. Until it finds a match, the computer goes through a series of comparisons or tests. A machine instruction may be required for each step and, if so, the program must provide for each comparison and must specify in each case what is to be done when a match is found. Thus, an extended sequence of machine operations is required to cover all the possible steps in just this one search.

So far, we have been talking about *comparisons* of data. We will see later that the ability to examine data in this way enables the computer to "make decisions" of many kinds. In the case of the search we have been describing, decision was limited to two possible courses of action—if the match was not found, the computer should make another comparison; if the match *was* found, the machine should replace the word ADD by the machine instruction (or instructions) equivalent to it.

The actual substitution of machine instructions for the word ADD would involve a *movement* of data, which is another basic kind of machine operation. When the processor creates the object program, it actually builds it up by obtaining the component parts (the machine instructions) and moving them into place in an area reserved for the purpose. It might be said that the COBOL-language program is a sort of blueprint, and that the processor studies it, marks off an area for building the object program, and then proceeds to locate the specified parts and move them into place, piece by piece.

In this particular example, we have been talking about comparisons and movements of data which are made in the course of creating the object program. However, the object program itself will frequently have to compare and move data in the same way. In the ADD statement we have been discussing, the actual values of DIVIDENDS and INCOME will be moved at *object time*—i.e., at the time the object program is run—before the addition can be performed.

In order to add two such values, one of them will usually have to be moved into a special location or register. Then the second value is added to the first. Speaking figuratively, it could be said that the second value is "moved in and superimposed on" the first value to create the sum. This sum may be left where it stands for further processing, but ultimately it will have to be moved elsewhere. This will usually require another machine instruction. Thus, it is clear, even a simple addition requires some kind of data movement. Some of this movement may occur automatically, but in other cases, data movement instructions must be inserted into the object program at the proper places. When the programmer uses the COBOL language to specify an arithmetic operation, the COBOL processor will provide these data movement instructions automatically.

Certain kinds of machine instructions occur constantly in computer operations, and the reader will find it useful to know something about them, even though the actual instructions are written for him by the processor. The major points are summarized in the brief discussion which follows.

Data Movement Instructions

Data movement instructions are used to transfer data from one place to another. The term is used here in a broad sense to include movements of data into and out of the computer system, as well as movements within storage. It is also used to include shifts of data. For example, numeric data is often shifted to the left or right in order to align decimal points correctly for calculation. It is often necessary to shorten, or to round, a number, and these operations may require one or more shifts.

Input and output instructions present special problems, and for this reason IBM provides an *input/output control system* (often referred to as an IOCS) for most of its computers. These systems facilitate the flow of data into and out of the computer. The COBOL processor for each machine is designed to take advantage of the input/output control system for the machine. Thus, the COBOL programmer can specify many of the important input/output operations directly in the COBOL language.

Arithmetic Instructions

A major function of an electronic data processing system is to perform calculations. Actually, most calculation can be reduced to a form of addition, although shifting

and comparing operations may be used to facilitate it. Thus, subtraction may be treated as negative addition, multiplication as a form of repeated addition, and division as a form of repeated subtraction.

These four basic arithmetic operations could be specified by writing the proper sequences of data movement, comparison, and addition instructions. However, most computers provide a single machine instruction for each basic type of arithmetic operation. Thus, a division can be specified in a single instruction; this saves the programmer from writing a long sequence of steps.

In addition to these basic operations, computers often provide machine instructions of a more sophisticated kind. An instruction for rounding a number will illustrate this point. Let us assume that a number is expressed within the computer by codes which correspond, digit for digit, to the numerals as we would write them on a sheet of paper. Suppose, further, that the number as stored in the computer consists of seven digits and that we wish to round it off to five. The programmer could accomplish this result by directing the computer to perform the following steps, or their equivalent:

1. Shift the number one place to the right.
2. Check to see whether the number is positive or negative.
3. Add a value of 5 to the right-hand digit if the number is positive, or subtract 5 if the number is negative.
4. Shift the number one more place to the right, dropping the digit to which the 5 had been added or subtracted.

The result of this procedure is that the fifth digit from the left will be increased by one if the sixth digit had previously contained a value of 5 or more.

This is not an exact description of the method by which rounding is actually accomplished in any particular machine. In any case, rounding can often be specified in one machine instruction. Thus it is clear that there are cases in which a single machine instruction can accomplish a result that might otherwise require a number of instructions.

Arithmetic instructions can in some cases be extremely complicated. On some computers, for example, it may be possible to use a single machine instruction to cause the calculation of a square root. The reader will recognize, of course, that execution of this one instruction would require many steps within the machine.

"Decision-Making" Instructions

We have seen that a computer can examine data to determine whether a particular condition exists. The following partial list will give the reader some idea of the kinds of conditions that can be tested:

An item can be examined to determine whether:

It is positive.
It is *not* positive. (This is not necessarily the same as being negative.)
It is negative.
It is *not* negative. (This is not necessarily the same as being positive.)
It is zero.
It is *not* zero.

Two items can be compared to determine whether:

They are equal.
They are *not* equal.
The first is greater than the second.
The first is less than the second.

An item may also be tested to determine whether it has certain other special characteristics. For instance, an item may be examined to determine whether the internal code used to represent it within the machine possesses a particular property. Such tests may be useful for such purposes as separating one class of data from another. In practice, the COBOL programmer will rarely have any direct concern with tests that depend on machine factors. However, a COBOL processor may make use of these factors in making a test, and in such cases the programmer will be able to take advantage of them indirectly. For example, the COBOL system permits the programmer to test an item to determine whether it consists of alphabetic characters. In the case of certain machines, such a test may be made easily by examining the codes for certain characteristics. The fact that the programmer may not know these codes is immaterial; he may still direct the processor to specify the test for him.

On some machines, a particular test may be specified in a single machine instruction. In other cases, it may not be possible to test for the particular condition by using just one instruction, but the same result can be obtained by using a succession of tests to determine that the alternative conditions are *not* present. In most cases, the COBOL processor will determine how to make a test of this kind, and the programmer will merely need to indicate the test he wishes made. However, as he will see, he still retains the power to specify a number of different ways of making tests. He will find this an advantage in meeting varying conditions.

How is a comparison or test used in the machine? The exact answer will vary with the machine, of course, but the method will be based in general on the following principles:

Normally, a computer will follow its instructions in the order in which they are stored within it. It advances, in a physical sense, from one storage location to the next, reading and carrying out the instructions in succession.

However, the programmer may specify that this sequence be interrupted at any point and that the computer take its next instruction from some other location. All machines provide "transfer" instructions for this purpose. The simplest of these may be called an "unconditional" transfer. Such an instruction directs the computer to skip—either forward or backward—to a specified location to find its next instruction. This feature makes it possible for the computer to repeat a procedure it has already performed or to skip over a procedure that is not wanted.

A transfer instruction can be specified in such a way that its execution depends on the results of a comparison or test. For example, it is possible to write an instruction which causes the computer to examine an item of data and then to transfer to a specified location if the item is found to be positive. If the item is *not* found to be positive, the computer will ignore the transfer instruction and will proceed to the next instruction in sequence.

Later in this chapter we will see how this "decision-making" power can be used to control data processing. It is an extremely important feature of the electronic computer. Since transfers can be made to virtually any location in storage, and since the point to which transfer is made may itself contain another test and/or transfer instruction, it is possible to set up a series of tests and transfers that will cause the computer to perform instructions in any sequence the programmer wishes to prescribe.

Matching the Machine and the Problem

It may seem to the reader that the machine has to do a great deal of work to solve a simple problem. But there are two points he should keep in mind. First, the machine can perform each individual step in a few millionths of a second, and it can finish a considerable number of calculations so rapidly that the stop light may

flash on before the operator can get his finger off the start key. Second, the steps performed by the machine parallel, more or less exactly, the steps that a human being would have to perform to do the same problem; a clerk may do many things by habit and instinct that the computer must be specifically instructed to do, but the fact remains that even the simplest clerical operation consists of a great many individual steps.

The relationship between machine instructions and the steps a clerk might take is fairly obvious in the case of the basic arithmetic instructions and those instructions which cause the machine to make decisions. Less obvious, perhaps, is the parallel between data movement instructions and clerical operations.

Let us see, however, how a clerk goes about performing a typical operation. Usually, he will have to locate one or more items on a business form of some kind. This may require reading labels, page numbers, column headings, and the like. Each such step requires obtaining the information in the first place, which is not unlike the concept of entering data into a machine. Each reading of a label, page number, and so on, actually involves a comparison of its "value" against some value the clerk has in mind. He will reject each value until he finds one that matches the one he is looking for. This process of reading, comparing, and rejecting will be repeated until the desired item is found. The clerk does all of this so rapidly, of course, that he is hardly aware that he does it at all.

Let us go a little further. When the clerk locates the item, he will want to use it according to some plan he has in mind—his "program," so to speak. As one of his first steps, he is likely to copy the item. Perhaps he will write it on a scratch pad or post it to another record. He may wish to perform a calculation on it, and this may require entering it on the keyboard of a desk calculator. All of these steps, of course, involve a kind of data movement, and many more will be required before he completes the job.

The operations of a computer closely parallel operations with which the programmer is already familiar, even though he may have been performing them unconsciously and automatically. This relationship provides him with a means of transferring a problem to the computer. In order to do so, he must analyze the problem, reduce it to its component steps, and find some way to express each step in terms the computer will accept. Since the COBOL language provides many short cuts for doing this, he will not have to think of all of the details, but he will write more efficient programs if he is aware of this fundamental relationship between the computer and the problems it solves.

Identifying Data for the Machine

We have seen that an electronic data processing system is capable of storing both data and instructions. The reader may wonder how it distinguishes between them. The answer is that it really doesn't. It has no way of knowing whether a particular series of codes represents an instruction or an item of data—at least, not by merely reading the codes. For this reason (among others), the programmer must identify the data for the computer, and he must also give the computer certain information about it. This is done in a part of the program known as the Data Division, and the information given in that division is usually referred to as *data description*. Description of data is one of the most important parts of programming, and the data description must be given to the processor before the object program is created.

One of the reasons for this—and it is the only one we need concern ourselves with at this point—is that each item of data must be assigned a specific location within the computer system. This is necessary because data can be identified within the system only by its location. If we were writing a program in machine language, we would have to know how to get each item into its proper place, and when we

wanted to obtain it, we would have to specify its location in each machine instruction that referred to it.

When we wrote the COBOL statement ADD DIVIDENDS TO INCOME, we were not only taking short cuts in specifying the machine operations; we were also using the names DIVIDENDS and INCOME as a short-cut way of identifying machine locations. The actual machine instructions, as produced by the COBOL processor, will direct the computer to add together the data found in two specific machine locations. If, by some chance, the data found there is *not* DIVIDENDS and INCOME, this fact will not matter to the computer. It will take whatever items it finds and will add them together anyhow. This means that it might try to add the letters of the alphabet, punctuation marks, and other characters. The results would vary, depending on the type of computer, but the programmer could not predict them, and they would be meaningless in any case. Furthermore, there is nothing to prevent the computer from trying to add two machine instructions. And even if DIVIDENDS and INCOME are found in their proper locations but, through an error in the data description, the decimal points are not properly aligned, the computer may add them just as it finds them, ignoring the decimal point. The resulting sum, of course, would be in error. The reader will thus recognize the importance of describing the data accurately.

When the processor reads the source program, it will note each kind of data to be used, together with its characteristics, and will assign it a location in storage. The programmer does not have to know what this location is. Instead, he gives each item a name, and when the object program is created, the processor will record both this name and the location in which the data will be stored. Later, when the processor reads the name of an item of data in a procedure statement, it will look up the name, read the *address*—i.e., the location—of the item, and insert that address into the appropriate machine instructions.

One important fact must be emphasized: A data-name always refers to a *kind* of data, not to a particular value. Thus, SALARY might be the name of a kind of data, while the individual value located there would vary for each employee. A data-name is like a column head in a ledger; it *identifies* the values in the column but does not *specify* them. When the programmer writes a data-name in a procedure statement, the computer will obtain whatever value happens to be located there at the time.

This capacity to refer to and identify data by name is one of the most useful and important features of the COBOL system. It saves the programmer from having to keep track of a great many storage locations. Equally important, the concept of naming makes it possible to assign names to procedure statements. Thus it becomes a simple matter to direct the computer to make a transfer from one procedure to another. For example, if a procedure for calculating a tax were called TAX-CALCULATION, it would be possible to cause the computer to perform it by writing such statements as GO TO TAX-CALCULATION or PERFORM TAX-CALCULATION. Some of the advantages this offers will be illustrated in the sample problem that occupies the remainder of this chapter.

Programming in the COBOL Language — A Sample Problem

The COBOL language provides a link between business English and the language of the computer. It is based on English, using English words and certain rules of syntax derived from English. However, since it is a computer language, it must be a very precise language. The programmer must therefore learn the rules which govern it and follow them exactly. The rules for doing this are specified in Parts II and III of this manual.

However, the reader may prefer to get a general idea of what the language looks like before he proceeds to the details. Accordingly, we will discuss a simplified inventory problem, showing how it may be expressed in the COBOL language.

Let us assume that we wish to write a procedure to record changes in the stocks of office furniture offered for sale by a manufacturer. We have said that one of the first things we must do is to identify and prepare the data to be used. In this case we will be working with such things as an item code to identify each type of product, the item name corresponding to it, the stock on hand, the unit price of each item, the value of the stock, and the order point used to initiate orders to replace depleted stock.

For the purposes of this problem we will consider only two aspects of data description: (1) We will inform the computer that we will work with two kinds of records, a master record, and a detail record, as shown below. (2) We will assign datanames to each of the items of data to be used.

First, we must organize the data into the two basic kinds of records. The first one we will call MASTER-RECORD and the second DETAIL-RECORD. The MASTER-RECORD will probably be derived from ledger records that look something like this:

Item Code	Item Name	Stock On Hand	Unit Price ($)	Stock Value ($)	Order Point
A10	2-drawer file cabinets	100	50	5,000	50
A11	3-drawer file cabinets	175	80	14,000	80
A12	4-drawer file cabinets	200	110	22,000	150
B10	Secretarial desks	150	200	30,000	120
B11	Salesmen's desks	50	175	8,750	50
B12	Executive desks	75	500	37,500	60
C10	Secretarial posture chairs	125	50	6,250	140
C11	Side chairs	50	40	2,000	60
C12	Executive swivel chairs	25	150	3,750	20

Figure 1-1.

Actually, there will be a MASTER-RECORD for each item in this list, but in describing the data for the computer, we will assume that all of these records will be of the same form—and we must be sure that they are. Thus we need specify the characteristics of only a single record. Accordingly, we will define MASTER-RECORD as consisting of data items having the following names:

ITEM-CODE
ITEM-NAME
STOCK-ON-HAND
UNIT-PRICE
STOCK-VALUE
ORDER-POINT

For the purposes of this chapter, it is not necessary to show just how this is done. We may note in passing, however, that when the actual data description is written, we will have to give the computer certain additional information. For example, we will mention that the data known as ITEM-CODE will consist of items three characters long, of which the first will be alphabetic and the second and third numeric. We will point out that STOCK-ON-HAND, UNIT-PRICE, STOCK-VALUE, and ORDER-POINT represent numeric values and that the data represented by ITEM-NAME may consist

of letters of the alphabet, numerals, and certain special characters. In general, the information required in the data description is fairly straightforward, and we do not need to concern ourselves here with the details.

MASTER-RECORD, of course, is the main record of current inventory. Changes to this record are made by entering the details of individual transactions or groups of transactions. Thus, receipts of new stock and shipments to customers will obviously change both STOCK-ON-HAND and STOCK-VALUE. These changes are summarized in the detail record for each item. A typical record might appear in a ledger in the following form:

Item Code	Item Name	Receipts	Shipments
B11	Salesmen's desks	25	15

Figure 1-2.

We will therefore designate this kind of record as DETAIL-RECORD and specify that it contains data items called:

ITEM-CODE
ITEM-NAME
RECEIPTS
SHIPMENTS

Again, full details will have to be written out in the data description, but we will assume that this has been done and that we are ready to tell the computer how to operate on our data.

Using COBOL to Perform Arithmetic

Figure 1-2 showed that changes had been made in the stock of salesmen's desks. Specifically, it showed receipts of 25 and shipments of 15, resulting in a net increase in stock of 10 desks. According to the record shown in Figure 1-1, the previous stock of this item was 50, its unit price was $175, and the stock value was $8,750. Now we must get the computer to bring this record up to date.

Let us assume for the time being that we have arranged to get these records into storage, ignoring the steps by which this was done. We will now reduce our problem to its very simplest elements and take it step by step.

We have already seen that the COBOL language contains the verb ADD. Thus, we may now add RECEIPTS to STOCK-ON-HAND by writing a simple ADD sentence, as follows:

```
ADD RECEIPTS TO STOCK-ON-HAND.
```

The computer will then find the value of RECEIPTS in the detail record and add it to the value of STOCK-ON-HAND in the MASTER-RECORD.

Now we must reduce this new value of STOCK-ON-HAND by the amount of SHIPMENTS. The COBOL language contains another verb which will accomplish this result, the verb SUBTRACT. Thus, we may write another simple sentence, such as:

```
SUBTRACT SHIPMENTS FROM STOCK-ON-HAND.
```

These two instructions, carried out in succession (or in reverse order, for that matter) will produce a current value of STOCK-ON-HAND.

Actually, there is a better way of doing this particular calculation. We have broken it into two steps, but the COBOL language provides another verb which permits us to specify more than one such step in a single sentence. This is the verb COMPUTE.

Consider the following sentence:

```
COMPUTE STOCK-ON-HAND = STOCK-ON-HAND +
    RECEIPTS - SHIPMENTS.
```

This sentence is interpreted as follows:

> "Compute the value of the expression at the right of the equal sign (i.e., STOCK-ON-HAND + RECEIPTS − SHIPMENTS) and change the value of the item at the left of the equal sign (i.e., STOCK-ON-HAND) to equal this value."

The name STOCK-ON-HAND occurs twice in this sentence, but this causes no difficulty. The expression at the right is calculated *first;* thus it is the current value of STOCK-ON-HAND which is used as the basis for computing the new value. When this new value has been calculated, it replaces the old value of STOCK-ON-HAND in the MASTER-RECORD.

A COMPUTE statement is always interpreted to mean that the value at the left of the equal sign will be changed to equal the value obtained by performing the calculation specified at the right of the equal sign. The reader will see that this one verb gives him a great deal of power to express calculations of many kinds.

So far, we have succeeded only in bringing the value of STOCK-ON-HAND up to date. But a change in this value will obviously cause a change in STOCK-VALUE as well. We will assume that this figure does not include allowances for quantity discounts, damage to stock, or other factors, and that the value of the stock is nothing more than the unit price of the item multiplied by the number of items in stock. As we might expect, the COBOL language provides us with a verb that will permit us to make this multiplication. Thus, we could compute STOCK-VALUE by writing the following sentence:

```
MULTIPLY STOCK-ON-HAND BY UNIT-PRICE GIVING STOCK-VALUE.
```

This is a simple way of doing the job. It is understood that the result of the calculation will be placed in the MASTER-RECORD as the new value of STOCK-VALUE.

But we have already seen that the COMPUTE verb allowed us to specify several additions and subtractions in one sentence. We will now see that it can be used to specify multiplication and division as well. Among the special characters of the COBOL language are four which represent the four basic arithmetic operations. These characters are as follows:

Character	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division

Thus, making use of the information we already have, we may now write this sentence:

```
COMPUTE STOCK-VALUE = UNIT-PRICE * (STOCK-ON-HAND +
    RECEIPTS - SHIPMENTS).
```

We may or may not wish to handle it this way. The STOCK-ON-HAND value, of course, is the original—not the updated—value, and if our sole purpose is to compute STOCK-VALUE (as it might be if we were preparing a special report), this method is very convenient. It does not provide, of course, for inserting the new value of STOCK-ON-HAND in the MASTER-RECORD, and if that is required, we would use one of the methods shown earlier.

The point, however, is that the COBOL language provides a variety of ways to accomplish the same result, and that some of them are extremely compact and powerful. It is up to the programmer to select the methods suited to his needs.

Before we leave the subject of calculating, we should note that in addition to the four basic arithmetic operations, the COBOL language allows the programmer to specify exponents in a formula. Thus, if we wish to express the square of a value called LENGTH (as in calculating an area), we could write it as LENGTH ** 2. The double asterisk means "exponentiated by," or, to put it less formally, "to the power of."

Directing the Computer to Make Decisions

We have already seen that the computer can examine data to determine whether or not some condition is present and that, depending on what it finds, it can be left to carry out an appropriate course of action. Our inventory example gives us an opportunity to see how this ability may be used.

The MASTER-RECORD, as we have defined it, contains an item called ORDER-POINT. Let us assume that an item is to be reordered when its stock has been reduced to or below its order point. Let us further assume that we have stored in the computer a procedure for initiating such an order and that we have given the name REORDER-ROUTINE to this procedure. We may now write the following two sentences:

```
IF STOCK-ON-HAND IS LESS THAN ORDER-POINT GO TO
    REORDER-ROUTINE.
IF STOCK-ON-HAND IS EQUAL TO ORDER-POINT GO TO
    REORDER-ROUTINE.
```

The effect of the first sentence will be as follows: The computer will compare the present value of STOCK-ON-HAND with the present value of ORDER-POINT. If STOCK-ON-HAND is the lesser value, the computer will transfer at once to REORDER-ROUTINE and begin the procedure for issuing the necessary order. If STOCK-ON-HAND is *not* less than ORDER-POINT, the computer will merely proceed to the next instruction.

Again it will be required to make a test, this time to see whether or not the present values of STOCK-ON-HAND and ORDER-POINT are equal. If they are, the computer will transfer immediately to REORDER-ROUTINE. If not, the computer will continue with the next instruction. We haven't indicated what that instruction will be, but presumably it will tell the computer what to do if STOCK-ON-HAND is greater than ORDER-POINT.

It may be useful to make these two tests separately, as shown above, and if the tests specify different transfers, each test must, of course, be specified individually. Unless there is some reason to keep them separate, however, we may simplify our instructions, just as we have used the verb COMPUTE to simplify calculation. Thus, we may write this sentence:

```
IF STOCK-ON-HAND IS LESS THAN ORDER-POINT OR EQUAL
    TO ORDER-POINT GO TO REORDER-ROUTINE.
```

Here we are using an *implied subject*. That is, STOCK-ON-HAND, which is the *subject* of the first condition, is understood to be the subject of the second condition as well. The rules for writing compound conditions, especially those using implied subjects, are quite specific and must be followed carefully. However, they do provide increased ability and flexibility in handling many kinds of problems.

The reader will surely have noted that in this example the computer is asked to test successively for two conditions out of a possible three. Unless he has some need to distinguish between these two conditions (as he might, if he were counting the number of times each situation occurred, or if he wanted to prescribe a different

action in each case), would it not be simpler to test for just the third condition instead?

This might, in fact, be the better way of making the test. The following sentence shows how it might be done:

```
IF STOCK-ON-HAND IS GREATER THAN ORDER-POINT NEXT
    SENTENCE OTHERWISE GO TO REORDER-ROUTINE.
```

The phrase NEXT SENTENCE is a special instruction built into the COBOL language. The computer will understand it to mean "go to the next sentence after this one."

While it should now be clear that the computer has the power to select a course of action in accordance with conditions it may find, the following examples may give the reader a better idea of the kinds of tests a computer is often called on to make:

```
IF GROSS-PAY IS LESS THAN 4800 GO TO FICA-CALCULATION.
IF ITEM-CODE IS ALPHABETIC PERFORM TYPE-A-FREIGHT-
    CALCULATION.
IF DISCOUNT-PERCENTAGE IS ZERO GO TO BILLING-
    ROUTINE.
IF HOURS-WORKED IS NEGATIVE GO TO ERROR-ROUTINE.
IF EMPLOYEE-NUMBER IS NOT NUMERIC PERFORM SPECIAL-
    CLASS-ROUTINE.
IF (A + B - C / D) ** E IS NOT GREATER THAN (F - G) / H +
    12.73 GO TO ROUTINE-F.
```

The actual rules for specifying tests and comparisons will be given in Chapter 3. It is enough for the present if the reader begins to realize some of the power inherent in the ability to pass on to the computer the task of making decisions.

Handling Exceptional Cases

Perhaps one small point should be noted here. In the fourth example above, reference is made to a procedure called ERROR-ROUTINE. Normally one would expect that HOURS-WORKED would always have a positive value. We would hope that if a negative value were discovered, the computer would recognize that something had gone wrong. Errors of this kind sometimes occur. For example, the data read in might be erroneous—perhaps the decimal point was in the wrong place and an otherwise normal subtraction produced a value below zero. Perhaps the programmer had not planned for all the possibilities that might develop when performing some previous part of the program. Perhaps a switch had somehow been set incorrectly. In any case, we assumed that if HOURS-WORKED turns out to have a negative value, a condition has arisen which requires special handling.

Good programmers learn to allow for certain kinds of errors, and very often they can write routines which will take care of them without stopping the machine. Suppose that in the example mentioned the value of HOURS-WORKED *did* turn out to be negative in a certain case, owing to an error in the original data. If the computer were to continue processing this record, it might calculate an unauthorized deduction from net pay, or make some other error. To prevent this, the programmer could anticipate the possibility of the error and write a special routine to be followed whenever the value of HOURS-WORKED is found to be negative. In the example mentioned, we have assumed that the programmer has written some such procedure and that he has called it ERROR-ROUTINE. Such a routine might cause the computer to stop processing the particular records, print out the data concerned so that it could be examined and corrected, and proceed automatically to the next records.

Such routines are used as double checks. When a program has been properly written and tested, few, if any, errors, are likely to occur which would either stop the

computer or allow inaccurate information to pass through without being marked for the operator's attention.

"Getting at the Data"

Earlier in this chapter, we saw that data processing requires a great deal of data movement. One of the most important kinds of data movement is input and output from and to the magnetic tapes, punched cards, or other media in which data is stored externally. The verb OPEN, as in the sentence OPEN INPUT INVENTORY-FILE, establishes communication between the computer and an external file of data. When the computer has no further need of the file, the verb CLOSE is used in a similar manner to sever this line of communication. While these verbs need not be discussed here, mention of them leads to a brief discussion of the two verbs used to obtain and release information after communication has been established with the file, the verbs READ and WRITE.

When we began to discuss our inventory problem, we assumed the data was already stored in the machine. Actually, to get it there, we would have had to write a sequence of steps like the following:

```
OPEN INPUT DETAIL-FILE, MASTER-FILE.
READ DETAIL-FILE RECORD.
READ MASTER-FILE RECORD.
```

As we have seen, input/output instructions actually involve movements of data. When the computer executes the READ statements given here, it will read the DETAIL-FILE and the MASTER-FILE and actually obtain one new DETAIL-RECORD and one new MASTER-RECORD; it will then move each record into a location in storage reserved for it.

Once the data is located in storage, the computer can perform the operations we have already specified. When these operations have been completed, we will wish to have the results written out in some form. In accordance with instructions given previously, MASTER-RECORD will contain new information after processing has been completed. In some cases it may be necessary to move this record to a location from which it can be transferred to some output medium, such as a magnetic tape, a printed form, or a series of punched cards. In order to make this transfer, we may use the verb MOVE. Consider the following sentence:

```
MOVE MASTER-RECORD TO UPDATED-MASTER-RECORD.
```

If we have properly defined a record called UPDATED-MASTER-RECORD, the entire MASTER-RECORD will be moved as a unit to UPDATED-MASTER-RECORD, where it will replace any data previously stored there. We may now have this information written out by means of such a sentence as WRITE UPDATED-MASTER-RECORD.

Here we should consider one of the special characteristics of electronic data processing. In most cases, when an item of data is "moved," what actually happens is that a copy of it is made in a new location. The old data remains intact where it was and can be used again. Thus, when we wrote READ MASTER-FILE RECORD, the computer copied one record into a reserved area in storage. The record as it originally appeared in its input medium (probably a magnetic tape) remained unaltered and could be used again—just as the tape for a tape recorder can be played over and over.

However, when an item of data is moved into a specified location, it usually will destroy any previous data stored there. We have gone to some trouble to calculate new values for our MASTER-RECORD, and if we wish to save it, we must move it out before a new MASTER-RECORD is read in. This is one of the reasons we wrote the statement MOVE MASTER-RECORD TO UPDATED-MASTER-RECORD. The record could have been written out directly, but in certain cases it may be better to move it first to an intermediate area, for reasons we need not discuss here. We have actually

created a new record, which may be used at a later date as a replacement for the old MASTER-RECORD. In the meantime, the original MASTER-RECORD remains intact on its original tape and can be retained as long as it is needed.

Making the Computer Repeat a Procedure—Cyclical Operations

Now let us proceed a step further. In our inventory example, there will probably be as many MASTER-RECORD and as many DETAIL-RECORD items to process as there are kinds of furniture kept in stock. It is customary to process all related records at one time before going on to another phase of the program. In this case, let us assume we want to repeat the processing cycle as long as there are records to process. To show one way of doing this, consider the following series of sentences:

```
NEXT-DETAIL-RECORD-ROUTINE. READ DETAIL-FILE RECORD.
NEXT-MASTER-RECORD-ROUTINE. READ MASTER-FILE RECORD.
    IF ITEM-CODE OF MASTER-RECORD = ITEM-CODE OF DETAIL-
    RECORD COMPUTE STOCK-VALUE = UNIT-PRICE * (STOCK-
    ON-HAND + RECEIPTS - SHIPMENTS)      MOVE MASTER-
    RECORD TO UPDATED-MASTER-RECORD    WRITE UPDATED-
    MASTER-RECORD    GO TO NEXT-DETAIL-RECORD-ROUTINE.
    IF ITEM-CODE OF MASTER-RECORD IS LESS THAN ITEM-CODE
    OF DETAIL-RECORD GO TO NEXT MASTER-RECORD-ROUTINE.
```

Figure 1-3.

The reader will recognize in this sequence a number of the statements we have previously examined. However, several new elements have been introduced.

We have previously noted that names can be given to procedures as well as to data. The above sequence contains two procedure-names, NEXT-DETAIL-RECORD-ROUTINE and NEXT-MASTER-RECORD-ROUTINE. The fact that these names extend to the left of the procedure statements has certain significance which will become apparent later in this book. For the moment, it is sufficient to note that when the program is printed, this device permits the reader to spot the procedure-names at a glance.

In the example above, the computer is required to compare two values: ITEM-CODE OF MASTER-RECORD and ITEM-CODE OF DETAIL-RECORD. If the reader will refer back to Figures 1-1 and 1-2, he will note that both the master and the detail records contained items which we called ITEM-CODE. Since identical names were used in our data description, we must have some means of distinguishing between them.

The naming system used in COBOL allows us to make this distinction by reference to the name of some larger group of data of which the item is part. Thus, the name ITEM-CODE OF MASTER-RECORD clearly identifies one of these items, while ITEM-CODE OF DETAIL-RECORD identifies the other. The use of an additional name (related to it by the word OF or IN) is called *name qualification.* Name qualification is often required in making distinctions between otherwise identical names.

In Figure 1-3, we are concerned with matching each detail record to the corresponding master record. This match is determined by comparing the two item codes. (We could have compared ITEM-NAME data, but this would take longer, since more characters would have to be compared.) The words IF ITEM-CODE OF MASTER-RECORD = ITEM-CODE OF DETAIL-RECORD will cause the computer to compare these values. If they are not equal, the computer will skip to the next sentence, which causes a test to see IF ITEM-CODE OF MASTER-RECORD IS LESS THAN ITEM-CODE OF DETAIL-RECORD.

In this example, we are assuming that both the master records and the detail records are arranged in ascending numerical order, and we are not allowing, at this point, for the possibility that any record is out of order. Neither are we pro-

viding a means of continuing to the next part of the program, or of stopping the computer when the last record has been processed. This, of course, would have to be done.

If the item codes are not equal, obviously the records do not correspond and no processing should be done. The next step is to get the next master record and see whether it is the one corresponding to the detail record. As the program specifies, the computer should transfer to NEXT-MASTER-RECORD-ROUTINE if ITEM-CODE OF MASTER-RECORD is found to be LESS THAN ITEM-CODE OF DETAIL-RECORD. When it makes this transfer, it will read the next MASTER-RECORD and it will then repeat the same tests all over again. If the new MASTER-RECORD still does not match the DETAIL-RECORD, the computer will continue bringing in new master records and testing them until a match is found.

When the records match, the computer will compute STOCK-VALUE, move the revised MASTER-RECORD to UPDATED-MASTER-RECORD, write out this record, and transfer to NEXT-DETAIL-RECORD-ROUTINE. It will continue to run through this cycle indefinitely, and additional instructions will be required to stop it after the last record in the file has been processed.

The ability to repeat a series of instructions in this manner greatly increases the power of the computer. In the example of Figure 1-3, we did not show how the computer would be directed to stop repeating a procedure and go on to a new phase of the program, and, of course, it would be essential to do so. The reader has seen how tests and comparisons can be used in making the computer repeat a procedure, and he can no doubt invent ways of his own to use such tests to terminate a repetitive process. Since the principles are based on those we have already considered, let us proceed instead and consider several other possibilities for controlling a repetitive procedure.

The verb PERFORM resembles the verb GO in a number of ways. Like GO, PERFORM specifies a transfer to the first sentence of a routine. Unlike GO, however, the PERFORM verb also provides that the computer will transfer back to the next statement following the PERFORM statement after it has completed the specified procedure. In addition, it provides various ways of determining how many times the procedure is to be performed.

A PERFORM statement may specify that a single sentence or paragraph be performed, or, if the desired procedure consists of more than one paragraph, it can specify two names which identify the beginning and the end of the procedure. To show a typical usage, let us rewrite the example of Figure 1-3 as follows:

```
NEXT-DETAIL-RECORD-ROUTINE. READ DETAIL-FILE RECORD.
NEXT-MASTER-RECORD-ROUTINE. READ MASTER-FILE RECORD.
    IF ITEM-CODE OF MASTER-RECORD = ITEM-CODE OF DETAIL-
    RECORD PERFORM STOCK-VALUE-CALCULATION MOVE
    MASTER-RECORD TO UPDATED-MASTER-RECORD WRITE
    UPDATED-MASTER-RECORD GO TO NEXT-DETAIL-RECORD-
    ROUTINE. IF ITEM-CODE OF MASTER-RECORD IS LESS THAN
    ITEM-CODE OF DETAIL-RECORD GO TO NEXT-MASTER-
    RECORD-ROUTINE.
                    .
                    .
                    .
STOCK-VALUE CALCULATION. COMPUTE STOCK-VALUE = UNIT-
    PRICE * (STOCK-ON-HAND + RECEIPTS - SHIPMENTS).
```

Figure 1-4.

After reading a MASTER-RECORD, the computer will test, as in the example of Figure 1-3, to see whether ITEM-CODE OF MASTER-RECORD matches ITEM-CODE OF DETAIL-RECORD. If it does, the computer will perform STOCK-VALUE-CALCULATION. This involves a transfer to the first instruction of that calculation, which in this case is the same calculation specified in Figure 1-3. However, when the calculation has been completed, the computer will transfer back to the main sequence of instructions, taking the next one in order, which in this case is the MOVE statement.

The PERFORM verb allows for a variety of ways of performing a given procedure. For example, the sentence containing the PERFORM verb may specify the exact number of times the routine is to be performed, or it may specify that the routine is to be performed until a certain condition exists. In fact, if no such information is supplied, the computer will interpret the command as if the number of times had been specified as 1.

To show how this might work, let us rewrite a portion of Figure 1-4. Instead of specifying the actual test to determine whether the records match, and then stating the required action, we could have written the test instructions as follows:

```
IF ITEM-CODE OF MASTER-RECORD IS LESS THAN ITEM-CODE
    OF DETAIL-RECORD PERFORM NEXT-MASTER-RECORD-
    ROUTINE UNTIL ITEM-CODE OF MASTER-RECORD =
    ITEM-CODE OF DETAIL-RECORD.
```

In fact, we could simplify this statement still further, thus:

```
PERFORM NEXT-MASTER-RECORD-ROUTINE UNTIL ITEM-
    CODE OF MASTER-RECORD = ITEM-CODE OF DETAIL-
    RECORD.
```

The phrase beginning with the word UNTIL specifies the condition which must be met in order to terminate the performance of this routine. The reader will find, in the discussion of the PERFORM verb in Chapter 7, that this is just one of several ways of controlling the number of times a routine is to be performed. Each way has its own special uses.

Similarly, the programmer may write GO and PERFORM statements to do the same job and yet have specific reasons for selecting one over the other. For example, it may be desirable to use the same procedure as parts of two entirely different sections of the program. In this case, PERFORM offers a convenient way of getting back to the point from which the transfer was made. Similarly, if there is a likelihood that there will be changes to the routine after the program has been written, it may be easier to change it if it is separate; if written separately, of course, a PERFORM statement will usually be required. On the other hand, if the programmer wishes to proceed directly to a portion of the program following the routine, a GO statement will often provide the best method of making the initial transfer.

Differences Among Computers

The COBOL language has been written in such a way that business problems may be expressed in it directly. That is, the structure of the language resembles that of English, and the kinds of operations that can be performed closely parallel those that are carried out every day in business offices. The COBOL language is said to be "problem oriented," whereas machine languages are necessarily "machine oriented." This is the reason why a processor is used to create a machine-language program from the initial COBOL-language program.

Nevertheless, in order to use a particular machine, the programmer must take account of its particular operating characteristics. For example, some machines store data in units of fixed length, usually called *machine words*, while in other

machines data may be stored in words of variable length. Where this difference occurs, there will be some differences in the way the data is handled. Similarly, in some machines, the internal codes are so designed that there is a particular code for each digit, letter, or special character of data, while in others a code system may be used in which there is no such correspondence. Again, this will result in some differences in the handling of the data.

To a very large extent, writing the procedure statements for a COBOL program requires no knowledge of machine characteristics. These characteristics do have some bearing, however, on the way the data is described in the Data Division, so that the data description may have to be written by someone who has some knowledge of the machine.

The Environment Division of a COBOL program, however, is the division in which the programmer relates his program to his particular machine. He will have to give such information as the amount of storage space available for data and instructions, the assignment of files to specific tape units, and the availability of printers, card readers, etc. For this reason, special publications will be provided to cover the environment description applicable to each computer. The reader may study the present manual, however, with relatively little thought of machine requirements. The material it contains has been arranged to stress the factors that are common to all COBOL programs, and the elements needed to adjust a program for a particular machine are identified. Detailed discussion of those elements, however, is left to the appropriate environment description publications.

In Chapter 2 the reader will find a general outline of this manual. Chapter 2 describes the organization of a COBOL program and guides the reader to those chapters in which the relevant material will be found.

Part II:
The COBOL Language — Components and Concepts

Chapter 2: The Organization of a COBOL Program

The COBOL system provides a convenient method of giving instructions to a computer. It consists of two main elements: the COBOL language, and, for each type of machine, a COBOL processor.

The COBOL *language* is the medium in which the programmer describes the operations he wishes the computer to perform. It is a language based on English. Its words are English words, and its "grammar" and punctuation are derived from English usage. A single COBOL statement may cause the computer to perform dozens of separate machine operations. Before a program can be run, each of these operations must be specified by an individual instruction in the internal language of the machine. Therefore, a program written in the COBOL language must be "translated" into the machine's language before any data can be processed. This translation is accomplished by means of a COBOL *processor.*

The COBOL language is capable of describing business problems of many kinds and of specifying the basic steps required to solve them. Procedures may be written in it with relatively little understanding of the detailed steps the computer will take in carrying out its assignment. The programmer will write more efficient programs if he has at least some understanding of machine operations, but the procedural portion of the language itself is, in large part, "machine independent."

Describing the Data

In order to write a workable program, the programmer must first analyze the problem he wishes the computer to solve. This means that he must analyze the data itself and the manner in which it is organized. This analysis is necessary because he will have to describe the data in such a way that the computer will be able to identify it. If the data is not in a form in which the computer can use it, he must arrange it in such a form before he can describe it.

Description of data includes writing down such information as the following:

- The name or names by which data is to be identified.
- The organization of each item of data with respect to other data—i.e., the scheme by which the individual items are grouped, and the relationships among the groups.
- The length of each kind of data.
- The location of the decimal point in numeric items.
- The "value" of constants—i.e., the actual values of the names, numbers, special characters, and so on which are to be stored in the computer for use in processing other data. Numbers in a table and names in a list are typical constants.

Each kind of data to be used in a program must be described in accordance with clearly defined rules. The portion of the program reserved for this purpose is known as the DATA DIVISION. The general concepts of data description are covered in Chapter 4 and the detailed rules in Chapter 6.

A substantial portion of the Data Division can be written in such a way that the same data description will be usable regardless of the type of computer on which the program is to be run. Certain portions of it, however, are related to specific machine characteristics, and therefore these portions must be rewritten should it become necessary to run the program on a different type of machine.

Analyzing the Problem

Having first described the data, the programmer must next determine the procedural steps required to solve his problem. This will mean writing instructions that cause the computer to perform such operations as the following:

Obtaining the data from some external source, such as a reel of magnetic tape or a deck of punched cards.

Obtaining the specific records which are to be processed.

Comparing key data whenever two or more records must be matched—for example, in making sure that the name on an invoice matches the name on the corresponding ledger account.

Examining data to determine which of several possible operations are to be performed.

Performing necessary calculations.

Assembling the data in the groupings in which it will appear as final output.

Issuing the data in final form, which may include printing it on a report, punching it into cards, recording it on magnetic tape, etc.

Repeating any or all of these operations selectively, using new data as required, until all desired records have been processed.

Stopping the computer when the job is done.

All of these steps, and other steps necessary to meet special situations, can be expressed by means of COBOL procedure statements. Most computer operations can be reduced to one or another of the following types:

Moving data into and out of the system and transferring it from place to place within the system.

Performing the four basic arithmetic operations: addition, subtraction, multiplication, and division.

Comparing two items of data to determine whether they are equal, or whether one is greater than the other.

Testing an item of data to determine whether it possesses certain characteristics, such as whether it is positive, negative, numeric, alphabetic, etc.

Altering the sequence in which the computer performs the instructions it has been given. Usually, such changes in sequence are carried out as a result of a comparison or test of data.

Accordingly, the heart of a COBOL program is the description of the operations to be performed. These operations are expressed in English-like sentences, and they are written in a part of the program called the PROCEDURE DIVISION. The rules governing procedure statements are covered in detail in Chapter 7 of this manual.

The procedure description of a COBOL program can usually be written without reference to machine characteristics. Thus, the Procedure Division of a program, if properly written, can be run on any computer which uses the COBOL system. In some cases, however, an expert programmer may wish to take advantage of special characteristics of his machine. In so doing, he may obtain maximum efficiency from his particular system, but at the same time he may find that his program will not run as smoothly on another computer. It is for the individual installation to decide whether it is more important for a program to make greater use of a machine's capacity or for it to be more readily usable on more than one kind of machine. The COBOL system is equally adaptable to either purpose.

Creating a Machine-Language Program

It has been pointed out that a COBOL-language program will have to be translated into a machine-language program before it can be used to process data. This translation is accomplished as follows:

For each IBM computer using the COBOL system, IBM will supply a COBOL processor. A processor is actually a special program which analyzes the words and characters of a COBOL-language program and creates a new program in the internal language of the machine. A single COBOL-language statement—sometimes even a single word or character—will often produce a great many machine instructions. It is the function of the processor to determine what these instructions should be and to combine them to form a new program. The processor must also take care of such supporting details as reserving space in storage for data and instructions and providing a means of identifying each item.

The processor is stored in the computer first. Then the *source program*—i.e., the COBOL-language program—is read in and the processor analyzes it and creates the *object program*—the machine-language program. The object program is prepared in a form suitable for the particular computer. It will be recorded on magnetic tape, punched in a deck of cards, or both. When the object program has been prepared, it may be read back into the computer at once, or it may be stored externally for future use. Before data can be processed, the object program must be read back into the computer's internal storage. The system is then ready to process data.

Since the object program is a machine-language program, the processor must take into account the operating characteristics of the machine and the capacity of the equipment available to it. Each installation will differ in some way from others of its type. For example, the storage capacity available for storing data and instructions internally may be greater in one installation than in another. Furthermore, the number of tape units available for external storage of data may vary, either as to the number of units actually installed or the number available for the particular job.

It is therefore necessary to furnish the processor with certain basic information about the equipment available to it. This is done by writing the proper COBOL-language statements in a portion of the program called the ENVIRONMENT DIVISION. The rules for doing so are set down in Chapter 8 of this manual.

The Environment Division also provides the programmer with the capacity to assign names to various units of the equipment so that these names can be used in the procedure statements. For example, if the programmer wishes certain information to be written out on a typewriter, he can write such a procedure statement as DISPLAY RESULT UPON TYPEWRITER, provided he has identified the typewriter by name in the environment description. This division also provides facilities for naming the conditions of switches for use in the procedure statements. Thus, such a phrase as IF INDICATOR-ON could be used in the Procedure Division to direct the computer in a course of action required when a switch called INDICATOR is on.

Distinguishing One Program from Another

The Data, Procedure, and Environment Divisions are the three main portions of a COBOL program. A fourth division, however, is provided so that the programmer can indicate certain information about the program itself, such as a name assigned to the program as a whole, the name of the programmer, the date of the program, and so on. This information may be specified in the portion of the program known as the IDENTIFICATION DIVISION, which is described in Chapter 9 of this manual.

A Guide to this Manual

When the source program has been written, it must be arranged in the following sequence:

IDENTIFICATION DIVISION
ENVIRONMENT DIVISION
DATA DIVISION
PROCEDURE DIVISION

However, it is felt that this is not the most convenient sequence in which to study the COBOL system. Accordingly, the discussion has been rearranged to aid the reader in learning how to organize and write a COBOL program. Following is a brief guide to this manual:

PART I: A COBOL PRIMER (Chapter 1)

PART II: THE COBOL LANGUAGE—COMPONENTS AND CONCEPTS

Chapter 2: *The Organization of a COBOL Program.*

Chapter 3: *The Structure of the Language.* This chapter describes the elements of the language—the basic character set, the rules for forming words, expressions, statements, sentences, paragraphs, and sections. It shows how arithmetic can be expressed in the COBOL language and states the rules for using the conditional expressions which give the computer the power to make decisions and select from a number of alternative procedures.

Chapter 4: *Concepts of Data Organization.* Chapter 4 discusses the fundamental principles of data description and gives definitions of many of the basic terms, such as file, record, and level. It also shows how the programmer may specify lists, tables, and constants for reference by the object program.

PART III: THE DIVISIONS OF A COBOL PROGRAM

Chapter 5: *Reference Format—The COBOL Program Sheet.* This chapter describes the format in which COBOL-language statements must be written in order to be accepted by the processor.

Chapter 6: *Data Division.* This chapter specifies in detail the rules which govern the writing of the data description.

Chapter 7: *Procedure Division.* Chapter 7 covers in detail the verbs used in the COBOL language to specify procedures to be carried out by the computer.

Chapter 8: *Environment Division.* This chapter provides certain basic information on how to specify details of the equipment to be used in running the object program.

Chapter 9: *Identification Division.* The forms to be used in identifying the program and in recording other useful information about it are specified in Chapter 9.

APPENDICES

Appendix A contains a summary of certain basic rules and other information, arranged in a form for quick reference.

Appendix B consists of short extracts from several sample programs which show how certain kinds of commercial problems might be described in the COBOL language.

Appendix C is a glossary of terms used in the COBOL system.

Notation Used in the Basic Formats of Verbs and Other Entries in this Manual

Throughout this manual, basic formats are prescribed for the various verbs, clauses, entries, and other essential elements of the COBOL language. These are generalized formats intended to guide the programmer in writing his own statements. The following rules of notation have been followed:

1. All words printed entirely in capital letters are COBOL words—i.e., words which have preassigned meanings in the COBOL system.
2. All underlined words are required unless the portion of the format containing them is itself optional—i.e., enclosed in square brackets. These are *key words,* and if any such word is missing or is incorrectly spelled, it is considered an error in the program.
3. All COBOL words *not* underlined may be included or omitted at the option of the programmer. These words are used only for the sake of readability. Misspelling, however, constitutes an error. These words are called *optional words.*
4. All italicized words represent information which must be supplied by the programmer. The nature of the information required is indicated in each case. In most instances, the programmer will be required to provide an appropriate data-name, procedure-name, literal, etc.
5. Material enclosed in square brackets [ ] may be used or omitted as required by the programmer.
6. When material is enclosed in braces { } , one, and only one, of the enclosed items is required; the others are to be omitted. The choice is to be determined by the programmer.
7. Punctuation, where shown, is essential. Other punctuation may be inserted by the programmer in accordance with the rules specified in this manual.
8. Special characters, such as the equal sign, are essential where shown, although they may not be underlined.
9. In certain cases, a succession of operands or other elements may be used in the same statement. In such a case, this possibility is indicated by the use of three dots following the item affected. The dots apply to the last *complete* element preceding them; thus, if a *group* of operands and key words are enclosed within brackets and the brackets are followed by three dots, the entire group must be repeated if any repetition is required, not merely the last operand.
10. Restrictions and comments on each basic format will be found in the accompanying text. The formats should not be used without proper study of the text.

Chapter 3: The Structure of the Language

This chapter will be devoted to explaining and defining the COBOL language by considering its basic elements and the ways in which they are combined. COBOL, like the English language, is built up from its smallest possible units, a set of characters (letters, numbers, punctuation marks, etc.). These characters are used to form meaningful words by following certain rules, just as English words are formed by following general rules of spelling. In English we find different types of words, such as nouns, verbs, conjunctions, etc. The types are combined, following the rules of grammar, to form expressions, statements, sentences, and paragraphs. The COBOL language also contains different types of words, and they are combined, using COBOL's rules of grammar, into statements, sentences, etc.

We will begin with the smallest parts of COBOL, the characters, and then build words, expressions and the larger units of the language. It is emphasized that the concepts, definitions and rules presented in this chapter must be thoroughly understood before proper COBOL programs can be written. In order to express an idea in COBOL, the programmer must use the language correctly and precisely.

Character Set

Each computer is constructed so that certain characters are meaningful to it. A set of such characters is referred to in this manual as a "computer character set." Because of the physical characteristics of each type of computer, the character sets for different types of computers may not be identical. Of course, all programs written in COBOL for a given machine must contain only characters from that machine's character set.

The COBOL *character set* given below is a set of characters which is common to all IBM computers on which COBOL programs may be run. The COBOL character set consists of the numerals 0 through 9, the 26 letters of the alphabet, and the special characters shown in the table below.

Special Characters Used in the COBOL Language

Name	Character†	Card Code
Space		(blank)
Plus Sign	+	12
Minus Sign Hyphen	−	11
Multiplication Sign Check Protection Symbol	*	11-4-8
Division Sign	/	0-1
Left Parenthesis	(	0-4-8
Right Parenthesis	)	12-4-8
Comma	,	0-3-8
Period Decimal Point	.	12-3-8
Dollar Sign	$	11-3-8
Equal Sign	=	3-8
Quotation Mark	'	4-8

†This group of special characters is one of several character sets available for IBM equipment. All sets use the same card codes, but one code may represent one character in one set and another character in another set. For example, a "12" punch indicates a plus sign in certain sets, while in other sets it represents an ampersand. The COBOL system employs the codes of Set H, shown above. The use of each of these characters will be explained later in this manual.

Names

Any language must contain words which stand as symbols for things. They are known as names or nouns. COBOL also employs names which the programmer uses to refer to things he is handling in his program. In ordinary language we do not usually have to distinguish consciously between the *name* of an object and the *object* itself. But this is very important in COBOL. A programmer must always keep in mind that when he gives an item of data a name and then refers to that item by writing its name in a program, he is referring to the data, *not to the name.* Thus, a name can be said to *represent* the value of the associated data item. A name may represent a data item that assumes many values during the course of a data processing job. If a payroll is being processed, the name HOURLY-RATE may represent the value of the wage rate of first one employee, and then of each succeeding employee as each paycheck is processed.

In other programming languages it is often said that a name is given to a location in storage, and that anything that occupies that location is referred to by the name given to the location. The concept that a name represents a value is a different way of saying exactly the same thing; however, when working with a language which is problem-oriented rather than machine-oriented, the latter concept is more convenient and more closely akin to problem-oriented terminologies. The way in which data is recorded internally varies widely from computer to computer, and so the idea of a name as representing a location may vary somewhat from machine to machine. In this light, the idea that a name represents a value is less restrictive.

There are four general categories of names in COBOL. They are *data-names, condition-names, procedure-names,* and *special-names.*

Data-Names

Data-names are names given to the data used in a program. As the reader will see, data-names usually will represent a number of values during the course of a program. For example, if a program is written to compute the payroll for a business firm, the programmer might name one item of data MAN-NUMBER. Then, as the payroll is processed, the data-name MAN-NUMBER refers to the man number of the man whose pay is currently being computed. In other words, the data-name MAN-NUMBER would represent a value equal to the man number of each man as his pay is being calculated. If RATE-OF-PAY is another data-name, it would represent the value of each man's rate of pay as his pay is being computed. The programmer, of course, controls the way in which a data item assumes different values. Data items assume different values as a result of reading data into the system, by using arithmetic expressions to compute new values, etc. This will be discussed later.

Data-names are devised and assigned to data following the rules given in this chapter which govern the formation of names. In general, all data referred to in a source program must be named.

Procedure-Names

Procedure-names are names assigned to individual portions of a program so that one procedure statement can refer to another. For example, suppose an item of data called SWITCH is to be tested to see if it is positive or negative; if it is positive, the program is to proceed normally, but if it is negative a special sequence of commands must be executed to compute a refund. This special sequence of commands (sometimes called a "routine") might be assigned the procedure-name REFUND-ROUTINE, and if the data is found to be negative, this routine might be entered by writing: IF SWITCH IS NEGATIVE GO TO REFUND-ROUTINE.

Like data-names, procedure-names are devised and assigned by the programmer as he needs them, following the rules for name formation given in this chapter. They are placed at the beginning of the portion of the program to which they apply.

A procedure-name must be assigned to each paragraph and section of the program. Further details on the use of procedure-names may be found later in this chapter, and in Chapters 5 and 7.

Condition-Names

The concept of condition-names will be more clearly understood after the reader has studied the discussion of conditional expressions later in this chapter. In general, however, a condition-name is a name which is assigned to denote one of a number of values which may be assumed by an item of data. It is often used when the procedure to be used in processing data depends on a "code" which is part of the data. For example, suppose a wholesaler is keeping an inventory of the number of rubber door mats he has in stock. These mats come in four colors, red, green, blue, and black. If each incoming or outgoing shipment were recorded on a punched card, there might be one column set aside to specify the color of the mats in the shipment. Red might be represented by punching a 1, green by a 2, blue by a 3, and black by a 4. If the inventory records were being maintained by a COBOL program, the programmer might create a data-name, COLOR. Then COLOR could assume the values 1, 2, 3, or 4, depending on the color of the door mats. Also, in the Data Division of the program, the condition-name RED would be associated with the value 1, GREEN with the value 2, BLUE with the value 3, and BLACK with the value 4. Then to test for the shipment of red door mats, we could write a statement beginning with the words IF RED. As a result, the data-name COLOR will be examined to see whether it has the value 1. If, for some reason, we wished to avoid using a condition-name, we could state the same test by writing IF COLOR = 1, which in this case is exactly equivalent to IF RED.

Condition-names are subject to the general rules for the formation of names, which are given in this chapter. They are assigned by the programmer at his discretion.

Special-Names

Special-names are names which may be assigned by the programmer to various physical parts of a computer. For example, one card punch unit might be named MASTER, while another punch unit might be named DETAIL. Then, any time the programmer refers to either device he uses the special-name he has assigned to the device, MASTER or DETAIL.

One purpose of assigning special-names is to enable one COBOL program to be run on different computers. For example, if a given program is to ACCEPT or DISPLAY information (see Chapter 7) by means of an input or output device, the device must be assigned a special-name by the programmer. Then, if the program is to be run on a different computer, with different types of input/output devices, the special-names are reassigned by the programmer, depending on the devices available on each machine.

Special-names are assigned in the Environment Division of a COBOL program, as explained in Chapter 8. Each type of computer has a different set of devices to which special-names may be assigned, and these will be listed in the publication covering the appropriate COBOL processor.

The rules for forming special-names are the same as those for writing data-names and are given below.

Rules for Forming Names

Names may be formed by combining any of the following characters: the letters A to Z, the numerals 0 to 9, and the hyphen. In addition, the following rules must be followed:

1. Names must not contain blanks.
2. They may contain from 1 to 30 characters.

3. They may neither begin nor end with a hyphen. However, hyphens may be used freely elsewhere in the name for the sake of readability.

4. Data-names, condition-names, and special-names must contain at least one alphabetic character. Procedure-names may consist exclusively of numerals if the programmer so desires.

5. Names may be "qualified" by the use of other names, as described below.

Qualification of Names

In many cases a program will contain "duplicate" names. This often happens when an input file is "updated" to produce an output file, since each file will usually contain the same kinds of records.

Suppose that an input record is named INPUT-MASTER and an output record is called OUTPUT-MASTER. Suppose, also, that each record contains the date of the last shipment of a certain type of goods and that in both records this date is called LAST-SHIPMENT-DATE. If the programmer writes MOVE LAST-SHIPMENT-DATE TO WORK-AREA, he is writing an ambiguous statement. Does he want LAST-SHIPMENT-DATE in OUTPUT-MASTER moved, or is he referring to LAST-SHIPMENT-DATE in INPUT-MASTER?

This problem can be easily resolved if he writes MOVE LAST-SHIPMENT-DATE IN INPUT-MASTER TO WORK-AREA. Now there is no doubt of the programmer's intent. This is an example of *qualification of names.*

If a name is not unique, it must be qualified by one or more additional names until it is unique. The words OF and IN are the key words which indicate qualification. One of them must appear between the name and the qualifier and one must also appear before each additional qualifier. Either OF or IN may be used, since they mean the same thing to the COBOL processor. They are known as *qualifying connectives.* Specific details of name qualification are discussed in Chapter 4.

Assigning Names in the Program

The reader has seen that the COBOL system uses names as a convenient – in fact, indispensable – means of identifying data, procedures, and conditions. It is now necessary to indicate how each name is placed in the program in a way that permits the processor to connect it with the item (or value) to which it refers.

Procedure-names differ from other names in one important respect. They are used as names for paragraphs and sections of the program, whereas data-names and condition-names represent the information being processed by the program.

Procedure-names are written in the Procedure Division immediately before the text associated with them, in accordance with the rules given in Chapter 5. Once the name has been written, any reference to the name is interpreted as a reference to the associated procedure.

Data-names and condition-names, however, require further discussion. The COBOL processor must know whether the data is numeric or whether it contains alphabetic or special characters. It must know where decimal points, if any, are to be placed, where to place dollar signs, and so on. There are a number of such details which must be specified.

Each data-name and each condition-name used in the Procedure Division must be properly accounted for in the Data Division, as explained in Chapter 4. Once this has been done, the programmer is free to refer to the name repeatedly throughout the Procedure Division.

Constants

Up to this point, it has been emphasized that a data item usually assumes one or more values during the running of a program. These values, we have said, might be entered into the computer as input data or could be created by computation during the course of the program.

However, the occasion often arises when a *fixed* value is used in processing. For example, suppose a program were written to handle the sales of a wholesale house. Assume that for certain kinds of items, a ten per cent tax must always be added. Since this value of ten per cent never varies, it would be convenient to be able to write it directly at the time the program is written, rather than having to enter it as data. A fixed value which never changes during the execution of a program is called a *constant.* A constant can be *numeric* or it can be *non-numeric,* i.e., alphanumeric. Generally, a constant may be anything that can be expressed by a combination of the characters in the given computer's character set. Thus, both of the following are constants:

```
.10
'THIS IS A NON-NUMERIC CONSTANT.'
```

A data-name which represents non-numeric information is said to represent a *non-numeric* value.

There are two general types of constants, *literals* and *named constants.*

Literals

One way to specify a constant is to enter the actual value of the constant at the point in the procedure at which it will be used. Such a constant is known as a *literal constant,* or simply as a *literal.* The programmer does not name a literal. The characters of a literal are written in the program at the point at which they will be used at object time. For example, the constant value ten per cent might be entered into a program as a literal in the following way:

```
COMPUTE TAX = .10 * PRICE
```

Thus, a literal is "literally" stated. It is not denoted by a name. There are two types of literals, *numeric literals* and *non-numeric* (i.e., alphanumeric) *literals.*

Numeric Literals

A numeric literal may consist of any combination of the numerals 0 to 9. One decimal point and/or one plus or minus sign may also appear in a numeric literal.

Rules for Forming Numeric Literals

1. A numeric literal intended for use in computation must consist of at least one numeral and not more than 18 numerals. Other numeric literals may contain as many as 120 characters.
2. One sign and/or one decimal point may appear in a numeric literal.
3. The sign of the literal, if present, must be the leftmost character. If the literal is unsigned, it is assumed to be positive.
4. The decimal point may appear anywhere in the literal except as the rightmost character. If no decimal point is used, the literal is treated as an integer.
5. Numeric literals must not be enclosed in quotation marks. If they are, they will be treated as non-numeric literals.

EXAMPLES

```
 173718281647356
 98.6
+1.54316
-.001
```

Note: When a plus or minus sign appears in a numeric literal, it must *not* be followed by a blank.

Non-Numeric Literals

There are occasions when it is convenient to write non-numeric information as a literal. For example, if a program were written to prepare a report in which every page was to have the same heading, the heading might be entered into the program as a non-numeric literal (sometimes called an alphanumeric literal). Another use of non-numeric literals is to display short messages at the computer console to inform the operator of any necessary manual operations during the course of the program. Non-numeric literals must never be used for computation. They are defined by enclosing them in quotation marks. The quotation marks are not part of the literals but are used only to define them as non-numeric literals to the COBOL processor. Thus, any set of consecutive characters, including blanks, which is enclosed in quotation marks will be treated as a non-numeric literal.

Rules for Forming Non-Numeric Literals

1. Any character in the computer character set except the quotation mark may be used in a non-numeric (alphanumeric) literal. Blanks are treated as characters and may be included freely.
2. A non-numeric literal may consist of from 1 to 120 characters.
3. A non-numeric literal must be enclosed in quotation marks.
4. If a literal conforms to the general format of a numeric literal but is enclosed in quotation marks, it is treated as a non-numeric literal and cannot be used for computation.

EXAMPLES

```
'THIS IS A NON-NUMERIC LITERAL'
'INVENTORY (OF DOOR MATS) BEGUN'
'−5326.7143'
```

Named Constants

The second way in which a programmer can specify a constant is to write it as a *named constant.*

A named constant is defined in the Data Division, where a data-name is assigned to it and its value is specified. A named constant is used exactly like any other data item, except that its value must *never* be changed during the course of the program. The details of specifying named constants are given in Chapter 6.

If, in the calculation of the tax shown earlier, the programmer had decided to use a named constant instead of a literal, he might have, in the Data Division, assigned the data-name TAX-PERCENT to the value .10. Then the following statements would have precisely the same meaning:

```
COMPUTE TAX = TAX-PERCENT * PRICE
COMPUTE TAX = .10 * PRICE
```

Figurative Constants

Figurative constants are certain named constants which have been provided with standard names. They are automatically recognized by COBOL so that the programmer need not define them in the Data Division of each program.

Hereafter in this manual, figurative constants will be considered as special types of literals. ZERO may be used as either a numeric or a non-numeric literal at any place in a program. All other figurative constants are considered as being alphanumeric and generally may be used wherever an alphanumeric literal would be appropriate.

A list of the figurative constants and their values is given below.

ZERO ZEROS ZEROES	These are the only figurative constants that may be used with either COMPUTATIONAL data *or* DISPLAY data. (See the discussion of USAGE in Chapter 6.) When used with COMPUTATIONAL data, ZERO, ZEROS, or ZEROES represents the numeric value zero. If used with a DISPLAY item, any one of the three represents a sequence of zero characters. The following figurative constants can be used only with data of DISPLAY USAGE:
SPACE SPACES	SPACE or SPACES represents a sequence of spaces (blanks).
HIGH-VALUE HIGH-VALUES	Either HIGH-VALUE or HIGH-VALUES represents a sequence composed of the highest character in the computer's *collating sequence.* (See the discussion of comparisons later in this chapter for an explanation of *collating sequence.*)
LOW-VALUE LOW-VALUES	These constants represent a sequence of the lowest characters in the computer's collating sequence.
QUOTE QUOTES	Either of these figurative constants represents a sequence of quotation mark characters.
ALL *(any non-numeric literal)*	This figurative constant generates a sequence of characters specified by *any non-numeric literal.* See the additional discussion several paragraphs below.

Note: No distinction is made between the singular and plural forms of figurative constants. When a figurative constant is used in such a way that the exact number of characters required cannot be determined, only one character is generated. For example, DISPLAY ZEROES would produce one zero character, since, in this case, the length of the sequence of zeros to be displayed cannot be determined. (See the discussion of DISPLAY in Chapter 7.)

EXAMPLES CONTAINING FIGURATIVE CONSTANTS

```
MOVE SPACES TO AREA-A
COMPUTE RATE = ZERO
IF NAME EQUAL TO HIGH-VALUES GO TO END-OF-JOB
DISPLAY QUOTE
DISPLAY 'PROGRAM' QUOTE 'ENDED' QUOTE
```

The last two examples would cause output to appear during the running of the program. The fourth statement would produce a single quotation mark. The message displayed by the fifth statement would be PROGRAM 'ENDED'; note that this message actually consists of four items, the non-numeric literal PROGRAM, the figurative constant QUOTE, the non-numeric literal ENDED, and the second figurative constant QUOTE.

ALL

The last figurative constant operates somewhat differently than the others. It consists of the word ALL, followed by an alphanumeric literal of the programmer's choice.

The reader will note that if we write MOVE SPACES TO PART-NUMBER the data item PART-NUMBER will assume the value of spaces. As the reader will see in Chapter 6, the programmer must specify in the Data Division the number of characters which

each data-name represents. If PART-NUMBER were specified as representing a value six characters in length, then the above command would make each of the six characters a space.

The ALL figurative constant works in a similar way. If we were to write MOVE ALL 'NONE' TO PART-NUMBER, the value assumed by the data item PART-NUMBER would become NONENO. The non-numeric literal used with ALL is repeated over and over until the number of characters represented by the data-name have been completely filled by the characters of the literal. If CODE is a data-name representing 11 characters, then the statement: MOVE ALL 'MAY 9' TO CODE would cause the item represented by CODE to assume the value MAY 9MAY 9M.

TALLY

The COBOL system finds use for one "special register" which is called TALLY. The word TALLY is the name of a data item which can assume a numeric value of up to 5 digits. The register it represents is used primarily to hold information produced by the EXAMINE verb (discussed in Chapter 7), but it may also be used by the programmer for other purposes.

Verbs

Verbs specify action. Whenever a programmer uses a verb, he will cause some action to take place. However, not all verbs cause action to take place as the program is being executed at object time. Some verbs are called *processor-directing verbs,* or, simply, *processor verbs.* They provide the processor with additional information needed to complete the source program. For example, during the course of writing a program, it is sometimes convenient to write information which will not be used in producing the object program, but which merely comments on the associated COBOL statements in order to make a listing of the program more meaningful to a reader. The processor verb NOTE is used to preface a message which will appear in the listing of the program, but which will have no effect whatsoever on the formation of the object program. The verb NOTE, therefore, may be used to complete the source program, but it does not cause an action in the object program. A complete discussion of processor verbs appears in Chapter 7.

Most verbs will cause some action to be taken at the time the program is executed. The verb DISPLAY will cause specified information to be printed or otherwise displayed, and the verb STOP will cause the computer to halt. Verbs which cause action at the time the program is executed are called *program verbs.*

Operators

There are other words and symbols besides verbs that cause action. For example, the expression A + B − C directs that C is to be subtracted from the sum of A and B. Thus, the symbols + and − are *operators.* In general, operators specify *actions* or *relationships* without actually expressing them in verb form. There are three basic kinds of operators: *arithmetic operators, relational operators,* and *logical operators.*

Arithmetic Operators

The complete list of arithmetic operators is given below. They are used in forming arithmetic expressions, as explained later in this chapter.

Operator	Meaning
+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Relational Operators

The need frequently arises in programming to make tests in order to determine what should be done next. COBOL provides a number of *relational operators* which enable the programmer to express the tests he wants performed. For example, the statement IF SALARY = ZERO is built around the relationship implied by the equal sign, which is a relational operator. Another statement might be IF AGE IS GREATER THAN 21 ADD A TO B. The words IS GREATER THAN form a relational operator. In this case, they specify that the IS GREATER THAN relationship between AGE and 21 must be fulfilled in order for A to be added to B.

A list of the relational operators in the COBOL language follows:

IS GREATER THAN
IS EQUAL TO or =
IS LESS THAN

Words in the above list which are underlined must be present when the operator is used. Words which are not underlined may be omitted, if the programmer desires, with no resulting effect on the meaning of the operator.

Relational operators are combined with data-names, literals, etc. to create conditional expressions and statements. The detailed rules for using relational operators will be presented later in this chapter under Conditional Expressions.

Logical Operators — AND, OR, and NOT

The three logical operators are AND, OR, and NOT. AND and/or OR are used when two or more tests are specified in the same expression. NOT is used to specify the negative of a condition.

Consider the following example:

IF MARRIED AND AGE NOT GREATER THAN 21 ADD A TO B

Notice how the words AND and NOT are used to augment the two basic tests. Because the tests are connected by AND, they both must be true for A to be added to B.

Consider the following:

IF MARRIED OR AGE NOT GREATER THAN 21 ADD A TO B.

This time the logical operator OR specifies that A is to be added to B if *either or both* conditions are fulfilled.

These logical operators will be discussed in detail and rules will be given for using them later in this chapter under Compound Conditions.

Restrictions on Words

The preceding sections of this chapter have discussed various types of words and how they may be used in a COBOL program. Generally, all words used in a program will be one of two kinds, those which are used as names (data-names, procedure-names, etc.), and those which specify procedure (verbs, connectives, etc.). The latter group actually includes every word in a COBOL program which is meaningful to the processor and which is *not* a name or literal.

As we have seen, names and literals are specified by the programmer. However, all other types of words used by the programmer must be selected from a preassigned list of words that have special meanings as explained in this manual. They *must* be used only according to COBOL rules. These "other" words are known as "COBOL *words*." COBOL words, then, are all words in a program, other than names and literals, which are meaningful to the processor. They include all verbs, con-

nectives, and operators, as well as certain other words to be discussed later. The most important rule governing COBOL words is that no COBOL word can be used as a name. A complete list of COBOL words is given in Appendix A. COBOL words may be separated into two categories: *key* words and *optional* words.

Key Words

Key words are words which are essential in conveying the meaning of a statement. A key word cannot be omitted. All verbs are key words. The word GREATER is a key word; logical operators are key words, etc.

Optional Words

Some words are not required in conveying the meaning of a statement but may or may not be used at the programmer's discretion in order to improve readability. They are called *optional* words. For example, all of the following expressions are correct and have the same meaning:

```
A IS GREATER THAN B
A GREATER THAN B
A IS GREATER B
A GREATER B
```

In these examples, the words IS and THAN are optional. They may be included or omitted without changing the meaning of the statement.

When two or more names are written in a series, the comma and/or the word AND may be used as optional *series separators.* The items of such a series must be separated by a space, but, in addition, they may be separated by a comma, the word AND, or a comma followed by AND. For example, all of the following expressions have exactly the same meaning:

```
ADD A AND B AND C
ADD A, B, AND C
ADD A, B, C
ADD A B C
```

Throughout the language, there are numerous cases where a word is optional in a certain place. If an optional word is used, it must be spelled correctly. If an optional word is omitted, it must not be replaced by any other word not explicitly stated to be its equivalent. It is important for the programmer to know which words are key words and which words are optional words in every statement he writes. Format notation used to identify key and optional words is given on page 28.

Syntax

The rules of syntax specify how words and symbols are put together to form intelligible statements. COBOL, like English, is written in sentences and paragraphs. However, before we consider a COBOL sentence, it is necessary to discuss thoroughly the main elements which make up a sentence. These are *expressions* and *statements.*

Expressions

An expression may be defined as a meaningful combination of names, literals, COBOL words, and/or operators which may be reduced to a single value. This definition will become clear after the reader has studied the two types of expressions employed in COBOL, the *arithmetic* expression and the *conditional* expression.

Arithmetic Expressions

An arithmetic expression is a combination of data-names and numeric literals joined by one or more arithmetic operators in such a way that the entire expression can be reduced to a single numeric value. (An arithmetic operator is any symbol

representing addition, subtraction, etc.; a list of arithmetic operators is given earlier in this chapter.)

The following are examples of arithmetic expressions:

```
(HOURS + OVERTIME * 1.5) * WAGE-RATE - FICA
PI * RADIUS ** 2 * HEIGHT / 3
WEEKLY-SALES * .05
```

Note that each of the above expressions is a combination of data-names and/or literals joined by arithmetic operators. At object time, each data-name will represent a value and, in each of the above examples, one numeric value will result from the specified computation. Thus, if WEEKLY-SALES has the value 574.20, the third example could reduce to the value of 28.71.

As will be seen later, arithmetic expressions may be included in *conditional* expressions. Thus, it is possible to test a given arithmetic expression to see whether it reduces to a specific value.

Order of Computation in Arithmetic Expressions

The way in which an arithmetic expression is to be evaluated can be specified by parentheses. Thus, the expression A * B + C might be considered ambiguous. Does the programmer mean (A * B) + C, or does he mean A *(B + C)? In COBOL, the programmer may use pairs of parentheses in order to describe exactly the way in which he wants the computation to proceed.

If parentheses are *not* written to specify the order of computation, COBOL will evaluate an arithmetic expression using the following rules:

1. All exponentiation is performed first.
2. Then, multiplication and division are performed.
3. Finally, addition and subtraction are performed.
4. In each of the three above steps, computation starts at the left of the expression and proceeds to the right. Thus, A * B / C is computed as (A * B) / C, and A / B * C is computed as (A / B) * C.
5. When parentheses are present, computation begins with the innermost set and proceeds to the outermost. Items grouped in parentheses will be evaluated in accordance with the above rules, and the result will then be treated as if the parentheses were removed.

Note: In general, two consecutive operators cannot appear in an arithmetic expression. All allowable possibilities are covered in the table of symbol pairs in Appendix A.

Conditional Expressions

In the general definition of expressions it was stated that any expression may be reduced to a single value. This definition holds for a conditional expression if we consider that a true expression reduces to the value of "truth," and a false expression reduces to the value of "falsity."

Thus, a conditional expression is an expression which, taken as a whole, may be either true or false, depending on the circumstances existing when the expression is evaluated.

Generally, a conditional expression will contain one or more variables, i.e., data items whose value may change during the course of the program. Then the truth or falsity of the expression depends on the particular value assumed by the variable or variables. For example, the expression YEARLY-INCOME IS GREATER THAN 5000 is a conditional expression, because it may or may not be true, depending on the value of YEARLY-INCOME. If YEARLY-INCOME has the value 4250, the expression is false; if it has the value 6700, the expression is true.

Conditional expressions are composed of one or more *conditions*. A condition expresses a relation which may or may not be true. Thus YEARLY-INCOME IS GREATER THAN 5000 states a relation between YEARLY-INCOME and 5000. It is a condition as well as a conditional expression. A conditional expression may contain several conditions, as will be seen later.

Items to Be Compared

The idea of determining the truth or falsity of a relation implies that two values will be compared and a decision will be made on the outcome of the comparison. It becomes important to know how comparisons are made.

Is it meaningful to ask: "Is the character A greater than the character 6?" "Are the numerals 0-9 greater or less than the letters A-Z?" "Is the asterisk less than 7? than X?" Since it is possible for a data item to have a non-numeric value, these questions must be resolved.

To answer them, we must see how a computer makes comparisons. The basic means the computer uses in comparing any two characters is its *collating sequence*. Each computer has its own character set, in which the characters have a specified order of precedence. This order is "built into" the machine, and every character meaningful to the computer has its position in this ordering. This order is called the collating sequence. Therefore, generally, it is meaningful to compare any character to any other character. The result will depend on the relative position of each character in the machine's collating sequence. The collating sequences for individual machines may be found in publications covering the various machines.

Comparison of Two Numeric Items

A comparison of two *numeric* items tests their arithmetic values. For example, a comparison of a data item which has a value of +000003 with a data item which has a value of +03 will result in an "equal" condition. Similarly, the value of 000000 is equal to the value of 000. In COBOL, zero is a unique value and is neither negative nor positive.

Two numeric items may be compared regardless of the USAGE specified in the Data Division for each item. See Chapter 6 for a discussion of USAGE.

Comparison of Non-Numeric Items

Comparisons are made in a different way if non-numeric items are being compared. It has been mentioned that the programmer must specify in the Data Division of his program the maximum number of characters each item of data may contain. In a comparison of two non-numeric items, each character in an item is compared with the corresponding character of the other item. The comparison begins with the leftmost character of each item. If these two characters are found to be equal, the next two are compared, and so on. As soon as an unequal condition is noted, the comparison stops and the result is recorded.

If each individual character comparison results in an equality and the two items consist of the same number of characters, the items are said to be equal.

When non-numeric items of unequal length are compared, the comparison proceeds as though the shorter item were filled out on the right with spaces until it is of the same length as the longer item. For example, if the value EXP9R73 were compared to the value EXP9T11, EXP9T11 would be found to be the greater because T is "greater than" R. In comparing the value EXP9R73 to the value EXP9R, EXP9R73 would be found greater, since the two items are of different lengths and the two additional right-hand characters in the longer item are not spaces. ABCD is less than BBCD, and *956Q4* is less than *957Q4*.

Simple Relational Conditions

The basic type of condition is the *simple relational condition.* Almost any relation between two items can be expressed by using simple relational tests. (However, it will be seen later that several "shorthand" ways of writing certain relations are available and are sometimes more convenient.) The general form in which a simple relational condition is written is:

$$\left\{\begin{array}{l}\textit{data-name}\\ \textit{literal}\\ \textit{arithmetic expression}\end{array}\right\} \left\{\begin{array}{l}\text{IS [NOT] GREATER THAN}\\ \text{IS [NOT] LESS THAN}\\ \text{IS [NOT] EQUAL TO}\\ =\end{array}\right\} \left\{\begin{array}{l}\textit{data-name}\\ \textit{literal}\\ \textit{arithmetic expression}\end{array}\right\}$$

The data-name, literal, or arithmetic expression to the left of the relational operator is known as the *subject* of the simple relational condition. The data-name, literal, or arithmetic expression on the right of the relational operator is called the *object.* The subject and the object of any given simple relational condition must not both be literals.

EXAMPLES:

```
STOCK IS LESS THAN (ORDER-POINT + 3500) / 3.25
.053 IS GREATER THAN SAMPLING-ERROR
AGE GREATER THAN 21
GROSS = NET
MARITAL-STATUS = 1
```

Using the Logical Operator NOT

It should be noted that the word NOT is included to make the relational operator specify exactly the opposite of the relation expressed without the word NOT. For example, AGE NOT GREATER THAN 21 is the opposite of AGE GREATER THAN 21. NOT is a *logical operator,* since it affects the "logical meaning" of the condition in which it is used.

However, NOT can be used in two ways with a simple relational condition. It may be used in the relational operator as in AGE NOT GREATER THAN 21, or it may precede the entire condition, as in NOT (AGE GREATER THAN 21). AGE NOT GREATER THAN 21 and NOT (AGE GREATER THAN 21) are exactly equivalent in meaning. If NOT precedes a simple relational condition which contains NOT in the relational operator, a "double negative" will result. For example, NOT (AGE NOT GREATER THAN 21) is a double negative and is equivalent to AGE GREATER THAN 21.

Condition-Names

Earlier in this chapter a condition-name was defined as a name given to one value of a data-name. In the Data Division, a condition-name is assigned to a particular value of a particular data-name. For example, in a program processing a payroll, the data item MARITAL-STATUS might be a code indicating whether an employee is married, divorced, or single. Let us assume that if MARITAL-STATUS has the value 1, the employee is single, if it has the value 2, he is married, and if it equals 3, he is divorced. To determine whether or not an employee is married, the programmer could test this condition by using a simple relational condition in a conditional statement such as IF MARITAL-STATUS = 2 SUBTRACT MARRIED-DEDUCTION FROM GROSS.

However, if he so chooses, he can associate a condition-name with each value that MARITAL-STATUS might assume. Thus, in the Data Division, the condition-name SINGLE might be associated with the "condition" that the data-name MARITAL-STATUS has a value of 1. MARRIED might be similarly associated with 2, and DIVORCED with 3. Then, as a "shorthand" form of the simple relational condition MARITAL-STATUS = 2, the programmer could write the single condition-name MARRIED. Therefore, the following two statements would produce identical results:

```
IF MARITAL-STATUS = 2 SUBTRACT MARRIED-DEDUCTION
    FROM GROSS
IF MARRIED SUBTRACT MARRIED-DEDUCTION FROM GROSS
```

The condition-name, then, is another form of a condition. It is an alternative way of expressing certain conditions which could be expressed by a simple relational condition. The details of specifying condition-names in the Data Division are given in Chapter 6.

Compound Conditions

As was stated earlier in this chapter, a conditional expression consists of one or more conditions. A conditional expression consisting of a single condition is known as a *simple conditional expression*; one composed of two or more conditions is called a *compound conditional expression.*

The conditions within a compound conditional expression are linked by the logical connectives AND and OR.

Suppose the following conditions were being used in a program:

```
NOT MARRIED
OVER-21
HOURLY-RATE IS LESS THAN 5.00
HOURLY-RATE IS GREATER THAN 3.50
```

These conditions could be linked by AND and OR in any sequence which would express a desired "overall" condition. Suppose we wanted to search a personnel file for every person married or not over 21 whose pay rate is between 3.50 and 5.00 dollars per hour. The following compound conditional expression would specify the desired overall condition:

```
(MARRIED OR NOT OVER-21) AND HOURLY-RATE LESS THAN 5.00
    AND HOURLY-RATE GREATER THAN 3.50
```

Rules for Forming Compound Conditional Expressions

1. In general, two or more conditions combined by AND and/or OR may make up a compound conditional expression.
2. The word OR is used to mean "either or both." Thus, the expression A OR B is true if: A is true, or B is true, or both A and B are true.
3. The word AND is used to mean "both." Thus, the expression A AND B is true if, and only if, *both* A and B are true.
4. Parentheses may be used to specify the order in which conditions are evaluated. Parentheses must always be paired. Evaluation begins with the innermost pair of parentheses and proceeds to the outermost. Thus, in the above example, MARRIED OR OVER-21 would be evaluated as either true or false before the rest of the expression is considered.
5. If the order of evaluation is not specified by parentheses, the expression will be evaluated in the following way: The conditions surrounding all ANDs will be evaluated first, starting at the left of the expression and proceeding to the right.

Then the ORS will be evaluated, also working from left to right. Thus, if the above example did not contain parentheses, it would be evaluated as MARRIED OR (NOT OVER-21 AND HOURLY-RATE LESS THAN 5.00 AND HOURLY-RATE GREATER THAN 3.50), which is not the desired condition.

6. If a compound conditional expression consists of several consecutive simple relational conditions, and if these conditions have common subjects and/or common relational operators, the common factors may be *implied* instead of explicitly repeated in each condition. The rules for writing implied subjects and operators are given later in this chapter.

The following examples of compound conditional expressions may help to clarify the above rules:

Compound Conditional Expression	Interpretation
MARRIED OR DIVORCED	Either or both of these two conditions must be true for the expression as a whole to be true.
MARRIED AND NOT AGE GREATER THAN 21	The condition MARRIED must be fulfilled and the condition AGE GREATER THAN 21 must *not* be fulfilled for this expression to be true.
A AND (B OR C) OR D	One of the following must be true for the expression to be true: 1. D must be true, or 2. A must be true and either B or C (or both) must be true, or 3. Both the above must be true.
A AND B OR C OR D	If both A and B are true, then the condition of C and D do not matter. If either A or B (or both) is false, then either C or D (or both) must be true for the expression as a whole to be true.

Other Types of Conditions

In order to facilitate the expression of many different types of conditions, COBOL contains three more ways of expressing a condition (in addition to the simple relational condition and the condition-name). These three forms of conditions are *sign conditions, class conditions,* and *switch-status-names.*

Sign Conditions

The *sign condition* can be used only in connection with numeric data. This form of condition may be used to express a test to see whether an item satisfies one of the following: (1) a "negative condition" (is less than zero), (2) a "zero condition" (equals zero), (3) a "positive condition" (is greater than zero). The general form for writing a sign condition is:

$$\left\{\begin{matrix}\textit{arithmetic expression}\\ \textit{data-name}\end{matrix}\right\} \text{ IS } [\underline{\text{NOT}}] \left\{\begin{matrix}\underline{\text{POSITIVE}}\\ \underline{\text{NEGATIVE}}\\ \underline{\text{ZERO}}\end{matrix}\right\}$$

If a data-name appears in a sign condition, it must represent a numeric value. If the value is unsigned and is not equal to zero, it is considered to be positive.

Note that GROSS IS NEGATIVE is equivalent to GROSS IS LESS THAN 0, that GROSS IS NOT ZERO is equivalent to GROSS IS NOT EQUAL TO 0, and that GROSS IS POSITIVE is equivalent to GROSS IS GREATER THAN 0. Any condition which can be expressed as a sign condition can be expressed as a simple relational condition. The sign condition is merely a convenient way of expressing certain things that could be expressed by a simple relational condition.

Class Conditions

The *class condition* can be fully understood only if the discussion of the CLASS clause in Chapter 6 has been studied. In general, the characters in a computer's character set may be placed into three categories. These categories are called *classes* and they are defined as follows:

1. The NUMERIC class includes the numerals 0 through 9 as well as *operational signs;* these will be discussed in Chapter 6.
2. The ALPHABETIC class includes the letters of the alphabet and the space.
3. The ALPHANUMERIC class includes every character in a given computer's character set.

The group of characters composing any given item falls into one of these three classes. Thus, it can be said that any item is NUMERIC or ALPHABETIC, or ALPHANUMERIC. The class condition may be used to test an item at object time to determine whether the item contains data which is either wholly numeric or wholly alphabetic. The form in which the class condition is written is:

---

data-name IS [NOT] $\left\{\begin{matrix}\underline{\text{NUMERIC}} \\ \underline{\text{ALPHABETIC}}\end{matrix}\right\}$

The *data-name* must be defined in the Data Division as being of the ALPHANUMERIC class.

Examples of class conditions are:

```
STOCK-NUMBER IS NOT ALPHABETIC
DATA IS NUMERIC
```

Switch-Status-Names

The last type of condition, the *switch-status-name,* is very similar to the condition-name. Most computers are equipped with one or more switches. Such switches may be set either "on" or "off," and their status can be tested by the program. A switch-status-name can be associated with either the "on" position of a switch or the "off" position, just as a condition-name is associated with one value of a data-name. Switch-status-names are assigned in the Environment Division (see Chapter 8). For example, suppose a given program produces a report. Sometimes it may be desired that the report be printed out immediately under control of the program (i.e., "on line"). At other times, the report is not so urgently needed and can be written on magnetic tape for later printing on an "off-line printer." The programmer could designate a switch to be "on" if immediate printing is desired, or "off" if delayed printing is wanted.

This switch may be given a name as explained earlier in this chapter under Special-Names. Then the "off" position could be given a *switch-status-name* such as DELAYED-PRINTING and the "on" position could be assigned the name IMMEDIATE-PRINTING. Then, at the point of writing the report, the following conditional sentence might be used to control the operation:

```
IF IMMEDIATE-PRINTING GO TO PRINT-ROUTINE OTHERWISE
    GO TO TAPE-ROUTINE.
```

Here, IMMEDIATE-PRINTING is the switch-status-name. The program will check to see if the *condition* of the indicated switch is "on," and control will be transferred accordingly.

Summary

We have seen that a conditional expression consists of one or more conditions. If two or more conditions make up a conditional expression, the conditions are linked by AND and/or OR.

The basic form of expressing a condition is the simple relational condition; however, four other forms may be used in certain cases. They are condition-names, sign conditions, class conditions, and switch-status-names.

Implied Subjects

Often a conditional expression will contain several consecutive simple relational conditions. These conditions may have common subjects. For example, the conditional expression AGE GREATER THAN 21 AND AGE LESS THAN 65 contains the common subject AGE. We will see how AGE can be *implied,* that is, stated in the first simple relational condition and then omitted in the second. Thus, if the second occurrence of AGE were implied, the expression would appear as AGE GREATER THAN 21 AND LESS THAN 65.

The expression AGE GREATER THAN 21 AND LESS THAN 65 OR = 16 OR = 18 would be interpreted as AGE GREATER THAN 21 AND AGE LESS THAN 65 OR AGE = 16 OR AGE = 18.

The following rules specify when and how implied subjects may be used:

1. Only conditions written as simple relational conditions may have implied subjects. Sign conditions and class conditions can never have implied subjects.
2. The first of a series of simple relational conditions must always consist of subject, operator, and object, and all of these must be *explicitly* stated.
3. Subjects may be implied only in a series of *consecutive* simple relational conditions connected by AND and/or OR.
4. When the subject of a simple relational condition is implied, the subject used is the first subject to the left which is explicitly stated. For example, A = B OR = C OR D = E AND = F is interpreted as A = B OR A = C OR D = E AND D = F, since D is the first stated subject to the left of = F.

The following examples illustrate these rules:

1. STOCK = ORDER-POINT OR STOCK IS LESS THAN ORDER POINT may be written as STOCK = ORDER POINT OR IS LESS THAN ORDER-POINT.
2. AGE GREATER THAN 21 AND MARRIED AND LESS THAN 65. This expression is *illegal,* because the subject of LESS THAN 65 is not stated and the next condition to the left is not a simple relational condition but a condition-name.
3. TAX IS GREATER THAN INCOME * .34 OR IS NEGATIVE. This expression is *illegal* because TAX IS NEGATIVE is not a simple relational condition; it is a sign condition and therefore cannot have an implied subject.

Implied Operators

In some cases, relational operators may be implied in a series of consecutive simple relational conditions in much the same way in which subjects can be implied. Thus, the expression AGE = 16 OR AGE = 18 OR AGE = 21 could be written as AGE = 16 OR 18 OR 21. Not only is the subject, AGE, implied in the last two conditions, but the relational operator, =, is also implied.

The following rules apply to the use of implied operators:

1. A relational operator can be implied *only* in a simple conditional relation where the subject is also implied. Thus, sign conditions and class conditions can never have implied operators (and, in fact, do not have operators).
2. When an operator is implied, it is assumed to be the operator of the nearest *completely stated* simple relational condition to the left. For example, TAX IS LESS THAN 100.00 OR (GREATER THAN 300.00 AND 500.00) would be interpreted as TAX IS LESS THAN 100.00 OR (TAX IS GREATER THAN 300.00 AND TAX IS LESS THAN 500.00).

Consider the following examples:

1. INCOME GREATER TAX AND INCOME GREATER INSURANCE could be written as INCOME GREATER TAX AND INSURANCE.
2. AGE = ZERO OR GREATER THAN 10 AND LESS THAN 21 OR 65 is equivalent to AGE = ZERO OR AGE GREATER THAN 10 AND AGE LESS THAN 21 OR AGE = 65.

Statements

The next larger unit of the COBOL language to be considered is the *statement.* The statement in COBOL is roughly comparable to the clause in English. In its simplest form, a statement contains a verb and its *operands.* (The word "operand" will be used frequently hereafter to designate *an item which is acted upon.)*

Imperative Statements

An imperative statement consists of one or more "commands." A simple imperative statement consists of one verb and its operands. Thus, ADD A TO B is a simple imperative statement. DISPLAY AREA UPON TYPEWRITER is another.

A compound imperative statement consists of a sequence of simple imperative statements. COMPUTE AREA = HEIGHT * WIDTH DISPLAY AREA UPON TYPEWRITER is a compound imperative statement, since it consists of two simple imperative statements. The simple statements making up a compound statement may be separated by the optional word THEN if the programmer so desires.

Conditional Statements

Imperative statements, as described above, direct the computer to perform certain specified actions. They give directions which are specific and unequivocal; the computer is given no option of *not* performing them. However, as has been shown earlier in this chapter, the programmer may wish the computer to perform an operation only under certain circumstances. In such a case, he would find it convenient to be able to modify an imperative statement so that its execution will depend on the evaluation of a conditional expression. He may therefore attach one or more conditional expressions to an imperative statement; such a statement then becomes a *conditional statement.*

There are three basic forms of conditional statements, but all have in common the fact that a stated action is performed only if a specified condition is present. The three forms are as follows:

Option 1

IF *conditional expression* *statement-1*

Option 2

IF *conditional expression* { *statement-1* / NEXT SENTENCE } { OTHERWISE / ELSE } { *statement-2* / NEXT SENTENCE }

Option 3

{ *statement-1* AT END / *statement-2* ON SIZE ERROR } *statement-3*

Whenever a conditional expression is encountered, it will be evaluated before any other action is taken. In general, if the expression is found to be true, the program will next carry out whatever action is specified immediately following the conditional expression, whereas if the expression is found to be false, the program will skip to some other statement, in accordance with the principles described below. The rules for using the three forms of conditional statements are as follows:

Option 1

If the *conditional expression* is found to be true, *statement-1* will be executed. *Statement-1* may be either of the following: (1) A simple imperative statement. (2) A compound imperative statement.

If the conditional expression is found to be false, *statement-1* will be bypassed; the program will instead proceed to the next sentence. A conditional statement in the form of Option 1 need not be the only statement in a sentence, but it must be the last. (See the discussion of sentences later in this chapter.)

EXAMPLES

```
IF SHIPPING-WEIGHT IS LESS THAN 10 GO TO PARCEL-POST-
    ROUTINE.
IF SERIAL-NUMBER IS GREATER THAN 2999 GO TO ERROR-
    ROUTINE.
```

In each of these cases, it is assumed that the only reason for deviating from the normal sequence of program steps would be the presence of the specified condition. There is thus no reason to specify an action to be taken if the condition is found to be false; where the programmer has such a need, he may use Option 2.

Option 2

Option 2 represents the fullest form of a conditional statement. If the *conditional expression* is found to be true, either *statement-1* or NEXT SENTENCE, whichever is specified, will be executed. *Statement-1* may be either (1) a simple imperative statement, or (2) a compound imperative statement. It may not be another conditional statement. If NEXT SENTENCE is specified, the program will bypass the remainder of the sentence containing the conditional statement and will proceed to the next sentence in the program.

If the *conditional expression* is found to be false, the program will take no action until it encounters either the word ELSE or the word OTHERWISE, one of which must be written. It will then execute *statement-2* or NEXT SENTENCE, whichever has been specified. *Statement-2* may be one of the following: (1) A simple imperative state-

ment. (2) A compound imperative statement. (3) A conditional statement. If NEXT SENTENCE is specified, a skip will be made to the next sentence in the program.

EXAMPLES

```
IF YR-TO-DATE-GROSS IS GREATER THAN 4800 NEXT SENTENCE
    OTHERWISE PERFORM FICA-ROUTINE.
IF YR-TO-DATE-GROSS IS GREATER THAN 4800 NEXT SENTENCE
    OTHERWISE IF CURRENT-PAY IS GREATER THAN ZERO PER-
    FORM FICA-ROUTINE.
```

In the first of these examples, *statement-2* is a simple imperative statement; in the second example, it is a conditional statement.

Option 3

This option represents a special form of conditional statement. It can be used only with the arithmetic verbs and the verb READ, as explained in Chapter 7, and its form is incorporated in the basic formats prescribed in that chapter for those verbs. It differs from other conditional statements because of the nature of the conditional expressions which are evaluated. There are two of these, as follows:

1. *The* AT END *condition.* When the records in a file of data are being made available to the object program by means of READ statements, the programmer may wish a particular action to be carried out if the end of the file is reached. The words AT END constitute a kind of switch which is used to alter the sequence of a program when an end of file is found. In Option 3, *statement-1* represents a READ statement, which must be written according to the rules specified in Chapter 7. The words AT END (or, simply, the word END) must be followed by *statement-3,* which prescribes the action to be carried out if the condition is found to exist. *Statement-3* must be an imperative statement. It *must not* be a conditional statement.

2. *The* SIZE ERROR *condition.* A computation specified by one of the arithmetic verbs may produce a result which is larger than the space the programmer has allowed for it in storage. In this case, a "size error" occurs, and it may have unforeseen effects. The programmer may therefore specify a test to determine if a size error has occurred. The special conditional expression used for this purpose consists of the words ON SIZE ERROR. In the format shown for Option 3, *statement-2* represents a statement employing one of the arithmetic verbs; this statement must be written in accordance with the rules given for those verbs in Chapter 7. It is followed by the words ON SIZE ERROR (or, simply, SIZE ERROR), and then by *statement-3,* which prescribes the action to be followed if a size error is found. Statement-3 must be an imperative statement. It *must not* be a conditional statement.

EXAMPLES

```
READ INVENTORY RECORD AT END PERFORM ANALYSIS-
    ROUTINE.
ADD INSURANCE TO TAX GIVING DEDUCTIONS ON SIZE
    ERROR GO TO RECOVERY-ROUTINE.
```

Sentences

The sentence is the basic unit of expression in the COBOL language, as it is in English. A COBOL sentence is made up of one or more statements, the last of which is terminated by a period.

Imperative Sentences

An imperative sentence consists of either a simple or a compound imperative statement terminated by a period and a space.

EXAMPLES

```
ADD PAY TO GROSS GO TO REFUND-ROUTINE.
COMPUTE AREA = HEIGHT * WIDTH DISPLAY AREA UPON
    TYPEWRITER SUBTRACT 1.5 FROM NEW-RATE.
MOVE INCOME TO WORK-AREA.
```

Processor-Directing Sentences

A sentence containing a processor-directing verb is called a processor-directing sentence. It must contain only the one verb and its operands (if any), followed by a period and a space. Two processor-directing verbs must never appear in the same sentence. Also, a program verb and a processor-directing verb must never be contained in the same sentence. Examples of processor-directing sentences follow:

```
ENTER AUTOCODER.
ENTER COBOL.
```

Conditional Sentences

A conditional sentence is a conditional statement terminated by a period; the conditional statement may be preceded by an imperative statement not terminated by a GO or STOP RUN (see Chapter 7). Examples of conditional sentences follow:

```
IF AGE GREATER THAN 21 ADD 1 TO MAJOR-COUNT.
IF AGE GREATER THAN 21 ADD 1 TO MAJOR-COUNT ELSE ADD
    1 TO MINOR-COUNT.
MOVE MASTER-RECORD TO WORK-AREA ADD 600 TO INCOME-
    TO-DATE ON SIZE ERROR GO TO ERROR-ROUTINE.
```

Punctuation of COBOL Sentences

Wherever possible in COBOL, punctuation has been made non-critical; that is, lack of punctuation will not result in an error. However, the following rules must be observed when punctuating:

1. At least one space must always appear between two successive words and/or parenthetical expressions. Two or more successive spaces are treated as a single space except in non-numeric literals.
2. If a period or comma follows a word or parenthetical expression, it must appear immediately after the word or expression and then be followed by a space.
3. Each sentence must be terminated by a period and a space in that order.
4. When an arithmetic operator or an equal sign is used, it must be preceded by a space and followed by another space.
5. When a series of operands occurs within a statement, the items may be separated by series separators, which are the comma, the word AND, and the comma followed by AND. For example, ADD A B, C AND D, AND E TO F.
6. Subscripts must be enclosed in parentheses. If two or more subscripts are present within the parentheses, they must be separated by spaces and can be separated by commas as well. (See the discussion of subscripts in Chapter 4.)
7. Parentheses may be used wherever needed in arithmetic expressions and conditional expressions. Where ambiguity would result from their omission, they *must* be used.

8. The rules of punctuation do not apply to constants defined in the Data Division, or to non-numeric literals, which may contain any characters except the quotation mark.
9. The statement separator THEN may be placed before or after any imperative statement but should not be used at the beginning or end of a sentence.

Paragraphs

COBOL sentences may be combined to form paragraphs. A paragraph, as in English, may contain one or more sentences. Every paragraph must begin with a procedure-name (i.e., there must be no unnamed paragraphs). A sentence within a paragraph cannot be assigned a procedure-name, but a paragraph may consist of only one sentence. For example, processor-directing paragraphs often will be one-sentence paragraphs. The rules for writing paragraphs on a coding sheet are given in Chapter 5.

Sections

One or more paragraphs can be grouped into a *section.* The section is the largest unit in COBOL to which a procedure-name may be assigned. Sections must always be named. This is done by writing a procedure-name, followed by the key word SECTION, followed by a period; the remainder of the line on which it is written must be left blank. The Procedure Division need not be broken into sections at all if the programmer does not find it convenient. The rules for writing sections on a coding sheet are given in Chapter 5.

Chapter 4: Concepts of Data Organization

In COBOL, writing the Procedure Division might be compared with teaching a set of rules to clerks who would manually maintain a set of files. These clerks would have to be instructed to transfer certain data from incoming forms to the master ledger, to perform any required computations, etc. In general, the rules which a clerk follows when maintaining a set of books are comparable to the statements which make up the Procedure Division of a COBOL program.

However, a clerk must have another type of information at his disposal in order to do his job effectively. He must know some of the general characteristics of the data itself that he will handle.

Often, much of the information regarding the nature and format of the data to be processed may be obtained from the form or ledger in which the data is recorded. The following questions would be relevant when organizing any new bookkeeping system:

1. What forms will be needed in this system? A master ledger? Several types of "transaction" forms to carry out everyday business?
2. What information will each form contain? How will the spaces for each item of data be placed on each form? What is the most convenient grouping of related data? Should the description of an item be placed next to its unit price? Should a given address be broken into separate items such as street, city, zone, and state?
3. How large should the space for each item be? Should spaces which will always contain monetary values be provided with a preprinted dollar sign? Should certain spaces contain an indication of where the decimal is to be placed to insure proper alignment of digits?
4. Where will these records be kept? What types of filing cabinets are available?

In effect, the answers to the above questions describe important characteristics of the data to be handled in a data processing job. To insure the proper and efficient handling of data by a computer, similar information must be provided to the COBOL processor regarding the data to be encountered by the object program. This information, concerned with the organization and the format of the data to be processed, is provided in the Data Division. The relation of data items to each other, the length of each data item, the placement of decimal points, dollar signs, etc. are all specified in the Data Division.

All data-names are assigned in the Data Division to the items they represent. Condition-names are also specified there. Every name which is referred to in the Procedure Division must be described in the Data Division with the following exceptions:

1. Procedure-names.
2. Special-names, which are defined in the Environment Division.
3. Figurative constants, which are names "pre-assigned" to standard values, as shown in Chapter 3.

The Organization of Related Data

In any application, related data is grouped in certain ways. Even the largest mass of data can be treated as a single item. Thus, a dictionary can be considered a book

(which is an *item*), or it can be considered a series of chapters (which is a *series* of *items*), or a series of definitions, words, or characters (which are other series of items). In each case, there is some pattern by which the items are grouped. Likewise, bodies of related information are grouped in an orderly way in COBOL.

A *file* is the largest body of related information in a COBOL program. A file may consist of any number of *records,* and records are made up of *group items* and *elementary items.*

Elements and Groups of Elements

An element of data (elementary item) is a piece of data which is never further divided. DATE could be the name of an elementary item if it were never referred to as anything but DATE. However, if the Procedure Division ever referred to only a part of the date, say, the month, then DATE would have to be broken into parts. These parts might be named DAY, MONTH, and YEAR. Then DATE would not be the name of an element, because it is subdivided into DAY, MONTH, and YEAR.

The term *item,* as used in this manual, refers to any element or group of elements. Thus, it is correct to say that DATE is the name of an item. However, it is not the name of an elementary item, because the item is actually a group—DAY, MONTH, and YEAR.

An item can also be a group of groups. Suppose the item YEAR in the previous example were divided into two parts, named DECADE and YR, for example. YEAR would not be the name of an element—it would be the name of an item consisting of a group of elements. Then DATE would be the name of an item that was a group containing a group.

To review, elements can be combined to form groups which, in turn, can form groups of groups. Either an element or a group may be regarded as an item. An element is a unit of data which is never broken into smaller units. The discussion of levels (later in this chapter) will help to clarify the concept of elements and groups of elements.

Records

Elements and groups of elements of data are combined to make up *records.* In a program processing a payroll, the permanent data concerning each employee would probably constitute a single record. Items in this record might be the employee's man number, his name, shift, rate of pay, marital status, number of dependents, etc. Thus, there would be a *series* of similar records, one for each employee. All of these records would usually be of the same format, i.e., would contain the same items; but the items would have different values for each employee. Probably, they would all be stored together on a single reel of magnetic tape or in a single deck of cards. Another series of records (again, one for each employee) might contain the record of each man's time card for the past week. Thus, the payroll program would have two records available pertaining to each employee, one containing permanent information, the other containing the record of the employee's past week of work. As will be seen below, these two different types of records would usually be kept in different *files.*

An important characteristic of the record is that it is the unit of data which is handled by the READ and WRITE verbs. (See Chapter 7.) If, in the above example, the records containing the permanent information for all employees are stored in a file called PERMANENT-PAYROLL-INFORMATION, then each time the statement READ PERMANENT-PAYROLL-INFORMATION RECORD is encountered, one man's permanent information would become available for processing. That is, it would be "read" into the system. In COBOL, it is not possible to read only a fraction of a record and not the rest of the record. Similarly, only an entire record can be "written," that is, made available for output to an external medium such as magnetic tape.

Files

Conceptually, a file in COBOL may be compared to a filing cabinet or group of cabinets which hold a collection of data, all of a certain type. The master records for all checking accounts in a bank might make up one file. If this master file of checking accounts were maintained manually, it would occupy a number of drawers in one or more filing cabinets. If it were maintained electronically, it might occupy one or more reels of magnetic tape. In either case the file would be made up of a number of records.

In COBOL, then, a *file* is a series of records containing related information. As will be seen in Chapter 6, however, all the records in a file need not have the same format. Every time a record is read by a READ statement, the incoming record replaces any record from the same file which was previously available. If the needs of the program require that two records from the same file be available concurrently, the first must be read and then moved to a *work area* before the second is read.

In addition to data records, a file often contains *label* records, usually preceding and following all of the data records. In a sense, these label records are similar to labels on filing cabinet drawers. In general, they serve as a checking device to make sure that the operator mounted the right tape on the right unit, that the second reel of a file is not processed before the first reel, etc. Most of these details are handled automatically. Further discussion of label records may be found in Chapters 6 and 7.

There is no relation between the length of a file and the capacity of a reel of tape. A file may occupy more than one reel of tape or one reel of tape may contain several files.

Two verbs which deal directly with a file are OPEN and CLOSE. Before any records can be read from a file, the file must be "opened." Among other things, the OPEN verb checks the label to insure that the proper tape reel is present. After the program has finished processing all the records in a file, the file should be "closed." CLOSE will process or write any labels at the end of the file, as specified by the programmer, and then make the file unavailable for further use until it is again opened. Thus, when a file is closed, a READ or WRITE verb cannot be used to refer to any records in that file. OPEN and CLOSE may be thought of as performing functions similar to unlocking or locking a filing cabinet.

Data Division Entries

After a programmer has determined how his items of data will be organized into records and files, he must make this organization known to the COBOL processor. He does this in the Data Division of his program. The Data Division is made up of *entries*, each of which is composed of a number of independent *clauses*.

In the Data Division, the programmer is not free to create sections at his own discretion. A Data Division always contains from one to three sections. They are called the *File Section*, the *Working-Storage Section* and the *Constant Section*, and they must be arranged in that order.

The *File Section* contains all entries which describe incoming and outgoing data. As will be seen later, each file is described by a *File Description entry* and each record is described by one or more *Record Description entries*. The specific format of each type of entry is described in Chapter 6.

The *Working-Storage Section* contains Record Description entries that specify any work areas which may be needed during the processing of information in data records. Work areas are to the computer what scratch pads are to the clerk. When doing a series of computations with pencil and paper, one often finds occasion to

"jot down" intermediate results. In the computer, work areas allow the temporary storage of intermediate results. Sometimes a record will be read and then parts of it will be moved to a work area where they can be separated, processed, and reassembled again. The details of writing the Working-Storage Section are discussed in Chapter 6.

In Chapter 3, we encountered named constants and said that they were specified in the Data Division. All named constants must be defined in the *Constant Section* of the Data Division. They are defined by the same type of entries that specify file records and work areas, that is, Record Description entries. The value of a constant is specified by one of the clauses which may appear in a Record Description entry, the VALUE clause. The details of writing this section appear in Chapter 6.

Levels

Level-numbers are used in COBOL to show how data items are related to each other. The general concept of *levels* can be observed in many places other than computing. The organization of this manual, for example, illustrates an application of the concept of levels. The structure of this manual may be summarized as follows: The manual is divided into three parts; each part is composed of one or more chapters; each chapter is made up of several main topics; each main topic is composed of subtopics, etc. This "hierarchy" is shown clearly in the Table of Contents by using different indentations and type faces to show the different *levels* of headings. All chapter headings are printed using the same style and size of type. The titles of the three parts are printed in the same way to show that each part is equal in the structure of the manual to each other part. The headings of the topics within each chapter are printed in the Table of Contents so the reader can determine the structure of the chapter at a glance.

Data in a COBOL program is organized into a hierarchy much like the hierarchy of organization of the material in this manual. We have already seen that the "highest level" (the most inclusive) grouping of data is the file and that the next lower level grouping is the record. Also, it was mentioned that records are made up of items which may be groups or elements. The COBOL processor must know what records are in each file, what items are in each record, and which items are groups and which are elements.

Consider a data record which contains, as one item, a man's address. The address might be given the data-name ADDRESS. However, if the programmer, somewhere in his program, has to refer to the individual parts of the address, he has to name them. He might name them STREET, CITY, ZONE, and STATE. If he wished to "move" the item CITY from one place in the computer to another (using the verb MOVE), he could do so in two ways. He could write MOVE CITY, which would move only the item CITY, or he could write MOVE ADDRESS and all the parts of ADDRESS could be moved at once, CITY included. In this case, ADDRESS is at a higher level than CITY, and ADDRESS includes CITY. Obviously, the COBOL processor must have a way of knowing that CITY, STATE, etc. are parts of ADDRESS. This is done by grouping the entries describing the related items and assigning a *level-number* to each item. If ADDRESS were assigned a level-number of 05 and STREET, CITY, ZONE, and STATE were each assigned a level-number of 06, this would indicate that the last four were of a lower level than ADDRESS. In general, the *higher* the numeric value of the level-number, the lower the hierarchical level of the item; the number denoting the record level is 01; the lowest level is 49. (The level-numbers 77 and 88 are special cases which will be explained later in this chapter.) The file level is indicated by the special level indicator FD in place of a level-number, as explained in Chapter 6.

The effect of levels is illustrated in the following example. This is the description of a record making up a file which contains the master information for the checking accounts of a bank. Each record contains information pertaining to one checking account. Items of data include: NAME, which is broken down into three elements, LAST, FIRST and MIDDLE; ACCOUNT-NUMBER; ADDRESS, which is broken down into STREET, CITY, ZONE, and STATE; and BALANCE. The record itself is called CHECK-ACCOUNT-RECORD.

The description of this record in the File Section of the Data Division would consist of a series of Record Description entries, one for each item. The first item of a Record Description entry is always a level-number; the second is the data-name which will be assigned to the item. These are followed by several clauses which completely describe the item. (These clauses are described in Chapter 6 and are omitted here for the sake of simplicity.)

The entries might be arranged as follows:

```
01  CHECK-ACCOUNT-RECORD . . . .
    02  NAME . . . .
        05  LAST . . . .
        05  FIRST . . . .
        05  MIDDLE . . . .
    02  ACCOUNT-NUMBER . . . .
    02  ADDRESS . . . .
        04  STREET . . . .
        04  CITY . . . .
        04  ZONE . . . .
        04  STATE . . . .
    02  BALANCE . . . .
```

In the above example, NAME is assigned the level 02. Immediately following are entries describing LAST, FIRST, and MIDDLE. These three items are identified as being contained in NAME because:

1. They follow NAME and no other item with a level-number equal to or numerically lower than that of NAME has intervened.
2. They have level-numbers numerically higher than that of NAME.

Consider another example, this one a portion of a payroll record:

```
01  PAYROLL-RECORD . . . .
    03  MAN-NUMBER . . . .
    03  NAME . . . .
        27  LAST . . . .
        27  FIRST . . . .
        27  MIDDLE . . . .
    03  ADDRESS . . . .
        12  STREET . . . .
        12  CITY . . . .
        12  ZONE . . . .
        12  STATE . . . .
    03  DATE-HIRED . . . .
        04  MONTH . . . .
        04  DAY . . . .
        04  YEAR . . . .
        14  DECADE . . . .
        14  YR . . . .
    03  RATE-OF-PAY . . . .
```

The principal rules for assigning level-numbers are illustrated in this example and may be summarized as follows:

1. The level 01 is reserved exclusively to identify a record.
2. Any item, B, is contained in any other item, A, if all of the following conditions are met:
   a. The entry for B follows the entry for A.
   b. B has been assigned a numerically higher level-number than A.
   c. An entry with a level-number numerically equal to or lower than that for A does not occur between A and B. Thus, even though DECADE has a higher level-number than STATE, it is not part of STATE, because entries with level-numbers lower than 12 occur between STATE and DECADE.
3. Level-numbers need not be assigned consecutively. Thus, MONTH, DAY, and YEAR could be assigned any level-number numerically higher than that assigned to DATE-HIRED and lower than that assigned to DECADE.
4. A group may include more than one level. However, all items which make up a level within the same group (i.e., MONTH, DAY and YEAR) must have the same level-number.
5. When an entry is to have a numerically lower level-number than the one immediately preceding it, the level-number must be chosen from the level-numbers of the groups which include the preceding item. Thus, ADDRESS must be assigned level 03, because that is the only level-number assigned to a group that contains the preceding item, MIDDLE. ADDRESS could not be assigned the level-number 01, because it is not a record. RATE-OF-PAY could assume one of two level-numbers, 03 or 04, depending on the programmer's wishes, since the item YR is contained in two groups, one with the level-number 04, and the other with the number 03. However, if it were assigned the number 04, it would be treated as a part of the group called DATE-HIRED, which would probably not be the programmer's intent.

Special Level-Numbers

There are two level-numbers which are used for identifying special types of entries. They are the level-numbers 77 and 88.

The level-number 77 is used when describing certain constants or work areas. If a constant consists of one item which is not further subdivided and which is not a part of any larger constant, then it is said to be *independent*. Similarly, if the programmer needs a work area that is not composed of several parts and is not part of a larger work area, it is called an independent work area. These need no level-numbers to show their relationship to other items, since they stand completely by themselves and are not a part of any hierarchy; the special level-number 77 is used to denote independent items of this kind.

The second special level-number, 88, is used to identify condition-names. A condition-name is *not* the name of an item; it is the name of a *value* which an item may assume. Thus, it is given the special level-number 88. Condition-names are assigned by writing an entry for the item of data itself, immediately followed by an entry for each condition-name to be associated with the item. For example, if the data item were MARITAL-STATUS, and the three conditions were MARRIED, SINGLE, and DIVORCED, the entries might be grouped like this:

```
05  MARITAL-STATUS . . . .
    88  MARRIED . . . .
    88  SINGLE . . . .
    88  DIVORCED . . . .
```

The details for writing these "special" Record Description entries are discussed in Chapter 6.

Qualification of Names

The general concept of qualification of names was presented in Chapter 3. Qualification is required in many cases when a single data-name has been used to name more than one item. Thus, if an item of data in MASTER-RECORD is called NAME, and an item of data in DETAIL-RECORD also is called NAME, then the data-name NAME is not *unique*. Whenever either item is referred to, it must be associated with one or more additional names to make it unique, as explained below. In this case, it must appear in qualified form such as NAME IN MASTER-RECORD or NAME IN DETAIL-RECORD, whichever is meant.

Rules for Qualification of Names

1. If any data-name, paragraph-name, or condition-name is assigned to more than one item in a program, it must be qualified whenever it is referred to in the Procedure Division, the Data Division, or the Environment Division.
2. A name may be qualified by writing either IN or OF after it, followed by the name of a group which contains it, as explained in Chapter 3.
3. A number of qualifiers may be required in order to make a name unique. More qualifiers may be used than are absolutely needed. Thus, if NAME OF VENDOR is needed to make NAME unique, NAME OF VENDOR OF DESKS IN MASTER-RECORD is also permitted, although not required.
4. A qualifier must be the name of a group which contains the item it is qualifying.
5. A name must not be assigned to a group which contains an item of the same name. In other words, a name cannot appear to qualify itself. For example, in the payroll record used previously to illustrate levels, the group YEAR contains an element YR; but YEAR could not contain an element named YEAR. Thus, YEAR OF YEAR is not permitted.
6. The name of an item to which condition-names have been assigned may be used to qualify any of its condition-names. For example, SINGLE OF MARITAL-STATUS is permitted where MARITAL-STATUS is the item which has a condition-name SINGLE associated with it.
7. A paragraph-name must not be duplicated within the same section.
8. A paragraph-name can be qualified only by a section-name. When it is qualified by a section-name, the word SECTION must not appear as part of the qualifier.

Subscripts

The need often arises to have entire tables of information accessible to a COBOL program. Specific items are often referred to in such tables by means of *subscripting*.

Like all other data, tables must be described in the Data Division of the program. Without subscripting, this could become tedious, since each item in the table would have to be described in a separate entry. Consider a table consisting of the populations of the 50 states. The first item is the population of Alabama, the second, the population of Alaska, etc. according to alphabetical order of the states. If the population of Alabama were named ALABAMA-POPULATION, and each of the 49 other entries were similarly named, any one population could be referred to at any time. However, this would entail writing 50 Record Description entries, one describing the population figure for each state. A shorter way of describing this table would be to use the OCCURS clause, one of the optional clauses which may be used in a Record Description entry. The details of this clause are described in Chapter 6; however, this clause essentially allows one to describe a series of similar items once for all of the items. In this case, we would write a Record Description entry de-

scribing the population data for one state—perhaps as an 8-place integer—and then say that this item "occurs" 50 times. Now we have written one entry which will describe 50 items, but they do not have individual names. They all have been named collectively by the name appearing in the entry describing all 50 items. Describing a table in this way would require a total of two Record Description entries. They would look something like the following:

```
01  POPULATION-TABLE . . . .
    02  POPULATION . . . OCCURS 50 TIMES . . . .
```

The data-name POPULATION-TABLE represents all of the fifty items, i.e., the whole table.

We can now refer to each population figure in the following way: We know that the populations are arranged by alphabetical order of the states. Thus, the population of Alabama is the first figure in the table, the population of Alaska is the second, etc. Therefore, we can refer to the population of Alabama as POPULATION (1) and the population of Alaska as POPULATION (2). This is called *subscripting*. Numeric literals or data-names used to locate items in a list or table are known as *subscripts*. Subscripts are always placed in parentheses following the data-name to be subscripted.

In the above example, we have, in effect, set up a code number for each state; that is, 1 represents Alabama, 2 represents Alaska, and so on for each of the 50 states. If we set up an item called STATE, and then STATE assumes a value of 1,2,3, . . . or 50, we could write POPULATION (STATE), and the current value of STATE would be used as the subscript of POPULATION. The two following examples are precisely equivalent in effect and would refer to the population of Arizona, provided the value of STATE is 3:

```
MOVE POPULATION (3) TO WORK-AREA
MOVE POPULATION (STATE) TO WORK-AREA
```

Thus, we have seen that a subscript can be either a numeric literal or a data-name. If it is a data-name, it may be qualified; for example, if STATE were not unique, then the subscript might be POPULATION (STATE IN ADDRESS OF VENDOR). Similarly, if POPULATION were not unique, it, too, could be qualified, as in POPULATION IN VITAL-STATISTICS (STATE IN ADDRESS OF VENDOR).

Suppose, now, we expand our table of populations so that it contains the population figures for 1900, 1910, 1920, 1930, 1940, 1950, and 1960 for each state. Now we have a list of 50 "states," each of which is actually a list of seven population figures. Any item in this table can be obtained by using two subscripts. The table would be described in the Data Division as follows: As before, the entire table will be called POPULATION-TABLE, and the entry for each state will be called POPULATION. However, we now have one additional entry, which will be called DECADE. DECADE represents the population at a particular date. Each "level" in the table requires one Record Description entry to describe it; thus, this table requires three entries to describe the 350 items. The entries might look like this:

```
01  POPULATION-TABLE . . . .
    02  POPULATION . . . OCCURS 50 TIMES . . . .
        03  DECADE . . . OCCURS 7 TIMES . . . .
```

To refer to the population of a particular state for a particular year, two subscripts must be used, one to indicate the state and the other to indicate the year. For example, if we want to refer to the population of Alaska in 1960, we would want the seventh number in the list of populations for the second state in the list of states. We could write this as DECADE (2, 7). Note that the name of the lower-level item, not the higher, is the one subscripted. As before, we could also represent one

or both of the subscripts by data-names, instead of numeric literals, and write DECADE (STATE, YEAR) and a population figure would be selected, depending on the current values of the data-names STATE and YEAR. If the name DECADE is not unique in the program, it must be qualified. In that case, the above examples might be written as DECADE OF POPULATION (2, 7) and DECADE OF POPULATION (STATE, YEAR), respectively.

COBOL allows a maximum of three subscripts. This may be illustrated by enlarging the above table so that for each year and each state two population figures are present, one for the urban population, and one for the rural population. Then the Record Description entries might look like the following:

```
01  POPULATION-TABLE . . . .
    02  POPULATION . . . OCCURS 50 TIMES . . . .
        03  DECADE . . . OCCURS 7 TIMES . . . .
            04 RURAL-URBAN . . . OCCURS 2 TIMES . . . .
```

Then the urban population of Alabama in 1930 could be referred to as RURAL-URBAN OF DECADE OF POPULATION (1, 4, 2). This assumes that for each year the rural population is given before the urban population. The unqualified form of this would be RURAL-URBAN (1, 4, 2).

To refer to *all* of the 14 populations which the table contains for the state of Wyoming, one could write POPULATION (50). Or to refer to the set of two populations for the state of Wyoming for the year 1920, one could write DECADE OF POPULATION (50, 3) or DECADE (50, 3). If one wished to refer to the *whole* population table, the data-name POPULATION-TABLE would be used.

General Rules Pertaining to Subscripting

1. A subscript must always have a positive integral value. It may be specified by a literal or a data-name.
2. Subscripts are placed in parentheses to the right of the subscripted data-name with a space intervening. If the subscripted data-name is in qualified form, the subscripts must appear to the right of *all* qualifiers, for example, WORD IN PAGE IN VOLUME (2, 61, 211).
3. When two or three levels of subscripts are present, the subscripts are separated by commas and arranged from left to right in order of decreasing inclusiveness. A space must follow each comma. Thus, word 211 on page 61 of the second volume of a book might be referred to as WORD OF PAGE OF VOLUME (2, 61, 211). A maximum of three levels of subscripting is permitted.
4. A subscripted data-name must be qualified when it is not unique, in accordance with the rules of qualification.
5. A data item which involves an OCCURS clause at its own level or which has an OCCURS clause implicit from a higher level must be subscripted whenever it is referred to by the programmer. Furthermore, subscripting can be used only with such a data item.
6. The programmer may refer to blocks (sets) of data within a table in the following way: The data-name of the block is written, followed by the subscript of the particular block wanted, plus any other subscript which might be necessary to locate it. For example, the whole of page 61 in Volume 2 can be referred to as PAGE (2, 61) or PAGE OF VOLUME (2, 61). All of Volume 2 may be referenced by VOLUME (2). A complete table may be referred to by using the name of the table.
7. If references to an item which has been assigned condition-names require subscripting, then the condition-names must also be subscripted whenever mentioned.
8. Subscripts must not themselves be subscripted.

Differences Between Qualification and Subscripting

Qualification and subscripting should not be confused. They are very different things. Qualification involves appending additional data-names to a name which has been used to represent several different items. Subscripting is used to refer to one item among a group of items which are organized as a table or a list.

The Library

The COBOL library, usually referred to hereafter simply as "the library," is a collection of pre-written parts of COBOL programs. It is built up by the programmers at a given installation and is usually recorded on tape and is made available to the COBOL processor at the time the source program is processed. The library may contain data descriptions and environment descriptions which are used in a number of the COBOL programs written at a given installation. For example, a series of Record Description entries describing a "standard" record format might appear in the library. Then any COBOL program which processes records of this type could copy the library entries describing these records into the source program by using the COPY clause as explained in Chapter 6.

The library contains two types of entries which correspond to two of the divisions of a COBOL program, Data and Environment.

Entries Associated with the Data Division

File Description entries and Record Description entries which have been placed in the library may be included in a source program by the use of the COPY clause, which is described in Chapter 6. Each File Description entry in the library is a separate entry. If such an entry is referred to by a COPY clause, only that single entry is extracted from the library. The programmer must make separate provisions for the appearance in the source program of the Record Description entries associated with that file. However, if a COPY is used with a Record Description entry, it may extract one entry, or a number of entries, from the library, depending on the material in the library. This is discussed in detail in Chapter 6 in connection with the COPY clause.

Entries Associated with the Environment Division

Any of the four main paragraphs which constitute the Environment Division may appear in the library and can then be placed in a source program through the use of the COPY option of each of the paragraphs.

A complete section of the Environment Division cannot be extracted from the library by a single COPY; that is, the COPY option operates only on the paragraph level. Details on the use of the COPY option appear in Chapter 8.

Part III:
The Divisions of a COBOL Program

Chapter 5: Reference Format—The COBOL Program Sheet

When the programmer defines a problem in the COBOL language, he must use the language very precisely, in accordance with the rules given in this manual. He must be equally precise in arranging and describing his data so that it will be acceptable to the object program, and in describing the machine components that affect the creation of the object program.

Since all of this information must eventually be entered into the computer, it follows that the programmer should also employ a very precise way of recording it so that it can be processed correctly. A standard *Reference Format* is used for this purpose. The Reference Format prescribes the sequence in which the source program is to be entered into the machine, and it also prescribes certain ways in which information within the source program must be arranged.

Reference Format Principles

Basically, the design of the Reference Format provides what might be called "starting points" or "left-hand margins" for writing each of various kinds of information used in the program. These margins furnish the processor with reference points which it requires in analyzing the source program. The programmer must observe these margins in recording his program; for his convenience, therefore, the COBOL Program Sheet, which is used for this purpose, incorporates in its design the principal features of the Reference Format. (See Figure 5-1.)

The Reference Format serves three main purposes:

1. It provides a standard and convenient form in which the programmer can state his program. The COBOL Program Sheet, which is based on it, is an aid to both the programmer and the card punch operator in arranging the program in the proper form.
2. It identifies to the COBOL processor certain kinds of items, such as procedure-names, for its use in creating the object program.
3. It provides a standardized form in which the printed listing of the source program will appear. This standard form has the particular merit that it may be used as the input form to a COBOL processor on another machine. The format is thus valuable in furthering compatibility among programs and machine systems.

This Reference Format, despite its necessary restrictions, is of a relatively "free" form. The programmer should note, however, that the rules for using it are precise and must be followed exactly, and that these rules take precedence over any other rules given in this manual with respect to the spacing of names, data, and other textual material.

The COBOL Program Sheet

Figure 5-1 shows a COBOL Program Sheet, which describes the Reference Format used in the COBOL system. The design of this form reflects the fact that much of the information to be entered into a computer is first punched into standard IBM cards. The form provides for 80 columns of information, corresponding to the 80 columns of a card. However, this format is not limited to card input or output, since the equivalent of card format can be represented in such other media as magnetic tape and punched paper tape.

IBM

COBOL PROGRAM SHEET

PAGE i 3	PROGRAM	SYSTEM	SHEET OF
	PROGRAMMER	DATE	IDENT. 73 80

SERIAL 4 6 | CONT. | A 8 | B 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72

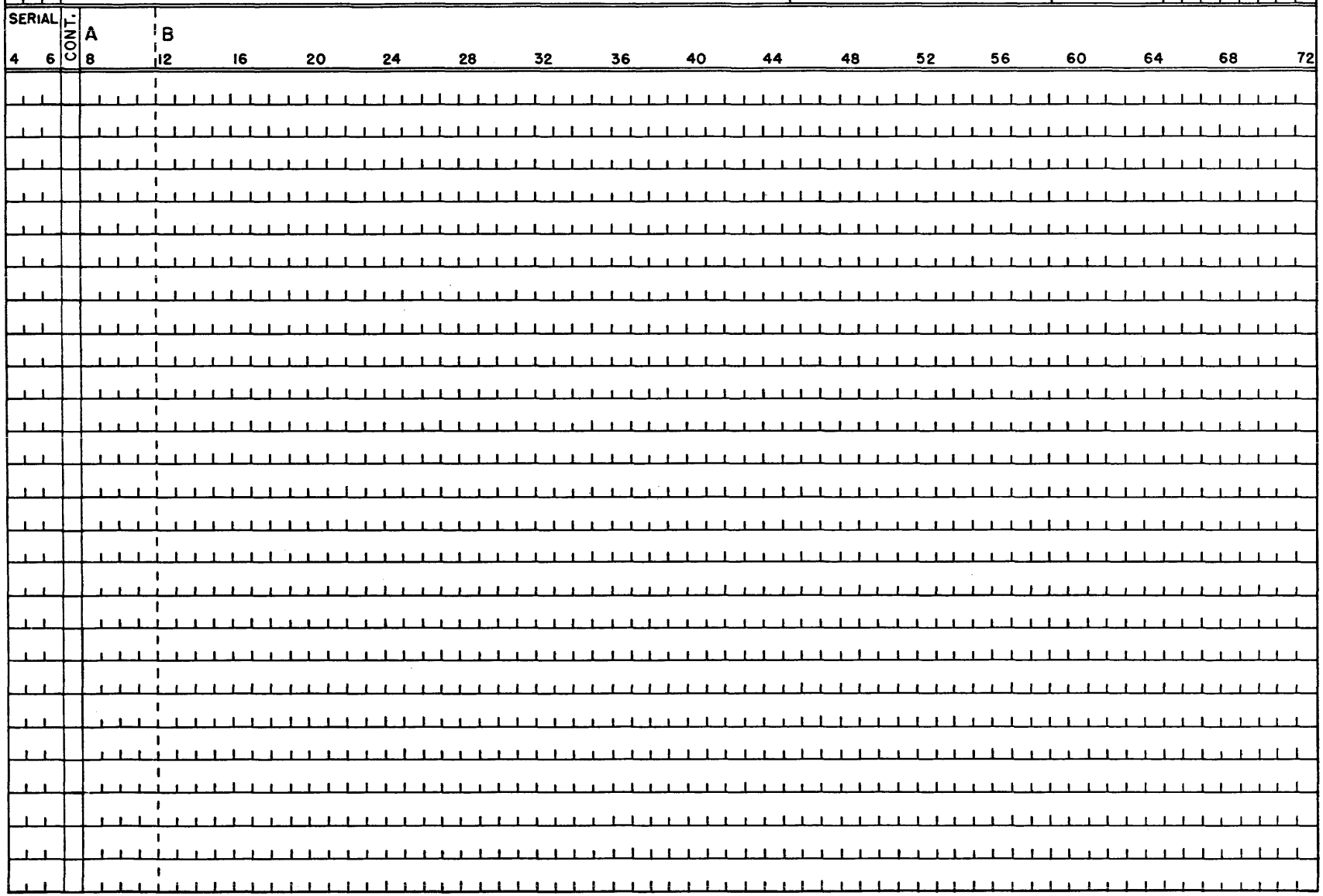


Figure 5-1.

Each line of the form is used to record the information to be punched on one IBM card. Thus, one such form, since it has 25 lines, can be used to specify the data to be punched in 25 cards. In Appendix B the reader will find a number of Program Sheets filled in with data such as might be used in a COBOL program.

It is assumed that the information in Columns 1 through 3 and Columns 73 through 80 will be identical for all 25 cards represented by the same Program Sheet. Thus, the information for these columns need be written only once on each sheet—in the spaces marked PAGE and IDENT., at the upper left and upper right, respectively, of the form. The information to be punched in Columns 4 through 72 is assumed to be variable. Thus, 69 spaces are provided on each printed line.

Sequence Number

Columns 1 through 6 are used for *sequence numbers*. These are numbers which may be assigned to the cards by the programmer to indicate the proper order of the cards. Each sequence number consists of two parts. The Page Number is shown in Columns 1 through 3, and the Serial Number is placed in Columns 4 through 6. It is standard practice to use only Columns 4 and 5 of the Serial Number when the program is initially written, to permit additions, if necessary, at a later time. The lines may be conveniently numbered from 01 to 25, making Column 6 zero. Then, if the programmer should need to insert additional material between lines already carrying serial numbers, he may use Column 6 to carry the numbers of the additional lines. Such numbers will be treated as decimal fractions—thus, 025 and 028, for example, would fall between 020 and 030.

The sequence number must consist only of numerals; letters of the alphabet and special characters may not be used. The sequence number has no direct effect as such on the program and need not be written. However, if the programmer supplies sequence numbers, the processor will check the source program cards and will indicate any errors in their sequence. When the source program is listed, the processor will assign sequence numbers in the order in which the cards were entered into the machine; these numbers may or may not correspond directly to those originally written by the programmer.

Program Identification Code

Columns 73 through 80 are reserved for any code name or group of symbols the programmer may wish to assign to the program. These are used to identify the cards as belonging to the particular program, and any characters from the COBOL character set, including the blank, may be used. The program identification code has no effect on the program.

The information entered in Columns 73 through 80 has no connection with the information entered in the Identification Division of a program, except, of course, that *all* cards punched for a program will normally carry an entry in these columns.

Continuation Indicator

Many paragraphs and entries in a program will require more than one line. In such a case, the programmer may choose to break a line between words or literals; if so, he will probably leave one or more blanks at the end of the line. In any case, it will be assumed that a blank should be understood to exist after the last character written on that line, unless the programmer indicates otherwise by placing a hyphen in Column 7 of the next line. If this hyphen does appear, however, the processor will assume that the first character of the continuation is to follow the last character written on the preceding line without an intervening space. This provision makes it possible to carry a word or literal over from one line to another without interruption.

Regardless of whether the continuation indicator is used or not, continued items must be indented in accordance with the rules specified in this chapter. If a literal

is to be carried over from one line to another, all blanks left at the end of the first line, as well as any additional blanks beginning at the point where the entry is to be continued on the next, will be counted as part of the literal.

Writing the Program

Columns 8 through 72 are used for the actual data and instructions to be entered into the computer. Two *margins* are prescribed for aligning the data in these columns. These are Margin A, placed between Columns 7 and 8, and Margin B, between Columns 11 and 12. In other words, when an item is aligned with Margin A, its first character will appear in Column 8, while an item aligned with Margin B will appear with its first character in Column 12.

In general, Margin A is used to locate major subdivisions of the program, while Margin B locates subordinate items and continuations of items from one line to another. Thus, the names of divisions, sections, and paragraphs are placed at Margin A, as are the main entries of the Data Division. Most other items are placed at Margin B.

Paragraph-names are used to designate sequences of procedure which may consist of one or more sentences, as in the Procedure Division, or one or more clauses or statements, as in the Identification and Environment Divisions. A paragraph-name may appear by itself on one line, in which case the associated text begins at Margin B on the next line. However, the text may begin on the same line as the paragraph-name. In either case, a period and at least one space should follow the paragraph-name.

The rules vary slightly for the data description entries of the Data Division. Any entry having a level indicator (FD) must begin at Margin A. Succeeding data entries may also begin at Margin A if the programmer desires. However, if he wishes to call attention to the way in which the data is organized, using the principle of indentation, he may do so. A subordinate entry—that is, one having a level-number of a numerically greater value—may be indented four spaces to the right of the starting position of the entry immediately preceding it. Indentation will not affect the meaning of the level-number; indentation is used only for its visual effect —it will appear on both the Program Sheet and the listing of the source program. Subject to this rule, the programmer may use as many indentations as he wishes. He is not required to use indentation every time a subordinate item appears; some items may be indented, some not.

When an item is carried over from one line to the next, the continued portion must begin at Margin B, unless it is a data description entry. In that case, it begins under the starting position of its own data-name. The data-name in such an entry always appears as the first item following the level indicator or the level-number. It is placed on the same line, leaving at least one space after the level indicator or level-number. Since the data-name may begin at many different positions on the line, any continuation of the same entry will have to be adjusted accordingly.

Summary of Format Rules

As noted in Chapter 2, the program must be read into the computer in the following sequence: Identification Division, Environment Division, Data Division, and Procedure Division. Page and serial numbers should be assigned accordingly.

The rules for placing instructions and data on the Program Sheet are tabulated for convenience in Figure 5-2.

Summary of Rules for Using the COBOL Program Sheet (Reference Format)

This table shows the starting position of each of the various types of entries which can be made on a COBOL Program Sheet. In this table, the word "line" should be understood to mean Columns 8 through 72, i.e., the portion of each line reserved for entries which are part of the actual COBOL program, as opposed to those items that are used to show the sequence of the entries or the identity of the program.

	Item	Divisions	Begins at	Remarks
Names	Division-Name	All	A (Col. 8)	Name is followed by a period; remainder of line must be left blank.
	Section-Name	Environment Data Procedure	A (Col. 8)	Name is followed by a space, then the word SECTION, then a period; remainder of line must be left blank.
	Paragraph-Name	Identification Environment Procedure	A (Col. 8)	Name is followed by a period and at least one space. Text may follow on same line, or at B on following line.
Data Description Entries	FD entry and other unindented entries	Data	A (Col. 8)	Each entry begins with a level indicator (FD) or a level-number. This is followed by one or more spaces and then the data-name. Each clause of the entry is separated from the preceding name or clause by one or more spaces.
	Other entries where indentation is desired	Data	4 spaces to the right of the starting position of the previous level-number or indicator	
Sentences†	Sentence beginning on same line as a paragraph-name or following the end of a preceding sentence	Identification Environment Procedure	Following the name or sentence, leaving at least one space after the period	
	All other sentences	Identification Environment Procedure	B (Col. 12)	
Continued Items	Data description entry	Data	Under the first letter of its own data-name	Breaks between lines may occur at any convenient point, leaving blank spaces at the end of the line if desired; if a word or literal is divided between two lines, a hyphen must be placed in Column 7 of the second line.
	Sentence†	Environment Identification Procedure	B (Col. 12)	
Non-Program Entries	Page number	All	Col. 1-3	These columns may be used for any purpose required by the programmer. Information in these columns will be ignored by the processor in creating the object program.
	Serial number	All	Col. 4-6	
	Program identification	All	Col. 73-80	

†Includes paragraphs of the Identification Division.

Figure 5-2.

Chapter 6: Data Division

Chapter 4 presented the concepts of the *file*, the *record*, and other items of data. It was stated that each file, record, and data item is described within a program by writing data description entries in the Data Division of the source program. Every data-name referred to in the Procedure Division except figurative constants must be described in the Data Division. Items and records are described by *Record Description entries*, and files are described by *File Description entries*. This chapter will be devoted to the details of writing the Data Division of a COBOL program.

Organization of the Data Division

The Data Division is composed of three sections, each of which consists entirely of entries, rather than paragraphs and sentences. The three sections are the File Section, the Working-Storage Section, and the Constant Section; they must appear in that order within the division. If any of the sections is not required in the particular program being written, it may be omitted entirely.

The Data Division begins with the header DATA DIVISION. Each of the sections also begins with a header; the name of the section is followed by the word SECTION, as shown below:

```
FILE SECTION.
WORKING-STORAGE SECTION.
CONSTANT SECTION.
```

The File Section consists of File Description entries and Record Description entries. The Working-Storage Section and the Constant Section are made up entirely of Record Description entries.

Entries

An entry consists of a level-number, a data-name, and one or more independent clauses. As will be seen later in this chapter, a File Description entry is identified by the level indicator FD in place of a level-number. The data-name immediately following the level-indicator is called the *subject* of the entry. All data-names other than the subject which appear anywhere in an entry must be qualified if necessary to insure uniqueness. An entry is always ended with a period. The specific rules for writing entries on the COBOL Program Sheet are given in Chapter 5.

Computer-Independent Data Descriptions

In a system such as COBOL, which is used on many different computers, it is necessary to have a way of describing data which is independent of any particular machine. This is because the internal representation of data may differ from one computer to another. For example, binary computers represent all information as a series of zeros and ones. Some decimal machines use combinations of the numbers 0 through 9. Some use still other schemes. Because of these differences, the COBOL system provides a standard way of describing the characteristics of data that will be meaningful for many kinds of computers. The COBOL system permits the programmer to think of data as it might appear on a printed form. Thus, he can deal with numeric data as if it were represented in the decimal system, regardless of how it might actually be represented for the particular computer, and he can deal with letters and other characters without regard to the codes used to represent

them. A typical binary computer, for example, might represent the letter A by the six binary digits 010001, but the programmer can think of it simply as A.

The COBOL processor, analyzing the entries in the Data Division, determines what conversions will be needed, how much space in storage should be reserved for each item, what format will be used to represent each item in storage, etc. The reader will understand more clearly how this is done after he has studied the individual clauses that make up the entries.

The File Description Entry

One File Description entry must be written for each file processed by a program. The general purpose of a File Description entry is to specify the physical characteristics of a file. It may include: (1) the mode in which the data is recorded in the file, (2) the number of characters in each record, (3) the grouping of records in the file, (4) the names and values of label records (if any) in the file, and (5) the names of data records which constitute the file.

A complete series of Record Description entries describing every item in a file must immediately follow the File Description entry for that file. For example, if a file called MASTER-CHECK-FILE contains records called SPECIAL-CHECK-RECORD and REGULAR-CHECK-RECORD, the entries describing the file might be organized as follows:

```
FD MASTER-CHECK-FILE . . . .
01 SPECIAL-CHECK-RECORD . . . .
02 NAME . . . .
                .
                .
                .
01 REGULAR-CHECK-RECORD . . . .
02 NAME . . . .
                .
                .
                .
```

The general form for a File Description entry is the following:

Option 1

FD *file-name* COPY *library-name.*

Option 2

FD *file-name* [RECORDING MODE IS *mode*]

[BLOCK CONTAINS *integer-1* { RECORD[S] / CHARACTER[S] }]

[RECORD CONTAINS [*integer-2* TO] *integer-3* CHARACTER[S]]

LABEL RECORD[S] { ARE / IS } { STANDARD / OMITTED }

[VALUE OF *data-name-1* IS *literal* [*data-name-2* IS . . .]]

DATA RECORD[S] { ARE / IS } *data-name-3* [*data-name-4* . . .] .

Option 1 is used when the desired File Description entry already exists in the library. (See Chapter 4.)

Several of the clauses in Option 2 may or may not appear, depending on the machine and the file. The use of each clause is explained below. A File Description entry is terminated with a period.

File Level Indicator and File Name

The level indicator FD identifies the beginning of the File Description entry and precedes the file name assigned by the programmer.

FD *file-name*

RECORDING MODE

The purpose of this clause is to specify the format of data as it appears in the file. The general form of the clause is:

[RECORDING MODE IS *mode*]

The RECORDING MODE clause is necessary for those computers which have more than one way of representing data. The mode in which the file is written will depend on the computer being used. For example, an IBM 7090 can read or write magnetic tapes with information coded in binary mode or in binary-coded-decimal mode, and either of these may be written in "high density" or "low density."

If the RECORDING MODE clause is omitted, it will be understood that the standard mode prescribed for the computer will be used. This standard mode, together with the other available modes and their names, will be specified in the COBOL publication for each processor.

BLOCK

Records in a file may be grouped physically in *blocks* to conserve external storage space—e.g., the length of magnetic tape required to store a file. This physical grouping does not affect the logic of the program. In many applications, however, the amount of storage conserved as a result of blocking may amount to one or more reels of tape. In such cases, the running time of the object program is reduced significantly. A block cannot contain less than one record; it must consist of one or more entire records. That is, a block and a record may be equal in size, or several records may be contained in a block. The general form of the clause specifying the size of a block is shown below:

[BLOCK CONTAINS *integer-1* {RECORD[S] / CHARACTER[S]}]

Integer-1 must be a numeric literal with an integral value. The BLOCK clause is required in all cases except two: It is not required when a block contains one, and only one, complete record. When a block size has been designated as standard for a particular system, omission of the BLOCK clause implies the use of the standard size. This size will be specified in the publications covering the various systems.

The size of a block must be stated in terms of CHARACTERS when variable-length records are used. The word CHARACTERS is an optional word. If neither RECORDS nor CHARACTERS is written, *integer-1* is assumed to be the number of characters in the block.

LABEL RECORDS

In Chapter 4, it was said that *label records* are often used at the beginning and the end of a file. They also can appear at the beginning and end of each reel of a tape file if the file is several reels in length.

The main purpose of label records is to identify a file or the reels of a file. Usually, the date the file was written is included in a label record. Also, the number of blocks of information on the reel may be indicated. At the time a file is written, label records are written containing information identifying the file. Then, when this file is used as input, this information is checked to be sure that, among other things, the right file is present and that no records in previous reels have been inadvertently skipped.

Each type of computer system has its own standard format for label records. These formats are explained in the publications covering the various COBOL processors. To specify whether label records are present in an incoming file, or should be included in an output file, the following clause is used in the File Description entry:

LABEL RECORD[S] {ARE / IS} {STANDARD / OMITTED}

This clause must always be present in a File Description entry. When describing an output file (a file being created by the program), the programmer may specify whether labels are to be present or not, according to his wishes. However, when describing an input file, if labels are present in the file, the programmer *must* write LABEL RECORDS ARE STANDARD. Thus, it is illegal to attempt to ignore existing label records by writing LABEL RECORDS ARE OMITTED.

VALUE

In certain cases, the programmer may want to specify the value of an item in a label record at the time he writes his program, just as he would specify the value of a named constant in the Constant Section. The general form of the VALUE clause is as follows:

[VALUE OF *data-name-1* IS *literal* [*data-name-2* IS . . .]]

Data-name-1 must be qualified, when necessary, but subscripting is not permitted. A figurative constant may be used in the VALUE clause where a literal is specified. Suppose a programmer wishes to have the name of a file appear in a label record of the file, so that the program can check to make sure that the proper file is present before processing is begun. Suppose, also, that the label record contains an item called NAME. If the name of the file is MASTER-FILE, the programmer could write the clause VALUE OF NAME IS 'MASTER-FILE' in the File Description entry for the file. Then, one of the following things would happen at object time:

1. If MASTER-FILE is an output file, then as the label record is written, NAME will be given the value of the non-numeric literal MASTER-FILE.
2. If MASTER-FILE is an input file (a file to be *read* by the program), then the item NAME, when read by the computer, will automatically be checked to see whether it has the non-numeric value MASTER-FILE. If not, an error will be indicated.

Because the format and content of a label record depend on the specific computer being used, all details of the rules for using the VALUE clause will be given in the publications describing the various COBOL processors.

DATA RECORDS

This clause allows the processor to cross-reference the File Description entry and the individual Record Description entries of each record whenever necessary. The general form of the clause is:

$$\underline{\text{DATA RECORD[S]}} \left\{\begin{matrix}\text{ARE}\\\text{IS}\end{matrix}\right\} \textit{data-name-3} \left[\textit{data-name-4} \ldots\right]$$

This clause is required in every File Description entry. *Data-name-3, data-name-4,* etc., must each be the subject of a Record Description entry with a level-number of 01.

The appearance of more than one data-name in this clause means that the file contains a corresponding number of different types of records. These records may be of differing sizes and formats. The order in which they are listed in the DATA RECORDS clause is not important.

It must be remembered that no two records of the same file are made available for processing at the same time. In other words, if one record is read from a file and then another record is read from the same file, the second record replaces the first record. One way to save the first record would be to move it to a work area, using the READ verb with the INTO option, as explained in Chapter 7.

COPY

This clause is used by the processor to incorporate a *prewritten* File Description entry into the Data Division of a program. When this clause is used, the processor obtains the specified entry from the library and places it in the program. Since the entire File Description entry is copied from the library, the COPY clause must appear as the only clause in the entry when this option is used. The general form of the complete File Description containing a COPY clause is given below.

$$\left[\underline{\text{FD}}\ \textit{file-name}\ \underline{\text{COPY}}\ \textit{library-name.}\right]$$

As the entry to be copied is extracted from the library and placed in the program, it retains the level indicator FD, while the subject of the COPY entry (*file-name*) replaces the subject of the library entry (*library-name*). For example, suppose the following entry exists in the library: FD OUTPUT-FILE-DESCRIPTION-ENTRY, RECORDING MODE IS . . . . This entry could be called from the library by the entry FD NEW-MASTER COPY OUTPUT-FILE-DESCRIPTION-ENTRY. The File Description entry, as it is placed in the source program, would appear as FD NEW-MASTER RECORDING MODE IS . . . .

It should be noted that the COPY clause will copy only a File Description entry, not the associated Record Description entries.

RECORD

The RECORD clause may be used to specify the size of a record in terms of the number of characters it contains. The general format of the clause is as follows:

[RECORD CONTAINS [*integer-2* TO] *integer-3* CHARACTER[S]]

Since the size of a record is completely defined by its Record Description entries, the RECORD clause is optional. If used, the following considerations apply:

Integer-2 is used to specify the minimum number of characters in the smallest record of the file, while *integer-3* indicates the maximum number of characters in the largest record. If all records in the file are of exactly the same size, only *integer-3* need be specified. Both *integer-2*, when used, and *integer-3* must be numeric literals with integral values.

The Record Description Entry

Record Description entries are the means by which items of data are described to the COBOL processor. Every item that is given a separate name must be described in a separate entry.

Like the File Description entry, the Record Description entry consists of a number of independent clauses. In the following discussion, the clauses that may be used in a Record Description entry will be covered separately. Special rules for writing entries in the Working-Storage and Constant Sections will be presented later in this chapter.

The general form of a Record Description entry is given below. In general, the order in which clauses are written in an entry is not critical (except where specifically noted). The entry must begin, however, with a level-number and name.

Option 1

level-number data-name [REDEFINES . . .] COPY . . .

Option 2

level-number {FILLER / *data-name*} [REDEFINES . . .] [SIZE . . .]
[USAGE . . .] [OCCURS . . .] [SIGNED] [SYNCHRONIZED . . .]
[POINT . . .] [CLASS . . .] [PICTURE . . .] [JUSTIFIED . . .]
[ZERO SUPPRESS . . .] [BLANK . . .] [VALUE . . .].

Those clauses which begin with SIGNED, SYNCHRONIZED, POINT, PICTURE, JUSTIFIED, ZERO SUPPRESS, and BLANK may appear *only* in an entry which describes an *elementary* item.

Level-Number and Name

Every Record Description entry must begin with a level-number and a name, as shown in the following general format:

level-number { FILLER / *data-name* }

The level-number is used to show the relation of the item to other items. The general concept of levels is discussed in Chapter 4. Level-numbers must be chosen from the integers 1 (or 01) through 49, 77, and 88. Single-digit level-numbers (1-9) must be preceded by either a zero or a blank. Level-number 1 (or 01) is applicable only to records, not to just a part of a record. Level-number 77 has a special use in connection with constants and work areas; this will be explained later in this chapter. Level-number 88 is used to denote condition-names; entries having this level-number must follow immediately after the entry describing the data-name with which the condition-names are associated.

Every Record Description entry must have a subject, that is, a name which is to be assigned to the item being described. This name must appear immediately after the level-number. The word FILLER may be thought of as a special kind of name which must be used if the entry describes an item to which no name has been given.

The subject of an entry must never be in qualified or subscripted form. Even if *data-name-1* is not unique, it must not be qualified at this point in an entry, because its level-number and placement in relation to other entries automatically supply qualification. Thus, it is correct to write:

```
05 VENDOR . . . .
   06 NAME . . . .
05 RETAILER . . . .
   06 NAME . . . .
```

If an item is given the key word FILLER as a name, that name must never be referred to in the Procedure Division. When FILLER is used to name an item, the only clause required in the entry is SIZE (or PICTURE).

SIZE

The purpose of the SIZE clause is to specify, in terms of the number of characters or digits, the size of an item of data. The general form of this clause is as follows:

[SIZE IS *integer-1* [{ CHARACTER[S] / DIGIT[S] }]]

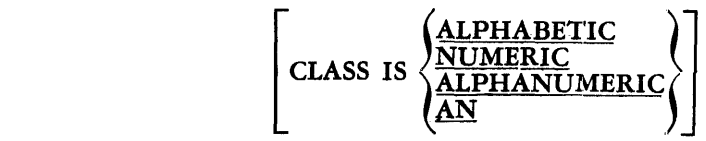
The size of an item is independent of its format within the computer. All DIGITS are considered to be decimal, regardless of the number base used by the computer. All alphabetic and alphanumeric CHARACTERS are considered as though they were in printed or typewritten form. Arithmetic signs belonging to items used in computations are treated as operators (operational signs) and are never counted in determining the size of an item. However, when the alphanumeric characters + and − are associated with non-computational items, as in edited reports, they are counted in determining the size.

Decimal points in numeric items used for computation are always "assumed." That is, the decimal point does not occupy an actual character position. Therefore, such a decimal point is not counted in determining the size of a numeric item. Further details concerning assumed decimal points will be found in the discussions of the PICTURE and POINT LOCATION clauses.

If the entry describes an *elementary* item, it must have either a SIZE clause or a PICTURE clause. If the entry describes a *group*, on the other hand, neither clause is required, but if the size is specified at a group level, it must be equal to the sum of the sizes of the elementary items within the group. If both PICTURE and SIZE appear, they must not contradict each other.

CLASS

Each computer has its own internal character set, which may or may not be limited to the characters of the COBOL character set. Each such set can be subdivided into numeric, alphabetic, and alphanumeric (alphameric) characters. Thus it can be said that any data item can be classified as numeric, alphabetic, or alphanumeric, depending on the characters it contains. The purpose of the CLASS clause is to indicate to the processor the class of the data as defined in this manner. The basic format is as follows:



AN is an abbreviation of ALPHANUMERIC.

These three classes may be described as follows:

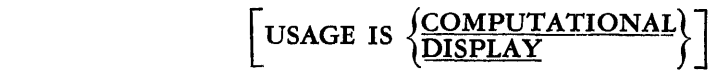
Class	Description
NUMERIC	Consists entirely of digits (0-9); may also contain an operational plus or minus sign; an actual decimal point is a non-numeric character and is not permitted.
ALPHABETIC	Consists entirely of letters of the alphabet; may also contain one or more spaces (blanks).
ALPHANUMERIC	Consists of any characters from the machine's character set. May be wholly numeric or wholly alphabetic.

The CLASS clause may be written at any level. If a CLASS clause appears in an entry describing an item which is a group, it specifies the class for each item within the group; it must not be contradicted by the CLASS descriptions (if any) of the members of the group. It is not considered a contradiction for an ALPHANUMERIC group to contain an item which is either NUMERIC or ALPHABETIC, since the ALPHANUMERIC class contains the other two classes.

If an entry contains a PICTURE clause, a CLASS clause is not necessary. However, if both are used, they must be compatible.

USAGE

Data may be used within a computer in various ways. In general, data processing operations are of two principal kinds: (1) operations involving computation, and (2) operations which ultimately affect the form in which the data is edited and/or presented—i.e., "displayed"—for reading. In order to improve the efficiency of the object program, it is desirable to indicate to the COBOL processor whether an item will be used most frequently for COMPUTATIONAL or for DISPLAY purposes. The USAGE clause serves this purpose. Its basic format is as follows:



The USAGE clause may be written at any level. If it is written at a group level, it specifies the USAGE of all items contained in the group and must not be contradicted by a lower-level entry in that group. If the USAGE of an item is not specified, it is assumed to be DISPLAY.

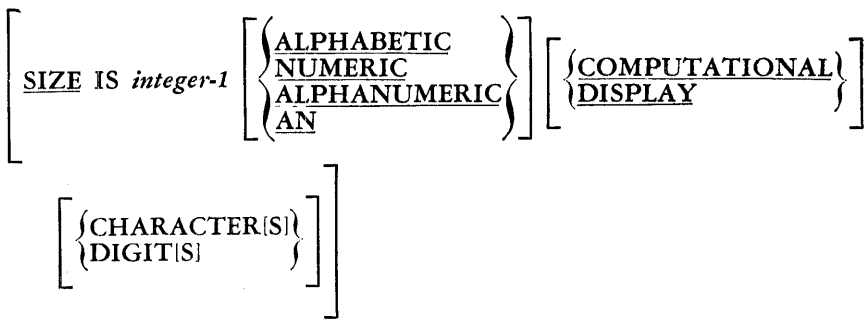
The USAGE specified in this clause does not limit the operations of the object program, nor does it restrict in any way the operation of any verb on the data. However, it may affect the way in which the data is represented inside the computer, and this in turn will affect the efficiency of the object program. The programmer should therefore indicate the manner in which each item of data will most frequently be used. Designation of one USAGE does not prevent an item from being used according to the other. For example, COMPUTATIONAL usage will be specified for most NUMERIC items; this description, however, does not prevent their being displayed. The converse is also true. In effect, the USAGE clause provides a description of the internal representation of data. It will not convert data from one representation to another except in conjunction with Procedure Division clauses.

If an item is specified as being COMPUTATIONAL, it *must* be of the NUMERIC class, and therefore the CLASS clause may be omitted.

Because the USAGE clause affects the way in which data is represented within the computer, a more complete discussion of its effects will be found in the publications covering the various processors.

Combining the SIZE, CLASS, and USAGE Clauses

If the programmer desires, he may combine the SIZE clause with the CLASS clause and/or the USAGE clause in a special form. This allows him to specify the SIZE, CLASS, and USAGE of an item by writing one clause instead of three. The general form for combining these clauses is:



This clause can be used to specify:

1. The SIZE and CLASS of an item.
2. The SIZE and USAGE of an item.
3. The SIZE, CLASS, and USAGE of an item.

The rules governing the use of this clause are the same as the rules pertaining to the use of the separate SIZE, CLASS, and USAGE clauses as discussed previously. The order in which the CLASS and USAGE are specified is not important; they may appear in either sequence.

EXAMPLES

```
SIZE IS 7  NUMERIC DISPLAY DIGITS
SIZE IS 7  DISPLAY NUMERIC DIGITS
SIZE IS 15 DISPLAY CHARACTERS
SIZE IS 10 NUMERIC CHARACTERS
```

POINT LOCATION

The POINT clause is used to specify the position of an assumed decimal point if a PICTURE clause is not used in describing a numeric item. For an explanation of assumed decimal points, see the discussion of PICTURE. The general form of the POINT clause is given below.

$$\left[\underline{\text{POINT}}\ \text{LOCATION IS}\ \left\{\begin{matrix}\underline{\text{LEFT}}\\ \underline{\text{RIGHT}}\end{matrix}\right\}\ \textit{integer-3}\ \text{PLACE[S]}\right]$$

An actual decimal point cannot be specified by means of a POINT clause; a PICTURE clause must be used instead. If neither a PICTURE clause nor a POINT clause is used with a NUMERIC item, the item will be assumed to be an integer.

Integer-3 must be a numeric literal with an integral value. The assumed decimal point will be located *integer-3* digit positions to the left or right of the right end of the item. Thus, the clause POINT IS LEFT 4 PLACES would cause the number –3597421 to be treated as –359.7421, whereas the clause POINT IS RIGHT 4 PLACES would cause the same number to be treated as –35974210000. This clause can be used only to describe an elementary item.

SIGNED

The purpose of this clause is to indicate that an elementary item has an *operational sign*. An operational sign is a character or code that indicates to the computer whether a value is positive or negative. Actual operational signs cannot be specified in the COBOL language; they are incorporated in the actual data for interpretation by the machine at object time. Sign conventions vary from one computer to another and will be specified in the publications covering the various processors. The general form of the clause is:

$$\left[\underline{\text{SIGNED}}\right]$$

A SIGNED item must be NUMERIC; thus, when the SIGNED clause is used, the CLASS clause may be omitted. When an operational sign is specified in a PICTURE clause, the SIGNED clause is not required. The SIGNED clause can be used only in describing an elementary item. An item for which editing is specified (in either an editing clause or a PICTURE clause) must not have an operational sign. Therefore, the SIGNED clause cannot be used in an entry containing any of the editing options (ZERO SUPPRESS, CHECK PROTECT, FLOAT DOLLAR SIGN, and BLANK WHEN ZERO). An operational sign is not counted as a character in determining SIZE.

VALUE

The VALUE clause is used to specify the value of a named constant, a condition-name, or an item in a work area. The general form of the VALUE clause is given below.

$$\left[\underline{\text{VALUE}}\ \text{IS}\ \textit{literal}\right]$$

Literal may be any numeric or non-numeric literal or figurative constant. It must be of the same CLASS specified for this item. If the VALUE clause is used to define the value of a condition-name, no other clause is needed in the same entry. The VALUE clause may be used in an entry in the Working-Storage Section to specify the value

of a condition-name or to specify the initial value of an item. In the File Section, it may be used only to describe condition-names, and any other use of this clause in that section is incorrect. The VALUE clause is used in the Constant Section to specify the value of a named constant.

When a VALUE clause appears in conjunction with a scaled item, a zero for each scaled position in the item must be included in the literal defining the VALUE. (See the discussion of the character P as used in the PICTURE clause.)

If the VALUE clause is used at the group level in the Working-Storage or Constant Sections, it must not appear in the entries within the group. Also, the VALUE clause must not appear in an entry containing an OCCURS clause or in an entry which is subordinate to an entry containing an OCCURS clause.

EXAMPLES

The following is an example of the use of the VALUE clause to show the value of three condition-names associated with a data-name:

```
05  MARITAL-STATUS SIZE 1 COMPUTATIONAL.
    88  SINGLE VALUE IS 1.
    88  MARRIED VALUE IS 2.
    88  DIVORCED VALUE IS 3.
```

The following examples might be used to specify the values of named constants or the initial values of items in work areas:

- VALUE IS 6 (CLASS is NUMERIC.)
  VALUE 'SIX' (CLASS is ALPHABETIC or ALPHANUMERIC.)
  VALUE IS '6.0' (CLASS is ALPHANUMERIC.)
  VALUE IS SPACE (CLASS is ALPHABETIC or ALPHANUMERIC.)
  VALUE QUOTE (CLASS is ALPHANUMERIC.)
  VALUE 'JOHN' (CLASS is ALPHABETIC or ALPHANUMERIC.)
- VALUE IS ALL '32' (CLASS is ALPHANUMERIC.)

PICTURE

The various clauses of a Record Description entry are used to describe certain characteristics of data so that the computer can process it efficiently. The PICTURE clause provides an alternative, and more compact, way of specifying most of the same information. Each character of a data item is represented by a code chosen to describe its principal characteristics. The "picture" of an item may be thought of as a kind of "blueprint" of it. The general form of the clause is as follows:

[PICTURE IS *any allowable combination of characters and symbols as described below*]

Only elementary items can be described by a PICTURE clause. When this clause is used, no other clause is required in the same entry, except that the REDEFINES clause is necessary when redefinition must be specified. If desired, however, the other Record Description clauses may also be written, provided they do not contradict the PICTURE clause.

Speaking generally, the PICTURE clause indicates the SIZE of an item, the CLASS, the presence or absence of an operational sign and/or an assumed decimal point, and provides additional information which would otherwise have to be specified in other Record Description clauses.

In addition, the PICTURE clause can be used to specify the *editing* of data. Editing may be described as an alteration of the format and/or punctuation of an item, usually for such purposes as improving readability or protecting it against unauthorized alteration. Editing involves the suppression of certain characters and/or the addition of others. For example, after computation, the digits representing a man's pay might be 0012531. However, they would be much more readable on a paycheck in an edited form, such as $**125.31; moreover, the use of the asterisks would hamper an attempt to alter the amount. Editing of data always requires moving it to an item for which the proper editing symbols have been specified.

In the following discussion, each character which may appear in a PICTURE is presented. Because the choice of characters in any given PICTURE depends on the type of data item being described, the characters will be grouped for discussion according to the type of data item they describe.

Numeric Items

A numeric item is an item which may contain only the numerals 0 through 9 and an operational sign. As will be seen below, a numeric item may also have an assumed decimal point associated with it. The PICTURE of any numeric data item may contain combinations of only the following characters: 9, V, P, and S. An explanation of each of these characters and their uses is given below.

Character	Meaning and Use
9	A 9 indicates that the character position will always contain a NUMERIC character.
V	A V indicates the position of an *assumed decimal point.* Since a numeric item cannot contain any character other than numerals and an operational sign, the actual decimal point (the special character period) cannot appear. Therefore, an assumed decimal point is used to provide the processor with information concerning the alignment of items involved in computation. An assumed decimal point, thus, does *not* occupy a character position at object time and is not counted in the size of an item. For example, if a data item is described as having a PICTURE of 99V9 and the digits 123 are moved to it, the value would be treated in calculation as 12.3, but the size of the item would be three characters, not four. If it were printed, it would print as 123 because the decimal point character is not actually present.
S	The character S is equivalent in meaning and use to the SIGNED clause; it indicates the presence of an *operational sign.* If used, it must always be written as the leftmost character of the PICTURE. It is *not* counted in the size of an item.
P	The character P indicates an *assumed decimal scaling position.* It is used to specify the location of an assumed decimal point when the point is not within the number as it appears in *input data.* In effect, the item will be treated as if a zero were substituted for each P and the decimal point were placed "outside" the last P—i.e., to the right if the zeros are on the right, to the left if the zeros are on the left. The character V may be used or omitted as desired. If it is used, it must be placed in the position of the assumed decimal point. For example, an item composed of the digits 123 would be treated by an arithmetic procedure statement as 123000 if the PICTURE were 999PPPV or as .000123 if the PICTURE were VPPP999. The character P is never considered as part of the size of an item; in the above examples, the size would be three characters.

Character	Meaning and Use

When P is used in a PICTURE in an entry in the Working-Storage Section or the Constant Section, and if the entry also includes a VALUE clause, the following rule applies: A zero must appear in the VALUE clause for every character position described by a P in the PICTURE clause.

The following examples show the PICTURE of an item, a corresponding VALUE clause which might appear in the same entry, and the actual digits which would be placed in storage as a result of these clauses.

PICTURE	VALUE Clause	Actual Characters Placed in Associated Storage Area
999PPP	VALUE IS 146000	146
PPP99	VALUE IS .0003	30
VP9	VALUE IS .01	1
999PPPV	VALUE IS 1234000	234
VPPP99	VALUE IS .000372	37

Note that the last two examples would each produce an error message when the object program is created and would cause truncation at object time. In general, if a VALUE clause and an associated PICTURE do not specify the same number of digits, truncation or zero fill will occur according to the rules of the MOVE verb given in Chapter 7.

Alphabetic Items

In the discussion of CLASS, an ALPHABETIC item was defined as one which could contain only the letters of the alphabet and the space. The PICTURE of an ALPHABETIC item can contain only the character A.

Character	Meaning and Use
A	The character A, when used in a PICTURE, indicates that the character position will always contain either a letter or a space.

Alphanumeric Items

An ALPHANUMERIC item has been defined as an item which may contain any character in the computer's character set. However, it is often convenient to think of ALPHANUMERIC items as being divided into two types: *non-report* items and *report* items. Non-report items are items for which editing is not specified. Report items are items for which editing has been specified.

Non-Report Items

The PICTURE of an ALPHANUMERIC *non-report* item may contain only the characters 9, A, and X. The characters 9 and A have been discussed above.

Character	Meaning and Use
X	The character X, when appearing in a PICTURE, indicates that the character position may contain any character in the computer's character set. For example, the PICTURE AAAXXXX indicates that the described item has a size of seven characters, that the first three characters will always be alphabetic, and that the last four characters may be any characters.

Report Items

It may be desirable to edit data which is being prepared for printing. Editing involves the insertion of certain characters and/or the suppression of others. Editing of data is accomplished by moving the data to a *report* item. A report item is an ALPHANUMERIC item governed by the following rules:

1. Any data which is moved to a report item is automatically altered according to the editing specifications given in the Record Description entry corresponding to the item. Editing may be specified by means of a PICTURE or by using the editing clause discussed later in this chapter.
2. A report item can receive only data which is numeric in content.

The characters which may appear in the PICTURE of a report item are shown below.

9 V , . + − Z * CR DB B 0 $

All the characters in the PICTURE of a report item, with the exception of V, must be counted in determining the size of the item. The uses of 9 and V have been discussed above. The remainder of the characters will be explained in three groups, zero suppression, insertion, and replacement characters.

Zero suppression and replacement characters are used to suppress and/or replace characters in accordance with the rules given in this section. Two general rules apply to these characters, as follows: (1) Except in the cases of 0 and B, suppression and/or replacement terminates with the character immediately preceding the first digit other than 0, or the decimal point, whichever is encountered first; e.g., zeros following a significant digit will not be suppressed or replaced. (2) If all data character positions in a PICTURE reserved for source data (as opposed to those additional positions used for insertion characters) contain suppression and/or replacement characters (other than 0), then all characters will be replaced by blanks if the value of the data is zero. Note that this rule is equivalent in effect to the BLANK clause.

ZERO SUPPRESSION CHARACTER

Character	Meaning and Use
Z	The character Z specifies *zero suppression* of the indicated characters. Zero suppression is the process of replacing unwanted left-hand zeros by blanks. The following table indicates the effect of zero suppression:

Source Item	Editing PICTURE	Edited Item
12345	ZZZ99	12345
00123	ZZZ99	123
00100	ZZZ99	100
00000	ZZZ99	00
00100	ZZZZZ	100
00000	ZZZZZ	

A Z must never be preceded by a 9, a B, or a 0.

INSERTION CHARACTERS

An *insertion* character is one which is inserted into a report item. An insertion character does not take the place of any data; it appears in addition to the information moved to the item. The insertion characters are $, . + − CR DB. When any of these characters is used, the size of the report item must be larger than the maximum number of digits which might be moved to the item. This principle is illustrated in the discussion of the dollar sign below.

Character — **Meaning and Use**

$ — The single dollar sign, placed in the leftmost position of a PICTURE, specifies that a dollar sign character is to be placed in that position in the data, as illustrated in the following table:

Source Item	Editing PICTURE	Edited Item
123	$999	$123
012	$999	$012
012	$ZZZ	$ 12
000	$ZZZ	
010	$ZZZ	$ 10

Note that the PICTURE of the item specifies four character positions; however, a maximum of three digits of data can be moved to the item.

— — If the minus sign is written as either the first character or the last character of a PICTURE, a *display* minus sign (as opposed to an operational sign) will be inserted into the indicated character position when the value of the item is negative. If the value of the item is not negative, a blank will be inserted. Consider the following:

Source Item†	Editing PICTURE	Edited Item
1$\bar{2}$	−99	−12
12	−99	12
1$\bar{2}$	99−	12−
12	99−	12
0$\bar{0}$	99−	00−
00	99−	00

+ — If the plus sign is written either as the first character of a PICTURE or as the last, a *display* sign will be placed in the indicated character position. If the value of the item is negative, a minus sign will appear; otherwise, a plus sign will be inserted. If an item is unsigned, it is assumed to be positive. The following table illustrates the above principle:

Source Item†	Editing PICTURE	Edited Item
1$\bar{2}$	+99	−12
1$\dot{2}$	+99	+12
1$\bar{2}$	99+	12−
1$\dot{2}$	99+	12+
0$\bar{0}$	99+	00−
0$\dot{0}$	99+	00+

†A sign over the units position of a number indicates an operational sign.

Character	Meaning and Use
,	The comma, when used to describe a character position, will be inserted at the indicated position in the data being edited. For example, the PICTURE 9,999 would cause 7461 to become 7,461 after editing. The comma itself will be suppressed if zero suppression has caused the elimination of all digits to the left.
.	This character represents an *actual decimal point*. When used to describe a character position: 1. The data being edited is aligned by decimal point. 2. An actual decimal point will appear in the indicated character position. Thus, the integer 7531 would appear as $7,531.00 if the notation $9,999.99 were used as the editing PICTURE. Unlike the assumed decimal point, the actual decimal point occupies a character position and is counted in determining the size of an item. A PICTURE may never contain more than one decimal point, assumed or actual.
CR and DB	The *credit* and *debit* symbols CR and DB may appear only at the right end of a PICTURE. These symbols occupy two character positions. When the value of the described item is negative, the specified symbol will be placed at the right end of the item. If the value of the item is positive, these character positions will contain blanks. For example, the PICTURE $99.99CR will cause $63\bar{2}5$ to become $63.25CR and $63\overset{+}{2}5$ to become $63.25 after editing.
0 (zero)	The character 0 (zero) will cause a zero to be inserted in the indicated character position. For example, if the digits 123456 are to be moved to an item with a PICTURE of 999000999, the item will appear as 123000456.
B	The character B specifies that a blank will be inserted in the indicated character position. For example, if the digits 121456 are to be moved to an item with a PICTURE of 99B99B99, the item will appear as 12 14 56.

REPLACEMENT CHARACTERS

Several of the characters used in PICTURE specify that, at object time, certain digits will be replaced by other characters much in the same way that the Z specifies the replacement of leftmost zeros with blanks. The list of *replacement* characters consists of: *, 0, B, the floating dollar sign, the floating minus sign, and the floating plus sign.

Character	Meaning and Use
*	The asterisk is used to indicate *check protection*, i.e. the suppression of each specified zero on the left and its replacement by an asterisk. The following table illustrates the use of the asterisk.

Character	Meaning and Use		
	Source Item	Editing PICTURE	Edited Item
	12345	***99	12345
	00123	***99	**123
	00100	***99	**100
	00000	***99	***00
	00000	*****	
	00100	*****	**100

An asterisk can be preceded only by a dollar sign, a plus sign, a minus sign, a decimal point, or a comma.

The floating dollar sign

Zero suppression with a *floating dollar sign* is specified by placing a dollar sign in each leading numeric character position to be suppressed. A dollar sign will be placed in the rightmost position in which suppression by a dollar sign is to occur. The following table illustrates the principle:

Source Item	Editing PICTURE	Edited Item
123	$$99	$123
012	$$99	$12
001	$$ZZ	$ 1
000	$$$$	

The floating minus sign

Zero suppression with a *floating minus sign* is specified by placing a minus sign in each leading numeric character position to be suppressed. If the value of the item is negative, a minus sign will be placed in the rightmost position in which suppression by a minus sign is to occur. If the value of the item is positive or zero, a blank will be inserted instead of a minus sign. The following table illustrates the principle:

Source Item	Editing PICTURE	Edited Item
123	--99	123
012	--99	-12
001	--ZZ	- 1
000	----	

All floating minus signs must be the leftmost characters in a PICTURE.

The floating plus sign

Zero suppression by means of a *floating plus sign* is specified by placing a plus sign in each leading numeric character position to be suppressed. If the value of the item is negative, a minus sign will be placed in the rightmost position in which suppression is to occur; if the value of the item is not negative, a plus sign will be inserted instead. The following examples illustrate the effect of the floating plus sign:

Source Item	Editing PICTURE	Edited Item
123	++99	+123
012	++99	+12
01$\bar{2}$	++99	-12
00$\bar{1}$	++99	-01
000	++++	

All floating plus signs must be the leftmost characters in a PICTURE.

Reserved Characters

The characters J and K are reserved for special uses in a PICTURE and are explained, as appropriate, in the publications covering the various processors.

General Notes on the PICTURE Clause

1. When an integer is placed in parentheses immediately following a PICTURE character, it indicates the number of successive times that character is to be present. For example, the notation P(4) 9(10) is equivalent to PPPP9999999999 and will be interpreted in the same way. The parentheses must follow the indicated symbol without an intervening space.
2. The number of characters in a PICTURE must not exceed 30. For example, $$ZZ.99 is a PICTURE containing seven characters, 9V9 contains three characters and 9(1000) contains seven characters. Thus, the number of characters in a PICTURE may be different than the number of character positions described by the PICTURE.

Examples of the PICTURE Clause

The following are examples of applications of PICTURE clauses which do not contain editing symbols. A sign over the units position of a number indicates an operational sign as opposed to a display sign.

Non-Editing Applications

If PICTURE is:	and the characters in the item are:	then the item will be used in procedures as:	and its class will be:
99999	12345	12345	NUMERIC
999V99	12345	123.45	NUMERIC
S999V99	1234$\bar{5}$	123.4$\bar{5}$	NUMERIC
S9(3)V9(2)	1234$\bar{5}$	123.4$\bar{5}$	NUMERIC
XXXXX	12345	12345	ALPHANUMERIC
AAAAA	ABCDE	ABCDE	ALPHABETIC
XXXXX	ABCDE	ABCDE	ALPHANUMERIC
999X99	123.45	123.45	ALPHANUMERIC
999AA	123AB	123AB	ALPHANUMERIC
999XX	123AB	123AB	ALPHANUMERIC
XXXAA	123AB	123AB	ALPHANUMERIC
XXXXX	123AB	123AB	ALPHANUMERIC
9(3)A(2)	123AB	123AB	ALPHANUMERIC
99PPP	12	12000	NUMERIC
99PPPV	12	12000	NUMERIC
P(3)9(2)	12	.00012	NUMERIC
VP(3)9(2)	12	.00012	NUMERIC

The examples which follow illustrate the use of PICTURE to edit data. In each example a movement of data is implied, as indicated by the column headings.

Editing Applications

Source Area		Receiving Area	
PICTURE	DATA	PICTURE	EDITED DATA
99999	12345	$ZZ,ZZ9.99	$ 1 2 , 3 4 5 . 0 0
99999V	00123	$ZZ,ZZ9.99	$ 1 2 3 . 0 0
9(5)	00100	$ZZ,ZZ9.99	$ 1 0 0 . 0 0
9(5)V	00000	$ZZ,ZZ9.99	$ 0 . 0 0
9(5)	00000	$ZZ,ZZZ.99	$. 0 0
9(5)	00000	$ZZ,ZZZ.ZZ	
999V99	12345	$ZZ,ZZ9.99	$ 1 2 3 . 4 5
V99999	12345	$ZZ,ZZ9.99	$ 0 . 1 2
9(5)	12345	$**,**9.99	$ 1 2 , 3 4 5 . 0 0
9(5)	00123	$**,**9.99	$ * * * 1 2 3 . 0 0
9(5)	00000	$**,***.99	$ * * * * * * . 0 0
9(5)	00000	$**,***.**	
9(5)	00000	$**,***.ZZ	
99V999	12345	$**,**9.99	$ * * * * 1 2 . 3 4
V99999	12345	$**,**9.99	$ * * * * * 0 . 1 2
9(5)	12345	$$$,$$9.99	$ 1 2 , 3 4 5 . 0 0
9(5)	00123	$$$,$$9.99	$ 1 2 3 . 0 0
9(5)	00000	$$$,$$9.99	$ 0 . 0 0
9(5)	00000	$$$,$$$.ZZ	
9999V9	12345	$$$,$$9.99	$ 1 , 2 3 4 . 5 0
V9(5)	12345	$$$,$$9.99	$ 0 . 1 2
S99999V	12345-	-ZZZZ9.99	- 1 2 3 4 5 . 0 0
S9(5)V	12345+	-ZZZZ9.99	1 2 3 4 5 . 0 0
S9(5)	00123-	-ZZZZ9.99	- 1 2 3 . 0 0
S99999	12345+	ZZZZ9.99-	1 2 3 4 5 . 0 0
S9(5)	12345-	ZZZZ9.99-	1 2 3 4 5 . 0 0 -
S9(5)	00123+	-------.99	1 2 3 . 0 0
S9(5)	00001-	-------.99	- 1 . 0 0
S9(5)	12345+	+ZZZZZ.99	+ 1 2 3 4 5 . 0 0
S9(5)	12345-	+ZZZZZ.99	- 1 2 3 4 5 . 0 0
S9(5)	12345+	ZZZZZ.99+	1 2 3 4 5 . 0 0 +
S9(5)	12345-	ZZZZZ.99+	1 2 3 4 5 . 0 0 -
S9(5)	00123+	++++++.99	+ 1 2 3 . 0 0
S9(5)	00001+	++++++.99	+ 1 . 0 0
9(5)	00123	++++++.99	+ 1 2 3 . 0 0
9(5)	00123	-------.99	1 2 3 . 0 0
9(5)	00000	++++++.ZZ	
9(5)	00000	-------.ZZ	
9(5)	12345	BB999.00	3 4 5 . 0 0
9(5)	12345	00099.00	0 0 0 4 5 . 0 0
S9(5)	12345-	$$$$$$.99CR	$ 1 2 3 4 5 . 0 0 C R
S9(5)	12345+	$$$$$$.99CR	$ 1 2 3 4 5 . 0 0

The Editing Clause

This clause allows the programmer to specify certain kinds of editing without using the PICTURE clause. Zero suppression, check protection, and the floating dollar sign are available by the use of this clause. (For a discussion of these editing terms, see PICTURE.) The general form of the editing clause is shown below:

This clause can be applied only to elementary items and to items which edit NUMERIC values.

ZERO SUPPRESS will replace all left-hand zeros with blanks until either a digit other than a zero or a decimal point (assumed or actual) is encountered.

CHECK PROTECT will replace all left-hand zeros with asterisks until either a digit other than zero or a decimal point (assumed or actual) is encountered.

FLOAT DOLLAR SIGN will cause all left-hand zeros to be suppressed and a dollar sign to be placed immediately to the left of either the first non-zero digit or the decimal point.

If the LEAVING option is used, the suppression or insertion will stop at such a point as to "leave" the specified number of places to the left of the decimal point. For example, a five-digit number described by the clause CHECK PROTECT LEAVING 3 PLACES would edit the number 7 so that it would appear as **007 when printed. *Integer* must be a numeric literal with an integral value.

The editing clause cannot be used to specify zero suppression to the right of a decimal point; the BLANK clause is used for that purpose. If a PICTURE clause is used in the same entry as an editing clause, the two must not be contradictory.

BLANK

The purpose of the BLANK clause is to make an item blank when its value is equal to zero. The general form of the clause is:

When this clause is present, the item being described will be filled with blanks (spaces) whenever the value of the item is zero. When the zero condition exists, all editing specifications in a PICTURE or editing clause will be overridden in favor of inserting all blanks. This clause can be used only with an elementary item.

JUSTIFIED

An item of data may be moved within the computer by means of a MOVE statement or as a result of computation or some other operation. If the location to which it is moved is larger in size than the data itself, it may be necessary to specify the position the data is to occupy in its new location. In the absence of instructions to the contrary, NUMERIC data will be "right justified" under these circumstances, unless an assumed decimal point has been explicitly specified for the item; in that case, decimal point alignment occurs. When an item is right justified, its rightmost character will be placed in the rightmost position of the new location, and any unused positions at the left will be filled with zeros. ALPHABETIC and ALPHANUMERIC data, on the other hand, will be "left justified" in the absence of instructions to the contrary and any unused character positions at the right will be filled with blanks.

If the programmer wishes to specify right justification in lieu of left justification, or the reverse, he may do so by means of the JUSTIFIED clause, which has the following form:

```
[JUSTIFIED {LEFT }]
           {RIGHT}
```

If the data is NUMERIC and an assumed decimal point has been explicitly specified, this clause *must not* be used.

SYNCHRONIZED

In many data processing systems, data is stored in machine "words" of a fixed length. Thus, it is possible for an item to be shorter than the space allowed for it. This means that there may be unoccupied "pieces" of words which are filled with non-significant characters, such as zeros or blanks, depending on the system. However, it may be desirable to fill all available space with data, so that more than one item may be placed in one machine word. This is known as "packing."

If the SYNCHRONIZED clause is used, the processor will understand that the item being described is *not* packed. The clause is used, therefore, to describe the "unpacked" arrangement of items within input records in the FILE SECTION, and to specify that items within output records are not to be packed. It can also be used to describe items in the Working-Storage and Constant Sections.

The general form of the SYNCHRONIZED clause is as follows:

```
[SYNCHRONIZED {LEFT }]
              {RIGHT}
```

If SYNCHRONIZED LEFT is specified, the leftmost character of the described item will occupy the left-hand position of the next machine word. This may mean that the right-hand portion of the preceding word is unoccupied. Note that the right-hand portion, if any, of the last machine word which the item requires will contain the first portion of the next item unless the latter is described as synchronized.

If SYNCHRONIZED RIGHT is specified, the rightmost character of the item will occupy the right-hand end of a word. If the item is not a multiple of full machine words in length, the left-hand portion of the first word in which it is contained will be unoccupied.

If synchronization is not specified, successive items of data are packed. Packing makes efficient use of storage space, but it may make each item relatively inaccessible and cause an increase in the running time of the object program. Therefore, if an item is referred to often, it may be advisable to synchronize it so that it can be obtained in less time.

OCCURS

The OCCURS clause is used to describe a sequence of data items of the same format, such as might appear in the form of a table. A single item which is part of such a sequence may be referred to in a procedure statement only by the use of subscripting. A general discussion of subscripting and the OCCURS clause is given in Chapter 4. (See also the discussion on constructing tables of constants, at the end of this chapter.) The general form of the OCCURS clause follows:

```
[OCCURS integer-2 TIME[S]]
```

The OCCURS clause must not be used in a Record Description entry having a level-number of 1 (or 01), 77, or 88. *Integer-2* must be a positive numeric literal having an integral value greater than zero.

REDEFINES

It is sometimes necessary to "overlap" items in storage, i.e., to use the same storage area for different items at different times. For example, suppose a work area called REFUND-WORK-AREA is needed in a program and another work area, BILLING-WORK-AREA, is used later in the same program. Normally, each area would be described separately in the Working-Storage Section, and each would occupy different portions of storage. However, if the programmer knows that REFUND-WORK-AREA is never used when BILLING-WORK-AREA is used, he may use the REDEFINES clause and cause both items to occupy the same physical area in storage. The general form of the REDEFINES clause is given below.

---

level-number data-name-1 [REDEFINES *data-name-2*]

As the format indicates, the REDEFINES clause, if used, must immediately follow *data-name-1*.

The following rules apply to the use of REDEFINES:

1. The level-numbers of *data-name-1* and *data-name-2* must be equal.
2. Redefinition starts at *data-name-2* and ends when a level-number numerically less than or equal to that of *data-name-2* is encountered.
3. The entries redefining an area must immediately follow the entries originally describing the same area.
4. The size associated with the redefinition, i.e., with *data-name-1*, must not exceed the size of the original area, i.e., *data-name-2*.
5. This clause is not used at the record (01) level in the File Section; a DATA RECORD clause in the FD entry that names more than one data record implies automatic redefinition.

When an area is redefined, all descriptions of the area remain in effect. Thus, if B and C are two separate items which share the same storage area, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same physical area would receive Y according to the description of C. A redefinition does not cause any data to be erased and does not supersede a previous description.

The original example shown in this discussion might be redefined as follows within the Working-Storage Section:

```
01  REFUND-WORK-AREA . . . .
                    .
                    .
                    .

01  BILLING-WORK-AREA REDEFINES REFUND-WORK-AREA . . . .
                    .
                    .
                    .
```

An additional application of the REDEFINES clause will be discussed later in this chapter under the heading Constructing Tables of Constants.

COPY

The purpose of the COPY clause is to enable the programmer to include prewritten Record Description entries in the Data Division. These prewritten entries may be copied from elsewhere in the Data Division or from the COBOL library. (See Chapter 4 for a discussion of the library.) Briefly, the effect of the COPY clause is to extract an entry or a series of entries from another place and put it into a program at the point where the COPY clause appears. The complete general form of a Record Description entry containing a COPY clause is the following:

---

level-number data-name-1 [REDEFINES *data-name-2*]

COPY *data-name-3* [FROM LIBRARY] .

As shown in the above diagram, when COPY is used, the only other clause that may appear in the entry is REDEFINES.

The information being copied is inserted at the point in an entry where the COPY clause appears. Thus, *level-number-1* and *data-name-1* are not replaced by the information being copied, nor is the REDEFINES clause if it is present.

Copying ends when, in the entries being copied, a level-number is encountered which is numerically equal to, or less than, the level-number *originally* associated with *data-name-3*. In this way, one entry or many entries may be duplicated with one COPY.

The level-number of *data-name-3* does not have to equal the level-number of *data-name-1*. If the level-numbers differ, all level-numbers of succeeding entries being copied will be adjusted appropriately by the difference between the level-numbers of *data-name-1* and *data-name-3*.

When *data-name-3* is an entry appearing elsewhere in the Data Division (rather than in the library), it may occur either before or after the entry referring to it. *Data-name-3* must never be subscripted.

When information is to be copied from the library, the FROM LIBRARY option must be included. Then if *data-name-3* is not at the 1 (01) level in the library, it must be qualified by the item containing it which *does* have a level-number of 1. This qualification is necessary, even though *data-name-3* may be unique. An entry being copied may itself contain a COPY clause only if it refers to the library.

Working-Storage Section

The Working-Storage Section is used to describe areas of storage where intermediate results and other items are stored temporarily at object time. This section consists of a series of Record Description entries, each of which describes an item in a work area. The entries must be preceded by the header WORKING-STORAGE SECTION. In general, work area items are of two types, *independent* and *grouped.*

Independent Work Areas

An independent work area consists of a single item which is not subdivided and is not itself a subdivision of some other item. It is always assigned the level-number 77. Each independent item must be described in a separate Record Description entry consisting of the following parts:

The level-number 77.

A data-name.

A CLASS or PICTURE clause.

A SIZE clause (if a PICTURE clause is not used).

The OCCURS clause must not be used in describing an independent item. The use of any other clause is optional. When writing the Working-Storage Section, entries for all independent items are placed before the entries describing grouped items.

Independent work areas are frequently used for the temporary storage of intermediate results pending completion of a calculation. For example, suppose the programmer wishes to total several items in order to obtain an average, but that he wishes to retain the total for some further calculation. In this case, the total would have to be stored temporarily. Unless it were to be used as part of a larger grouping of items, it would often be convenient to store it in some independent work area.

Group Work Areas

A group work area consists of two or more items grouped to form a record. The entries used to write the description of a group work area in the Working-Storage Section are exactly the same in format as those used to describe a record in the File Section. Any clause which may be used in a Record Description entry may also be used to describe an item in this part of the Working-Storage Section.

Initial Values

The initial value of any item in the Working-Storage Section may be specified by using a VALUE clause in the Record Description entry describing the item. An initial value is a value to be assumed by the item at the time execution of the object program is begun.

Any initial value, numeric or non-numeric, may be assigned to an item in the Working-Storage Section with the following restrictions:

1. The value must be compatible with the CLASS of the item. For example, if the CLASS is NUMERIC, only a numeric value may be assigned as an initial value.
2. The size of the initial value must not exceed the SIZE of the item. If the size of the value is less, standard rules for justification apply.
3. If the initial value of a work area is not specified by a VALUE clause, its initial value will be unpredictable at object time.

Any item within a work area may be assigned condition-names.

The example below shows how entries for a Working-Storage Section might be written. The header WORKING-STORAGE SECTION is followed immediately by the entries describing independent items, and these are followed in turn by entries describing grouped items.

```
WORKING-STORAGE SECTION.
77  TAX-DISCOUNT PICTURE 99999V99 USAGE IS COMPUTATIONAL
    VALUE IS ZERO.
77  MARITAL-STATUS CLASS IS NUMERIC SIZE IS 1 USAGE IS
    COMPUTATIONAL.
88  MARRIED VALUE IS 1.
88  SINGLE VALUE IS 2.
                                .
                                .
                                .
```

```
01  WORK-MASTER.
    02  NAME CLASS IS ALPHANUMERIC USAGE IS DISPLAY . . . .
    03  LAST-NAME . . . .
                                .
                                .
                                .
01  PAY-CHECK-RECORD . . . .
                                .
                                .
                                .
                                .
```

Constant Section

The concept of the named constant was introduced in Chapter 3. A named constant is a named item having a value that does not change during the course of a program. The Constant Section of the Data Division contains the entries which describe named constants and specify their values.

The Constant Section is organized in exactly the same manner as the Working-Storage Section. It is begun with the header CONSTANT SECTION and consists solely of Record Description entries. As in the Working-Storage Section, entries are of two types, *independent* and *grouped.*

Independent Constant Entries

An independent constant entry describes a single item which is not subdivided and is not itself a subdivision of some other item. It is always assigned the level-number 77. Each independent item must be described in a separate Record Description entry consisting of the following parts:

The level-number 77.

A data-name.

A CLASS or PICTURE clause.

A SIZE clause (if a PICTURE clause is not used).

A VALUE clause.

Neither the OCCURS clause nor the REDEFINES clause is meaningful in an independent constant entry. When the Constant Section is written, independent entries precede the group entries.

Grouped Constants

Grouped constant entries consist of two or more constants grouped to form a record. Such entries are often used to describe a series of constants to be stored for use in a table. The entries used to write the description of a group of constants in the Constant Section are exactly the same in format as those used to describe a record in the File Section. Any clause which may be used in a Record Description entry may also be used to describe an item in the Constant Section.

Values

Any value, numeric or non-numeric, may be specified for a constant, with the following restrictions:

1. The size of the literal in the VALUE clause must not exceed the SIZE of the item. If the size of the literal is less, standard rules for justification apply.
2. The value specified must not contradict the CLASS of the item.

Constructing Tables of Constants

Reference data is often organized in the form of a *table*. The COBOL system provides two methods of constructing such tables. One method involves naming and describing each item of data individually; this is often done in the Constant Section, but input data may be organized into tables by means of appropriate entries in the File Section. The second method involves the use of subscripting, as explained in Chapter 4.

If the programmer wishes to employ subscripting to obtain items from a table, he must first describe the table in the Constant Section as a group of constants. He must then *redefine* the table by means of a REDEFINES clause, accompanied by an OCCURS clause which furnishes the processor with the information necessary to permit subscripting.

For example, suppose that a table consists of the 1960 populations of the 50 states, organized according to the alphabetical order of the state names. This table could be written in the Constant Section as a record containing a group of constants. The following entries might be used:

```
01  POPULATION-RECORD
    02  ALABAMA PICTURE 99999999 USAGE COMPUTATIONAL
        VALUE 3266740.
    02  ALASKA PICTURE 99999999 USAGE COMPUTATIONAL
        VALUE 226167.
    02  ARIZONA PICTURE 99999999 USAGE COMPUTATIONAL
        VALUE 1302161.
                .
                .
                .
```

At object time, this table of 50 population figures will be available for use in processing data. For example, the population for Arizona could be moved to a work area called TOTAL-POPULATION by writing MOVE ARIZONA TO TOTAL-POPULATION.

Suppose, however, that the programmer wishes to use subscripting in referring to the table. In effect, subscripting allows the programmer to select an item by its relative *position* in the table, as opposed to the technique of referring to it by name. Thus, a subscript which selected the third POPULATION-RECORD would in this case select the figure for Arizona.

Before the programmer can make use of subscripts, he must *redefine* the group of constants representing the population figures. This might be done by the following entries, which would create a table called POPULATION-TABLE, consisting of items called STATE-POPULATION.

```
01  POPULATION-TABLE REDEFINES POPULATION-RECORD.
04  STATE-POPULATION PICTURE 99999999 USAGE IS COMPU-
    TATIONAL OCCURS 50 TIMES.
```

The effect of the OCCURS clause is to inform the processor that the series of constants consists of 50 successive items, each with a PICTURE of 99999999 and COMPUTATIONAL USAGE. Once this has been done, any of the 50 items can be referred to by the use of a subscript.

In the example given, it is assumed that the population figure for Arizona will be the third figure in the table. The subscript 3 is therefore required to obtain it. The

programmer might specify the subscript directly, using such a statement as MOVE STATE-POPULATION (3) TO TOTAL-POPULATION. On the other hand, he could arrange to have the value 3 assumed by a data-item called STATE; then he might write MOVE STATE-POPULATION (STATE) TO TOTAL-POPULATION. In the latter case, the object program would employ the current value of the item called STATE, namely 3, as the subscript. This technique permits the programmer to vary the subscript as a result of changes in input data or by calculation; it allows him to select any value in the table by means of a single procedure statement, whereas if he specified a particular subscript value, he would be limited to the one item indicated. The technique of subscripting, in other words, permits the programmer to cause the selection from the table to be made at object time in accordance with the nature of the data being processed.

The general procedure of specifying a table as a group of constants and then redefining it so that it can be referred to by subscripting can be applied to one, two, or three levels of subscripting.

Chapter 7: Procedure Division

Introduction

Just as verbs in the English language designate action, so it is with the COBOL verbs. Whereas the entries in the other divisions of a COBOL source program describe or define *things,* the verbs specify action, or *procedures,* to be carried out. Accordingly, the COBOL verbs form the basis of the Procedure Division of a source program. The verbs fall into two main categories. Most of them are used in statements that specify the data processing steps the object program is to perform and thus they are called *program verbs.* The other category comprises the verbs that direct the processor; they are known as processor-directing verbs, or simply as *processor verbs.*

Verbs

The COBOL verbs are listed below. The organization of this chapter is based on the classifications used in this list:

Program Verbs

Input/Output	OPEN READ WRITE CLOSE ACCEPT DISPLAY
Data Manipulation	MOVE EXAMINE
Arithmetic	ADD SUBTRACT MULTIPLY DIVIDE COMPUTE
Sequence Control	GO TO ALTER PERFORM STOP

Processor Verbs

ENTER
EXIT
NOTE

Format

Each verb in COBOL has one or more fixed formats, or contexts, in which it can be employed. The format indicates the arrangement of a verb and its operands and thus defines a particular type of procedure statement. The verb formats as presented in this chapter are set off by horizontal lines to distinguish them from text and examples.

As noted in Chapter 3, commas can be used as series separators when a verb has two or more operands. Because this usage is optional, such commas do not appear

in the verb formats. Commas do appear in some of the sample statements, however, to illustrate how the programmer can insert them for the sake of readability.

The way in which statements are combined to form sentences and the ways in which the larger units of procedure (i.e., paragraphs and sections) are formed are discussed in Chapter 3. The rules for entering this information on COBOL Program Sheets are given in Chapter 5.

Program Verbs

Each of the COBOL program verbs causes some event or series of events to take place at object time, that is, at the time at which the object program is run. In order to simplify the discussion of the program verbs and the statements in which they appear, expressions such as, "when the COMPUTE verb is executed," are used occasionally in this chapter. The reader should realize that such usage actually refers to the object-time execution of the corresponding machine instructions produced by the processor.

The Input/Output Verbs

In a data processing system, the flow of data through the computer is governed by an input/output control system. Associated with this system in COBOL are the input/output verbs. Four of the verbs—OPEN, READ, WRITE and CLOSE—are used to specify the flow of data to and from files stored in external media. The remaining verbs, ACCEPT and DISPLAY, are used to govern low-volume information that is to be obtained from or sent to I/O devices such as a card reader or console typewriter.

The input/output control system is a record processing system. That is, the unit of data made available by a READ or passed along by a WRITE is the *record*, as described in the chapters on data description. The programmer is concerned only with the use of individual records; the input/output control system automatically provides for such operations as the movement of data into buffers and/or internal storage, validity checking, error correction (where feasible), unblocking and blocking, and tape alternation.

OPEN

The OPEN verb is used to initiate the processing of one or more input and/or output files. Its format is:

OPEN [INPUT *file-name-1* [*file-name-2* . . .]] [OUTPUT *file-name-3* [*file-name-4* . . .]]

At least one of the two optional clauses (INPUT or OUTPUT) must be written. An OPEN statement can name just one file or it can name all of the files to be processed by the program. In other words, the programmer can open all the files at one time, if desired, or he can open one or more at a time according to the requirements of the program. In any case, an OPEN statement must be executed for a given file before a READ or a WRITE pertaining to that file can be executed.

Some examples of the use of the OPEN verb are:

```
OPEN INPUT BACK-ORDERS
OPEN INPUT MASTER-IN TRANSACTIONS OUTPUT MASTER-OUT
    INVOICES EXCEPTIONS
OPEN OUTPUT STATISTICS
```

These additional points should be noted in connection with OPEN:

1. Each file named in an OPEN statement must be defined by an FD (File Description) entry in the Data Division of the program.
2. If the FD entry for a given file indicates that label records are used, the execution of an OPEN statement causes the checking of the label record (if INPUT) or the writing of a label record (if OUTPUT).
3. When the file being opened is an input file, the OPEN does *not* cause the first data record to be made available for processing. This occurs only as a result of the first READ.
4. A second OPEN of a particular file can be executed only if preceded by execution of a CLOSE of that file.
5. If a file being opened is an input file designated as OPTIONAL in the FILE-CONTROL paragraph of the Environment Division, the object program will contain an interrogation procedure to determine whether the file is present. If the file is not present, the OPEN will not be executed; furthermore, a message will be displayed indicating the absence of the file and the object program will proceed as though the file were in end-of-file status. Thus, when the first READ for the file is encountered, the end-of-file action specified in the READ statement (see the AT END option of the READ verb) will occur.

READ

The function of the READ verb is to get the next record from an input file and make it available for processing. Provision is made for the execution of an imperative statement when the end of file is reached. The format of a READ statement is:

READ *file-name* RECORD [INTO *area-name*]

[AT END *any imperative statement*]

An OPEN statement for the *file-name* file must be executed prior to the execution of the first READ for that file.

When a READ is executed, the next record in the named file becomes accessible in the input area defined by the associated Record Description in the File Section of the Data Division. The record remains available in the input area until the next READ (for that file) is executed. The named file must be defined by an FD entry in the Data Division of the program.

If a file contains more than one type of record, the READ verb delivers the *next* record regardless of type. The differing records automatically share the same input area; thus the programmer must provide for determining the type of the current record and must refer only to information that is present in the current record.

Each time an end-of-reel condition occurs in a reel other than the last, the READ verb causes the following operations to take place:

1. If labels are present (as specified in the FD for that file) the standard end-of-reel label subroutine of the input/output control system is executed.
2. A tape alternation occurs, if appropriate.
3. If labels are present, the standard beginning-of-reel label subroutine is executed.
4. The next record in the file is made available for processing.

If the *file-name* file is OPTIONAL and is not present at object time, the *any imperative statement* will be executed when the first READ for the file is encountered. (See the comments on OPTIONAL files in the discussion of the FILE-CONTROL paragraph in Chapter 8 and the OPEN verb in this chapter.)

The INTO Option

The INTO *area-name* option converts the READ into a READ *and* MOVE. The *area-name* specified must be the name of either a working area or an output record area. If the format of the INTO area differs from that of the input record, the data will be moved in accordance with the rules for the MOVE verb without the CORRESPONDING option.

When the INTO *area-name* option is used, the current record becomes available in the input record area, as well as in the INTO area.

The AT END Option

The AT END option permits the programmer to specify one imperative statement to be executed when the end of file is detected. The AT END *any imperative statement* is actually a special type of conditional statement and is discussed as such in Chapter 3.

Every READ statement must contain either an explicit or an implicit AT END statement. If none is stated, the processor will examine all other READ statements associated with the particular file. If just one of the READ statements contains an AT END statement, the processor will, in effect, append that statement to each of the other READ statements. However, if more than one, but not all, READS for a given file have AT END statements, this will constitute an error and the processor will so indicate.

Once an AT END statement has been executed, an attempt to READ from the associated file will constitute an error unless a subsequent CLOSE and OPEN have been executed for that file.

EXAMPLES

Some typical READ statements are:

```
READ TRANSACTIONS RECORD
READ MASTER-IN RECORD INTO WORK-AREA AT END GO TO
    END-OF-JOB
READ STATISTICS RECORD AT END GO TO SUMMARY
```

WRITE

The WRITE verb is used to release a record for insertion in an output file. The format for a WRITE is:

WRITE *record-name* [FROM *area-name*]

The file associated with *record-name* must be defined by an FD entry in the Data Division of the program. At object time, an OPEN statement for that file must be executed before the first WRITE for the file is executed.

When a WRITE statement is executed, the *record-name* record is released. Accordingly, all the desired processing steps must be performed before the WRITE occurs.

At the end of each reel other than the last in the output file, the WRITE verb causes the following operations to take place:

1. If labels are specified (in the FD for that file), the standard end-of-reel label subroutine of the input/output control system is executed.
2. A tape alternation occurs, if appropriate.
3. If labels are specified, the standard beginning-of-reel subroutine is executed.

The FROM Option

The FROM *area-name* option of the WRITE verb is comparable to the INTO *area-name* option of the READ verb. It effectively converts the WRITE into a MOVE *and* WRITE. The *area-name* must be the name of an input record area, a working area, or a constant area. If the format of the FROM area differs from that of *record-name,* the data will be moved in accordance with the rules for the MOVE verb without the CORRESPONDING option.

When the FROM *area-name* option is used, the information in the *area-name* area continues to be available.

Note: The names used for *record-name* and for *area-name* cannot be the same.

EXAMPLES

Some sample statements illustrating the use of the WRITE verb are:

```
WRITE INVOICE
WRITE MASTER-OUT FROM WORK-AREA
WRITE VOLUME FROM TABLE (R-VALUE)
```

CLOSE

The purpose of the CLOSE verb is to terminate the processing of one or more input and/or output reels or files. Provision is included for optional rewinding and/or locking. The CLOSE format is:

CLOSE *file-name-1* [REEL] [WITH {LOCK / NO REWIND}] [*file-name-2* . . .]

Each file named in a CLOSE statement must be defined in an FD entry in the Data Division of the source program.

CLOSE file-name

An OPEN statement must be executed for a given file before it can be closed. When a CLOSE *file-name* statement is executed at object time, the final closing conventions are performed for each file specified and the data areas are released.

Detailed information concerning the functioning of the CLOSE verb with respect to the IOCS (input/output control system) will be presented in the publications dealing with the respective processors. In general, however, the following events occur when a CLOSE is executed with respect to an entire file (i.e., as opposed to a reel of the file):

1. If the file is an input file and it is in end-of-file status, ending-label checking will be performed (assuming labels are present) by the end-of-file label subroutine of the IOCS, the end-of-file routine of the IOCS will be executed, and the data

areas will be released. If the file is not in end-of-file status, no label checking will occur, but the other steps will be performed.

2. If the file is an output file and labels are specified, label writing will be accomplished by the IOCS end-of-file label subroutine.
3. Furthermore, for either an input file or an output file:
   a. If neither the LOCK nor the NO REWIND option has been specified, the current (final) reel of the file will be rewound.
   b. If the NO REWIND option is used, the current (final) reel of the file will remain in its current position.
   c. If the LOCK option is used, the current (final) reel will be rewound using an appropriate technique to insure that it cannot be read or written upon.

If a file mentioned in a CLOSE statement is an OPTIONAL input file (see the FILE-CONTROL paragraph of the Environment Division), the closing conventions will not be performed when the file is not present at object time.

CLOSE file-name *REEL*

The CLOSE *file-name* REEL option can be employed with either an input or an output file. This option is used only when it is desired to CLOSE a reel of a file prior to its normal end. Details will be specified in the respective processor publications; but, in general, the following action will take place with respect to the current reel of the file:

1. If the reel is part of an input file: There will be no checking of the ending label. If the reel happens to be the last one in the file, an error may result at object time, since no end of file will occur.
2. If the reel is part of an output file, the standard end-of-reel processing takes place immediately.
3. Furthermore, for either input or output:
   a. If neither LOCK nor NO REWIND is specified, the reel will be rewound.
   b. If the NO REWIND option is used, the current reel is not rewound.
   c. If the LOCK option is used, the current reel will be rewound utilizing a technique to insure that it cannot be read or written upon.

EXAMPLES

Some examples of CLOSE statements are:

```
CLOSE TRANSACTIONS
CLOSE MASTER-IN WITH LOCK MASTER-OUT WITH LOCK
CLOSE SAMPLING-ANALYSIS WITH NO REWIND
CLOSE INVOICES REEL, BACK-ORDERS REEL
```

ACCEPT

The function of the ACCEPT verb is to obtain low-volume data from an input device. The format of ACCEPT is:

ACCEPT *data-name* [FROM *mnemonic-name*]

For each machine system, a *standard* ACCEPT device will be specified in the publication covering the details of the Environment Division for that system. In most cases, this device will be the card reader. Unless the FROM option is used, the *data-name* data is read from the standard device. Data can be obtained from an alternate device by utilizing the FROM *mnemonic-name* option. In this case, *mnemonic-name* corresponds to an input device defined in the SPECIAL-NAMES paragraph of the Environment Division of the program. Thus, the programmer might write ACCEPT statements such as:

```
ACCEPT CANCELLATIONS
ACCEPT DATE FROM CONSOLE
ACCEPT CODE FROM CARD-READER
```

The maximum size of the data represented by *data-name* will be specified in the publications dealing with the respective processors. If the format of *data-name* contains fewer than the maximum number of characters, the data will appear in the leftmost positions of the input area associated with the device. The processor will provide appropriate instructions in the object program to accommodate any difference in size and to move the data to the *data-name* area.

DISPLAY

This verb is used to display low-volume data on an output device. The format of the DISPLAY verb is:

$$\underline{\text{DISPLAY}} \left\{ \begin{matrix} \textit{data-name-1} \\ \textit{literal-1} \end{matrix} \right\} \left[\left\{ \begin{matrix} \textit{data-name-2} \\ \textit{literal-2} \end{matrix} \right\} \ldots \right] \left[\underline{\text{UPON}}\ \textit{mnemonic-name} \right]$$

For each machine system, a *standard* DISPLAY device will be specified in the publication dealing with the details of the Environment Division for that system. In most cases, the standard device will be a typewriter or a printer. Unless the UPON option is used, the specified data-name(s) and/or literal(s) will be "written" on this device. The programmer can cause information to be displayed on an output device other than the standard DISPLAY device by utilizing the UPON *mnemonic-name* option. The *mnemonic-name* must correspond to an output device defined in the SPECIAL-NAMES paragraph of the Environment Division of the program.

A combination of data-names and literals can be used in a DISPLAY statement in order to convey the desired information. A literal employed in the statement will itself appear in the resulting message, whereas it is the *value* of a data-name that will appear in the information displayed. To illustrate this point, suppose that the programmer has written the following DISPLAY statement:

```
DISPLAY 'VALUE OF CHECK-SUM IS ' CHECK-SUM.
```

Assume further that the value of the data item named CHECK-SUM is 0342112 at the time this DISPLAY statement is executed. Then, the information that will appear on the DISPLAY device at object time is:

```
VALUE OF CHECK-SUM IS 0342112
```

Some other examples of DISPLAY statements are:

```
DISPLAY 'END OF PHASE 1'
DISPLAY GRAND-TOTAL UPON PRINTER
DISPLAY HIGH-VALUES
DISPLAY QUOTE 'QUOTE' QUOTE
```

As indicated in the last two of these sample statements, figurative constants can be used in DISPLAY statements. It should be noted that the third statement above will cause a single HIGH-VALUE character to be displayed at object time, since any other number of such characters is indeterminable. The last example will cause the following information to appear at object time:

'QUOTE'

The DISPLAY device produces output only in multiples of some minimum unit; the data items will appear one after the other so as to fill the first unit, then the second, and so on. For example, if a card punch is used for DISPLAY purposes and three 50-character data items are to be displayed, the first item and the first 30 characters of the second item will be punched in the 80 columns of the first card, and the balance of the second item and all of the third item will be punched in the first 70 columns of a second card.

Data Manipulation Verbs

The movement of data from one place to another within the computer and the inspection of data are implicit in the functioning of several of the COBOL verbs. For example, execution of the COMPUTE verb can involve editing of, as well as movement of, the result. This handling of data is incidental to the main purpose, however, except in the case of the two data manipulation verbs, MOVE and EXAMINE. The MOVE verb has as its primary function the transmission of data from one area to another. EXAMINE involves the inspection of data within the computer, with or without movement. These two verbs are discussed in detail in the following paragraphs.

MOVE

The MOVE verb is used to transfer information from one data area to one or more other areas within the computer. Concurrent editing takes place automatically in certain cases according to the format of the data items as described in the Data Division. The MOVE verb can be used in either of two formats:

Option 1

$$\underline{\text{MOVE}} \left\{ \begin{matrix} \textit{data-name-1} \\ \textit{literal} \end{matrix} \right\} \underline{\text{TO}} \ \textit{data-name-2} \left[\textit{data-name-3} \ \ldots \right]$$

Option 2

$$\underline{\text{MOVE CORRESPONDING}} \ \textit{data-name-1} \ \underline{\text{TO}} \ \textit{data-name-2} \left[\textit{data-name-3} \ \ldots \right]$$

The Simple MOVE

When the simple MOVE (Option 1) is executed at object time, the data represented by *data-name-1* or the specified literal is moved to the area designated by *data-name-2*. The same information is moved also to any additional area(s) mentioned in the statement, i.e., to the *data-name-3* area, etc. This movement does not destroy the original data—it makes "copies" of it in the designated areas.

Since both a "source" (specified by *data-name-1* or *literal*) and a "receiving" area (designated by *data-name-2, data-name-3,* etc.) can be either an elementary data item or a group item, a MOVE can involve one of four possible situations:

elementary item	→	elementary item
elementary item	→	group item
group item	→	elementary item
group item	→	group item

All four of these cases are permitted. However, when a group item is involved, as in the latter three cases, the data is moved without any regard to the level structure of the group items involved and without any editing. Thus, when a group item is present, the data being moved is treated simply as a sequence of alphanumeric characters and is placed in the receiving area in accordance with the rules for moving elementary non-numeric items (see below). If the two items differ in size, the processor will produce a warning message when the statement is encountered. Normally, when a group item is involved, the MOVE is a group-to-group transfer of data and the descriptions of the two items are identical.

When both the source and the receiving areas are elementary items, editing appropriate to the format of the receiving area occurs automatically in the execution of the MOVE. The editing that is performed depends on whether the source data (specified by *data-name-1* or *literal*) is numeric or non-numeric, as follows:

NUMERIC DATA ITEMS

1. The data from the source area is aligned with respect to the decimal point (assumed or actual) in the receiving area. This alignment may result in the loss of leading digits or of low-order digits (or both if the source area is larger than the receiving area); a situation that would result in the loss of leading digits will cause the processor to produce a warning message at process time. Any excess positions in the receiving area will be filled with zeros.
2. If necessary, the data from the source area is converted to the mode of internal representation specified for the receiving area. For example, a difference in USAGE might require conversion from COMPUTATIONAL mode to DISPLAY mode.
3. If required by the format of the receiving area, zeros are replaced by spaces (blanks); and dollar signs, decimal points, and commas are inserted.
4. If no decimal point has been specified, the data will be right justified unless the data description of the item specifies JUSTIFIED LEFT.

NON-NUMERIC DATA ITEMS

1. The data from the source area is placed in the receiving area beginning at the left (or at the right if the format description of the receiving area specifies JUSTIFIED RIGHT). Note that when a group item is moved, left justification is standard.
2. If the receiving area is not completely filled by the data being moved, the remaining positions are filled with spaces.
3. If the receiving area cannot contain all of the data being transferred, the MOVE terminates when the receiving area is filled. A MOVE statement that would produce this situation will cause the processor to produce a warning message at process time.

Some examples of MOVE statements are:

```
MOVE MASTER-RECORD TO WORK-AREA
MOVE RESULT TO A-RECORD, B-RECORD, C-RECORD
MOVE HIGH-VALUES TO CONTROL-ITEM
MOVE ALL '9' TO SERIAL-NUMBER
```

Figure 7-1 contains several examples illustrating the editing feature of the MOVE verb.

Source Area			Receiving Area		
PICTURE	Data before MOVE	Data after MOVE	PICTURE	Data before MOVE	Data after MOVE
99V99	1234	1234	99V99	9876	1234
99V99	1234	1234	99V9	987	123
9V9	12	12	99V999	98765	01200
XXX	A2B	A2B	XXXXX	Y9X8W	A2B
9V99	123	123	99.99	87.65	01.23
AAAAAA	REPORT	REPORT	AAA	JKL	REP
99V99	1234	1234	$ZZZ9.99	$8765.43	$ 12.34

Figure 7-1. Examples of data before and after MOVE is executed. Standard justification is assumed (see the JUSTIFIED clause in Chapter 6).

Note that in each case in Figure 7-1 the data in the source area remains unaltered after the MOVE has been executed. Note also, as in the fourth example, that the information in any excess positions of a non-numeric receiving area is replaced by spaces at the right. The sixth example shows a situation that would cause the processor to produce a warning message at process time.

The CORRESPONDING Option

The CORRESPONDING option (Option 2) of the MOVE verb permits the programmer to specify the transfer of a group item containing one or more elementary items that require editing in conjunction with the MOVE. When a MOVE CORRESPONDING statement is executed at object time, selected items within the source area (*data-name-1* area) are moved, with any required editing, to selected areas within the receiving area (*data-name-2, data-name-3,* etc.). Items are selected by matching the data-names of items within *data-name-1* with like data-names of areas within *data-name-2,* according to these rules:

1. At least one of the items of a selected pair must be an elementary item.
2. The two data-names must be identical, including all qualification up to but not including *data-name-1* and *data-name-2.*

Each corresponding item in the source area is moved to its corresponding receiving area. Editing appropriate to the format of the receiving area takes place automatically. The rules stated for the simple MOVE apply to each pair of corresponding items in the MOVE CORRESPONDING; thus, the effect of a MOVE CORRESPONDING statement is equivalent to a series of simple MOVE statements.

The following additional rules apply only to the CORRESPONDING option:

1. No area described by an OCCURS clause (in the Data Division) can be involved in the MOVE.
2. Data items with level numbers 77 and 88 (i.e., independent data items and condition-names) cannot be referenced.

To illustrate the use of MOVE CORRESPONDING, suppose that the programmer wishes to transfer corresponding items from a work area named INVENTORY-POSTING to an output area designated INVENTORY-RECORD. He could write this statement:

```
MOVE CORRESPONDING INVENTORY-POSTING TO
    INVENTORY-RECORD
```

Figure 7-2 shows the movement of data that might result from this statement. Note that non-corresponding items in the source area are *not* moved and that non-corresponding items in the receiving area are not affected.

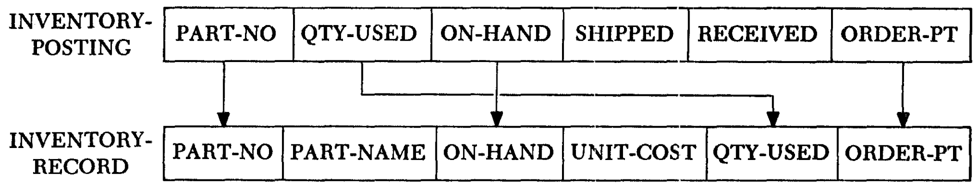


Figure 7-2. Movement of data resulting from execution of MOVE CORRESPONDING.

EXAMINE

The EXAMINE verb is used to replace a given character and/or to count the number of times it appears in a data item. Its format is:

```
                       { TALLYING { ALL          }                                       }
                       {          { LEADING      }                                       }
                       {          { UNTIL FIRST  }                                       }
EXAMINE data-name      {                    literal-1 [REPLACING BY literal-2]           }
                       {                                                                 }
                       { REPLACING { ALL            }                                    }
                       {           { LEADING        }  literal-3 BY literal-4            }
                       {           { [UNTIL] FIRST  }                                    }
```

The EXAMINE verb can be applied only to a data item whose USAGE is defined as DISPLAY. If USAGE is COMPUTATIONAL, the processor will indicate an error condition.

Any literal used in an EXAMINE statement must be a member of the character set associated with the CLASS specified for *data-name*. In other words, if the description of *data-name* in the Data Division specifies a CLASS that uses less than the full character set (e.g., NUMERIC or ALPHABETIC), then each literal used in an EXAMINE statement must be one of the characters in the restricted set. Thus, if the CLASS of *data-name* is NUMERIC, each literal used in the statement must be a numeric character.

It is important to note that all literals in EXAMINE statements are employed in the sense of alphanumeric literals. If a literal is a numeric character, say 0, the data item is examined for the presence of the *character* 0, not for the *value* 0. Accordingly, all literals in EXAMINE statements are considered alphanumeric and therefore are enclosed in quotation marks.

When an EXAMINE statement is executed, the examination begins with the first (i.e., the leftmost) character of the data item and proceeds to the right. Each character in the item represented by *data-name* is examined in turn. If the data item being examined is numeric, any operational sign associated with the item will be ignored.

The effects of an EXAMINE statement depend on the options employed by the programmer, as follows:

If TALLYING is specified:

When the TALLYING option is used, a count of the number of certain characters in *data-name* is made and this count replaces the value of a special register called TALLY, which is accessible to the programmer. The count depends on which of the three options of TALLYING is employed:

1. If ALL is specified, all occurrences of *literal-1* in the data item are counted.
2. If LEADING is specified, the count represents the number of occurrences of *literal-1* prior to encountering a character other than *literal-1*.
3. If UNTIL FIRST is specified, the count represents the number of characters other than *literal-1* encountered prior to the first occurrence of *literal-1*.

If REPLACING is specified:

When the REPLACING option is used (either with or without the TALLYING option), the replacement of characters depends on which of the four options of REPLACING is employed:

1. If ALL is specified, a *literal-2* (or a *literal-4*) is substituted for each occurrence of *literal-1* (or *literal-3*).
2. If LEADING is specified, the substitution of *literal-2* for *literal-1* (or *literal-4* for *literal-3*) terminates when a character other than *literal-1* (or *literal-3*) is encountered or when the right-hand boundary of the data item is reached.
3. If UNTIL FIRST is specified, the substitution of *literal-2* (or *literal-4*) terminates as soon as the first *literal-1* (or *literal-3*) is encountered or the right-hand boundary of the item is reached.
4. If FIRST is specified, only the first occurrence of *literal-3* is replaced by *literal-4*.

Several examples of EXAMINE statements are included in Figure 7-3. These examples illustrate the various ways in which EXAMINE can be used and also show the effect of each statement on a hypothetical data item and on the TALLY register. (Figure 7-3 appears on the following page.)

The Arithmetic Verbs

COBOL provides a verb corresponding to each of the four basic arithmetic operations: ADD, SUBTRACT, MULTIPLY and DIVIDE. A fifth arithmetic verb, COMPUTE, is provided to permit the programmer to include *arithmetic expressions* in his source program. These verbs are discussed in turn in the following paragraphs, although the itemized rules stated under the ADD verb apply to all of the arithmetic verbs.

ADD

ADD is used to add two or more numeric values and to substitute the resulting sum for the current value of an item. The format of an ADD statement is:

$$\underline{\text{ADD}} \left\{ \begin{matrix} \textit{data-name-1} \\ \textit{literal-1} \end{matrix} \right\} \left[\left\{ \begin{matrix} \textit{data-name-2} \\ \textit{literal-2} \end{matrix} \right\} \cdots \right]$$

$$\left[\left\{ \begin{matrix} \underline{\text{TO}} \\ \underline{\text{GIVING}} \end{matrix} \right\} \textit{data-name-n} \right] \left[\underline{\text{ROUNDED}} \right]$$

$$\left[\text{ON } \underline{\text{SIZE}} \ \underline{\text{ERROR}} \ \textit{any imperative statement} \right]$$

An ADD statement must name at least two addends. Thus the minimum ADD statement is of the form, ADD *data-name-1 data-name-2* or of the form, ADD *data-name-1* TO *data-name-n*.

EXAMINE Statement	ITEM-1 Data		Resulting Value of TALLY
	Before	After	
EXAMINE ITEM-1 TALLYING ALL '0'	101010	101010	3
EXAMINE ITEM-1 TALLYING ALL '1' REPLACING BY '0'	101010	000000	3
EXAMINE ITEM-1 TALLYING LEADING '0'	004070	004070	2
EXAMINE ITEM-1 TALLYING LEADING 'A' REPLACING BY SPACE	AA4070	4070	2
EXAMINE ITEM-1 TALLYING UNTIL FIRST 'X'	ZZZ1X2	ZZZ1X2	4
EXAMINE ITEM-1 TALLYING UNTIL FIRST 'Z' REPLACING BY 'Y'	ZZZ1X2	ZZZ1X2	0
EXAMINE ITEM-1 REPLACING ALL '0' BY '1'	101010	111111	(unchanged)
EXAMINE ITEM-1 REPLACING LEADING '*' BY SPACE	**7000	7000	(unchanged)
EXAMINE ITEM-1 REPLACING FIRST '*' BY '$'	**1.94	$*1.94	(unchanged)
EXAMINE ITEM-1 REPLACING UNTIL FIRST 'C' BY 'D'	ABCABC	DDCABC	(unchanged)

Figure 7-3. Sample EXAMINE statements showing the effect of each statement on the associated data item and on the TALLY register.

When the GIVING option is used, the value of *data-name-n* is made equal to the sum of the values of the preceding data-names and/or literals. If *data-name-n* is not used as an addend, its format can contain editing symbols (see Rule 5 below).

When the TO option is used, the values of all the data-names, including *data-name-n*, and literals in the statement are added and the resulting sum becomes the value of *data-name-n*.

If neither the GIVING nor the TO option is used, the rightmost addend must be a data-name. The values of the literals and/or data-names are summed and the result replaces the current value of the rightmost data-name.

Some examples of the use of the ADD verb are:

```
ADD BASEPAY, O-T-PAY, BONUS GIVING GROSSPAY
ADD 1 TO COUNTER
ADD 40, OVERTIME TO HOURS-WORKED
ADD QTY-USED, YR-TO-DATE-USAGE
```

In the first of these examples, the values of BASEPAY, O-T-PAY, and BONUS are added and the resulting sum becomes the new value of GROSSPAY. In the second case, the value 1 is added to the current value of COUNTER. The next example results in the sum of 40 and the value of OVERTIME being added to the value of HOURS-WORKED. In the last example, the value of QTY-USED is added to the value of YR-TO-DATE-USAGE.

General Rules for Arithmetic Verbs

The following rules apply to the ADD verb and to the other four arithmetic verbs as well. These rules are not repeated in the discussion of SUBTRACT, MULTIPLY, DIVIDE and COMPUTE:

1. All data-names used in arithmetic statements must represent numeric data items that are defined as elementary items in the Data Division of the program. A data-name that is defined as having a constant value cannot appear in an arithmetic statement as the name of a result.
2. All literals used in arithmetic statements must be numeric.
3. The maximum size of any operand (data-name or literal) is 18 decimal digits. If the format for any operand specifies a size greater than 18 digits, the processor will produce an error message when the discrepancy is encountered. The limit of 18 digits does not apply to intermediate results, which are carried out by the object program to one more place (both on the right and on the left) than is specified for the largest operand involved in the computation.
4. The formats of the two or more operands in an arithmetic statement may differ from each other; e.g., the programmer can ADD *data-name-1* TO *data-name-2* even though the PICTURES of the two operands are 99V9 and 9V99, respectively. Decimal point alignment is supplied automatically throughout computations.
5. The format of any data item involved in computations (e.g., addends, subtrahends, multipliers, etc.) cannot contain editing symbols. If this rule is violated, the processor will indicate the error by an appropriate message. Operational signs and implied decimal points are not considered editing symbols (see the discussion of editing and PICTURE in Chapter 6). The *data-name* in the GIVING option of each of the four simple arithmetic verbs and the *data-name-1* in the COMPUTE verb format represent data items which must not enter into computations if they contain editing symbols.
6. The only figurative constant permitted in arithmetic statements is ZERO (including ZEROS and ZEROES).

The ROUNDED Option

If the number of *decimal* places in a computed result (sum, difference, product or quotient) exceeds the number of decimal places in the format of the data-name associated with the result (i.e., the data-name that is to take on the value of the result), truncation will occur unless the ROUNDED option has been used. Truncation, which is simply the dropping of excess digits, is always determined by the format of the data-name associated with the result. When ROUNDED is specified, however, the least-significant digit specified by the format of the result is increased by 1 whenever the most-significant digit of the excess is greater than or equal to 5.

Thus, with a format of 9999.9, the value 1076.36 becomes 1076.4 when the ROUNDED option is specified, and 1076.3 when ROUNDED is not used and truncation occurs.

The ON SIZE ERROR Option

Whenever the number of *integral* places (those to the left of the decimal point) in a computed result exceeds the number of integral places in the format of the data-name associated with the result, a *size error* condition arises. In this event, one of two situations will obtain, depending on whether the ON SIZE ERROR option has been used:

1. If ON SIZE ERROR is not used and a size error condition arises, the effect is unpredictable. Testing for a size error condition occurs only when the option is specified in the arithmetic statement.
2. If the ON SIZE ERROR option is used and a size error condition occurs, the data-name associated with the result will retain its value and will *not* take on the

value of the computed result. Instead, the *any imperative statement* specified in the ON SIZE ERROR option will be executed.

Some typical applications of ON SIZE ERROR are:

```
ADD PAGE-TOTAL TO INVOICE-TOTAL ON SIZE ERROR
    GO TO INVOICE-ERROR
ADD BASEPAY, OVERTIME GIVING GROSSPAY ON SIZE-
    ERROR GO TO EXCESS-PAY-ROUTINE
```

The ON SIZE ERROR *any imperative statement* is actually a special type of conditional statement which is discussed in Chapter 3.

SUBTRACT

Using the SUBTRACT verb the programmer can specify the subtraction of one or more numeric values from a specified value and the substitution of the resulting difference for the current value of an item. The format is:

SUBTRACT $\left\{\begin{matrix}data\text{-}name\text{-}1\\literal\text{-}1\end{matrix}\right\}$ $\left[\left\{\begin{matrix}data\text{-}name\text{-}2\\literal\text{-}2\end{matrix}\right\}\ \cdots\right]$

FROM $\left\{\begin{matrix}data\text{-}name\text{-}n\\literal\text{-}n\end{matrix}\right\}$ $\left[\text{GIVING } data\text{-}name\text{-}m\right]$

$\left[\text{ROUNDED}\right]$ $\left[\text{ON SIZE ERROR } any\ imperative\ statement\right]$

The effect of the SUBTRACT verb is to sum the subtrahends, i.e., to add the values of all the operands that precede the FROM, and then to subtract that sum from the minuend, i.e., from the value of the FROM operand. If the GIVING option is used, the resulting difference replaces the current value of *data-name-m*. Otherwise it replaces the current value of the minuend (*data-name-n*); accordingly, the minuend must not be a literal when the GIVING option is not used.

Note: Rules 1 through 6, the ROUNDED option and the ON SIZE ERROR option (which appear following the discussion of ADD) apply also to the SUBTRACT verb.

Some examples of SUBTRACT are:

```
SUBTRACT RETURNS, ON-ORDER FROM ORDERS
SUBTRACT YR-TO-DATE-FICA FROM 144.00 GIVING FICA-DUE
```

In the first of these statements, the sum of the values of RETURNS and ON-ORDER is subtracted from the value of ORDERS and the difference becomes the new value of ORDERS. In the second case, the value of YR-TO-DATE-FICA is subtracted from 144.00 and the difference replaces the current value of FICA-DUE.

MULTIPLY

The MULTIPLY verb is used to multiply two numeric values and to substitute the resulting product for the current value of an item. The MULTIPLY format is:

MULTIPLY $\left\{\begin{matrix}data\text{-}name\text{-}1\\literal\text{-}1\end{matrix}\right\}$ BY $\left\{\begin{matrix}data\text{-}name\text{-}2\\literal\text{-}2\end{matrix}\right\}$

$\left[\text{GIVING } data\text{-}name\text{-}3\right]$ $\left[\text{ROUNDED}\right]$

$\left[\text{ON SIZE ERROR } any\ imperative\ statement\right]$

When the GIVING option is used, the product of the multiplication replaces the current value of *data-name-3*. If GIVING is not specified, the result replaces the current value of the multiplier (i.e., the second operand). In this case the multiplier cannot be a literal.

Note: Rules 1 through 6, the ROUNDED option and the ON SIZE ERROR option (which appear following the discussion of ADD) apply also to the MULTIPLY verb.

Typical usage of MULTIPLY is shown in these statements:

```
MULTIPLY INTEREST-RATE BY MIN-BALANCE ROUNDED
MULTIPLY QTY-USED BY UNIT-COST GIVING MONTHLY-COST
```

In the first example, the product of the values of INTEREST-RATE and MIN-BALANCE, after rounding, will replace the current value of MIN-BALANCE. In the second case, the result of the multiplication will become the new value of MONTHLY-COST.

DIVIDE

The DIVIDE verb provides a means of dividing one numeric value into another and using the result to replace the value of an item. The format is:

DIVIDE { *data-name-1* / *literal-1* } INTO { *data-name-2* / *literal-2* }

[GIVING *data-name-3*] [ROUNDED]

[ON SIZE ERROR *any imperative statement*]

When the GIVING option is used, the resulting quotient replaces the current value of *data-name-3*. If GIVING is not specified, the quotient replaces the current value of the dividend (i.e., the second operand). In this case the dividend cannot be a literal.

Division by zero constitutes a special case of the size error condition. The programmer can anticipate this condition by testing the value of *data-name-1* for zero before the division occurs. Otherwise, the rules of the ON SIZE ERROR option apply.

Note: Rules 1 through 6, the ROUNDED option and the ON SIZE ERROR option (which appear following the discussion of ADD) apply also to the DIVIDE verb.

These examples illustrate the use of DIVIDE:

```
DIVIDE PURCHASE-QUANTITY INTO TOTAL-COST
    GIVING UNIT-COST ROUNDED
DIVIDE 18.75 INTO BOND-DEDUCTIONS
    GIVING NO-OF-BONDS
```

Summary of Simple Arithmetic Verbs

In order to summarize briefly the functioning of the four simple arithmetic verbs, each verb is shown in Figure 7-4 in each of its major variations. The tabular information on the right shows how each operand in each statement is affected by the execution of the statement.

Arithmetic Statement	Value of Each Data Item after Execution of Statement			
	A	B	C	D
ADD A, B, C	A	B	A+B+C	—
ADD A, B TO C	A	B	A+B+C	—
ADD A, B, C GIVING D	A	B	C	A+B+C
SUBTRACT A, B FROM C	A	B	C−(A+B)	—
SUBTRACT A, B FROM C GIVING D	A	B	C	C−(A+B)
MULTIPLY A BY B	A	A×B	—	—
MULTIPLY A BY B GIVING C	A	B	A×B	—
DIVIDE A INTO B	A	B/A	—	—
DIVIDE A INTO B GIVING C	A	B	B/A	—

Figure 7-4. Examples of arithmetic statements showing results of execution.

COMPUTE

With the COMPUTE verb, the programmer can use arithmetic expressions to specify one or a series of arithmetic operations. The format of this verb is:

```
COMPUTE data-name-1 [ROUNDED] = arithmetic expression

[ON SIZE ERROR any imperative statement]
```

The *arithmetic expression* can consist of any meaningful combination of data-names, numeric literals, and the figurative constant ZERO, joined by arithmetic operators. The arithmetic expression may consist simply of a single item. Thus the COMPUTE verb permits most arithmetic operations to be specified in a much more concise manner than is possible using the simple arithmetic verbs. The following examples illustrate this point and also show typical usage of COMPUTE:

```
COMPUTE QTY-ON-HAND = STOCK + RECEIPTS + RETURNS
    - ORDERS-FILLED
COMPUTE D = A + B + C ON SIZE ERROR GO TO EXCESS-D
COMPUTE GROSSPAY ROUNDED = BASE-RATE *
    (HRS-WORKED + .5 * (HRS-WORKED - 40.0))
COMPUTE VOLUME = 4 / 3 * PI * R ** 3
```

These examples include at least one usage of each of the five arithmetic operators, which are shown in the following table:

Operator	Arithmetic Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation

The third of the foregoing examples illustrates the way in which parentheses are used in arithmetic expressions to specify the desired sequence of operations. Parentheses can also be used simply to facilitate reading. Chapter 3 contains additional

information concerning the arithmetic operators and the use of parentheses in arithmetic expressions.

Additional points that should be noted in connection with the COMPUTE verb are:

1. Rules 1 through 6, the ROUNDED option and the ON SIZE ERROR option (which follow the discussion of ADD) apply also to COMPUTE.
2. The ON SIZE ERROR option applies only to the final result, not to any of the intermediate results.
3. When a COMPUTE statement is reduced to its simplest form, i.e., COMPUTE *data-name-1* = *data-name-2* or COMPUTE *data-name* = *literal*, the resulting operation is equivalent to a MOVE.

Note: The formal rules for forming arithmetic expressions are included in Appendix A.

The Sequence Control Verbs

Four of the verbs in COBOL are designed to specify the sequence in which the various source program procedures are to be executed. These verbs are referred to as the sequence control verbs; they are GO TO, ALTER, PERFORM and STOP. Unless one of these verbs is encountered, the statements, sentences and paragraphs of the Procedure Division of a source program are executed one after another in the order of their appearance. The verbs GO TO and PERFORM are used to interrupt the normal execution sequence and to transfer control to some other point in the program. The other two verbs are supplementary—ALTER provides a means of modifying GO TO statements and STOP is used to halt execution of the program. Detailed specifications and examples of these four verbs follow:

GO TO

The GO TO verb provides a means of departing from the normal sequence of procedures, i.e., it is used to specify transfer-type operations. There are two formats in which GO TO can be used:

Option 1

```
GO TO [procedure-name]
```

Option 2

```
GO TO procedure-name-1 procedure-name-2 [procedure-name-3 . . .]
    DEPENDING ON data-name
```

The Unconditional GO TO

The first form of the GO TO verb specifies an unconditional transfer of control to the point named, that is, to the *procedure-name* paragraph or section. As indicated in the format, however, *procedure-name* can be omitted. This alternative of leaving *procedure-name* unspecified can be used only if a procedure-name is to be supplied by an ALTER statement prior to the first execution of the GO TO. (See the discussion of ALTER in the next section of this chapter.) If, at object time, such a GO TO sentence is not completed by means of an ALTER before its first execution, an error stop provided by the processor will occur.

Because of its transfer effect, the unconditional GO TO can be used only as the final statement in the sequence in which it appears. In other words, the programmer

must take care not to use a GO TO in such a way as to bypass succeeding statements and prevent their execution.

The following are typical uses of the unconditional GO TO verb:

```
GO TO FICA-ROUTINE.
PERFORM EQUAL-ROUTINE GO TO EXIT-3.
```

Note: A GO TO sentence that is to be altered must be an unconditional GO TO and must appear as a separate, named paragraph consisting solely of the GO TO sentence. (See the discussion of the ALTER verb.)

The Selective GO TO

The selective GO TO (Option 2) constitutes a multiple branch-point. Control is transferred to one of two or more procedure-names according to the current value of the *data-name* specified in the DEPENDING ON phrase. The *data-name* must have a positive integral value. Control goes to the 1st, 2nd, . . . nth procedure-name as the value of *data-name* is 1, 2, . . . n. If *data-name* should happen to have a value other than an integer in the range 1 to *n*, no transfer takes place and control passes to the next statement after the GO TO statement.

To show the use of the selective GO TO, suppose that in a payroll accounting problem, one of three tax routines is to be used depending upon the period of time involved. The programmer could write:

```
GO TO QUARTERLY-TAX, SEMI-ANNUAL-TAX, ANNUAL-TAX
   DEPENDING ON PERIOD-CODE.
```

In this case, a transfer to one of the three tax routines will occur depending on whether the value of the data item PERIOD-CODE is 1, 2 or 3.

ALTER

The purpose of the ALTER verb is to modify the effect of GO TO sentences elsewhere in the program and thus to change a predetermined sequence of operations. The format for ALTER is:

ALTER *procedure-name-1* TO PROCEED TO *procedure-name-2*

[*procedure-name-3* TO PROCEED TO *procedure-name-4* . . .]

A GO TO sentence that is to be altered must

1. be an unconditional GO TO sentence;
2. be written as a separate paragraph consisting solely of the GO TO sentence, preceded by a procedure-name.

The effect of an ALTER statement is to replace the *procedure-name* specified in the GO TO sentence by the *procedure-name-2* specified in the ALTER. Thus, if the named GO TO sentence,

```
SWITCH-1. GO TO PRIMARY-RUN.
```

were modified by the ALTER statement,

```
ALTER SWITCH-1 TO PROCEED TO SECONDARY-RUN
```

the effect would be to change the GO TO sentence to:

```
SWITCH-1. GO TO SECONDARY-RUN.
```

PERFORM

The PERFORM verb provides a means of departing temporarily from the normal sequence of procedure execution in order to execute some other procedure a specified number of times or until a specified condition is satisfied.

PERFORM has several different formats which vary in complexity. In the simplest format, the procedure referred to is executed just once each time the PERFORM is encountered. Other formats permit repetitive execution, or "looping," of the referenced procedure, using one or more of several optional controls.

The five formats in which the PERFORM verb can be used are:

Option 1

```
PERFORM procedure-name-1 [ THRU procedure-name-2 ]
```

Option 2

```
PERFORM procedure-name-1 [ THRU procedure-name-2 ]
    { integer-1     }
    { data-name-1   } TIME[S]
```

Option 3

```
PERFORM procedure-name-1 [ THRU procedure-name-2 ]
    UNTIL condition-1
```

Option 4

```
PERFORM procedure-name-1 [ THRU procedure-name-2 ]
    VARYING data-name-1 FROM { numeric-literal-1 }
                             { data-name-2       }
    BY { numeric-literal-2 } UNTIL condition-1
       { data-name-3       }
```

Option 5

```
PERFORM procedure-name-1 [ THRU procedure-name-2 ]
    VARYING subscript-name-1 FROM { integer-1   } BY { integer-2   }
                                  { data-name-1 }    { data-name-2 }
    UNTIL condition-1 [ AFTER subscript-name-2 FROM { integer-3   }
                                                   { data-name-3 }
    BY { integer-4   } UNTIL condition-2 ] [ AFTER subscript-name-3
       { data-name-4 }
    FROM { integer-5   } BY { integer-6   } UNTIL condition-3 ]
         { data-name-5 }    { data-name-6 }
```

General Discussion

When a procedure is performed, i.e., executed, the PERFORM transfers sequence control to the first statement of *procedure-name-1* and also provides for return of

control. The point at which control is returned depends upon the structure of the procedure being executed and is determined as follows:

1. If *procedure-name-1* is a paragraph-name and a *procedure-name-2* is not specified, control is returned after the last statement of the *procedure-name-1* paragraph.
2. If *procedure-name-1* is a section-name and a *procedure-name-2* is not specified, control is returned after the last statement of the last paragraph of the *procedure-name-1* section.
3. If *procedure-name-2* is specified and is a paragraph-name, control is returned after the last statement of the *procedure-name-2* paragraph.
4. If *procedure-name-2* is specified and is a section-name, control is returned after the last statement of the last paragraph of the *procedure-name-2* section.

Note: The sentence containing the "last statement" referred to in each of the above cases must not include an unconditional GO TO statement.

When *procedure-name-2* is specified, the only required relationship between *procedure-name-1* and *procedure-name-2* is that of logical sequence. That is, execution sequence must proceed from *procedure-name-1* to the last statement of the *procedure-name-2* paragraph or section. GO TO statements and other PERFORM statements are permitted between *procedure-name-1* and the last statement of *procedure-name-2,* provided that the sequence ultimately returns to the final statement of *procedure-name-2.* If the logic of a procedure requires a conditional exit prior to the final sentence, the EXIT verb is used in order to comply with the foregoing requirements. In this case, *procedure-name-2* must be the name of a paragraph consisting solely of the verb EXIT; all paths must lead to this point. (See the discussion of EXIT.)

A procedure referenced by one PERFORM statement can be referenced by other PERFORM statements. Moreover, a procedure referenced by one or more PERFORMS can also be executed by "dropping through," that is, by entering the procedure through the normal passage of control from one statement to the next in sequence. Accordingly, *procedure-name-1* normally should not be the next statement after the PERFORM. If it were the next statement, the procedure probably would be executed one more time than was intended because, after execution of the PERFORM, control would pass to *procedure-name-1* in the normal continuation of sequence.

The Simple PERFORM

Option 1 of the formats shows the simple PERFORM. A procedure referenced by this type of PERFORM statement is executed once and then control passes to the next statement after the PERFORM.

Some examples of the simple PERFORM are:

```
PERFORM INVENTORY-ANALYSIS
PERFORM GROSS-PAY THRU NET-PAY
```

The TIMES Option

Option 2 is the TIMES option. This form provides a means of performing a procedure repetitively a specified number of times. The number of times, whether stated as a number or as a *data-name,* must have a positive integral value and can be zero.

When the TIMES option is used, a counter is set up and this counter is tested against the specified number of executions (the TIMES) *before* control is sent to *procedure-name-1.* After control is returned, the counter is increased by 1 and is tested

again. This process is repeated until the value of the counter equals the specified number of executions, whereupon control passes to the statement following the PERFORM statement. If the initial value is zero, there will be no executions.

These examples illustrate the use of the TIMES option:

```
PERFORM MONTHLY-INTEREST 3 TIMES.
PERFORM CREDIT-CARD-ISSUE NO-OF-COPIES TIMES.
```

The UNTIL Option

The UNTIL option (Option 3) is essentially the same as the TIMES option except that no counting takes place and the PERFORM causes evaluation of the specified conditional expression instead of testing the value of a counter against a specified number of executions.

Condition-1 can be any simple or compound conditional expression as described in Chapter 3. The conditional expression is evaluated *before* the specified procedure is executed; if it is found to be unsatisfied, i.e., not true, control passes to *procedure-name-1*, the procedure is executed once and control returns to the PERFORM. This process is repeated until such time as the conditional expression is determined to be true, at which point control goes to the next statement after the PERFORM. Note that if the conditional expression is true when the PERFORM is encountered the specified procedure will not be executed.

Typical PERFORM statements utilizing the UNTIL option are:

```
PERFORM RE-ORDER UNTIL ON-ORDER + ON-HAND =
    MONTHLY-USAGE * 2.5
PERFORM SALES-ANALYSIS THRU SALES-REPORT UNTIL
    STATE-CODE IS GREATER THAN 50
```

The VARYING data-name *Option*

The VARYING *data-name* option (Option 4) makes it possible to PERFORM a procedure repetitively, increasing or decreasing the value of a data item once for each repetition, until a specified conditional expression is satisfied. The value of only one data-name can be varied in a PERFORM statement using this option.

In this form, the PERFORM first sets the value of *data-name-1* equal to the specified initial value (the FROM value) and then causes the conditional expression (the UNTIL *condition*) to be evaluated. If the expression is true at this point, no execution of the procedure takes place and control goes to the statement immediately following the PERFORM. If the expression is false, the procedure is executed once, after which the PERFORM augments the value of *data-name-1* by the specified increment or decrement (the BY value) and again causes the conditional expression to be evaluated. This process continues until the conditional expression is found to be true; thereupon, control passes to the next statement after the PERFORM. The items used in the BY and FROM clauses may have any numeric value and need not be integers; such values may be positive, negative, or zero. A diagram illustrating this option of the PERFORM verb is given in Figure 7-5.

It will be noted from the diagram (Figure 7-5) that after execution of the PERFORM, the value of *data-name-1* will be one increment (or decrement) greater than (or less than) its last-used value.

To show the use of this type of PERFORM, suppose that a manufacturing concern has developed a formula for pricing its products and that a routine called PRICING-FORMULA has been programmed to apply the formula. For a particular product, management might want to know how the computed price would be affected if

one of the factors involved in the formula varied between certain limits. The desired information could be obtained by repeated use of the PRICING-FORMULA routine, varying the factor in question and holding other factors constant. To accomplish this, the programmer could use a PERFORM statement such as this:

```
PERFORM PRICING-FORMULA VARYING FACTOR-X
    FROM -1.00 BY .05 UNTIL FACTOR-X = .50
```

The execution of this statement would result in repetitive execution of the PRICING-FORMULA routine, using a different value of FACTOR-X and producing a different computed result (price) for each of thirty iterations. The value of FACTOR-X would be −1.00 in the first iteration, .45 in the last iteration, and .50 after completion of the PERFORM. Note that UNTIL means "until but not including."

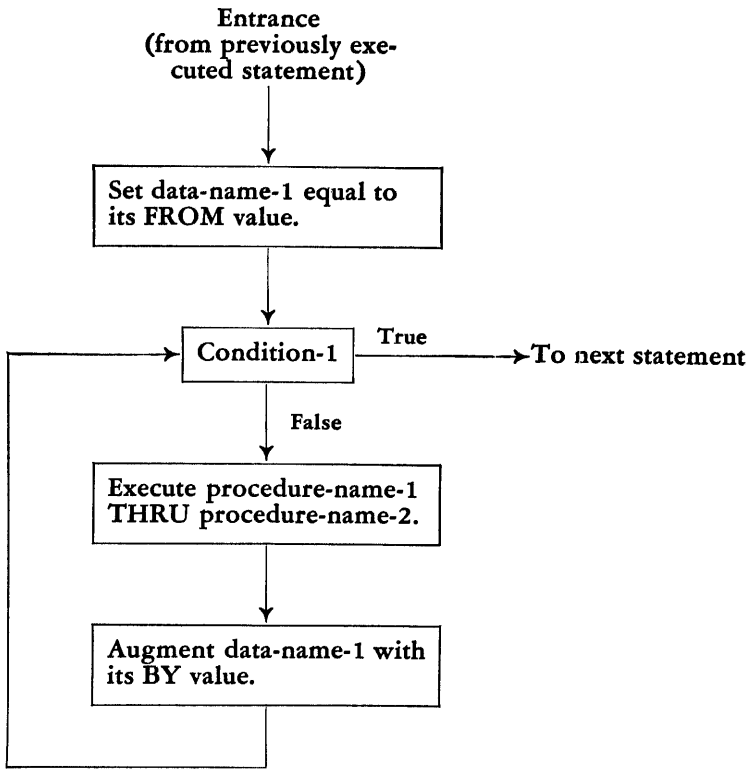


Figure 7-5. Functioning of the PERFORM verb when the VARYING *data-name* option is used.

The VARYING subscript-name *Option*

Option 5, the VARYING *subscript-name* option, is the most powerful form of the PERFORM verb. It is used when the programmer wishes to increment the value of one or more subscripts in a "nested" fashion in conjunction with repetitive execution of a procedure. As indicated by the format of this option, a maximum of three subscripts can be varied in a given PERFORM statement.

When only one subscript is being varied, this option functions in the same manner as the VARYING *data-name* option. (See Figure 7-5.)

The functioning of the VARYING *subscript-name* option employing two subscripts is shown in detail in Figure 7-6. After the two subscripts are set to their initial (FROM) values, *condition-1* is first evaluated; if it is found to be unsatisfied, i.e., false, then *condition-2* is evaluated. If *condition-2* is unsatisfied, the specified procedure is executed once for each value of *subscript-name-2* until *condition-2* is

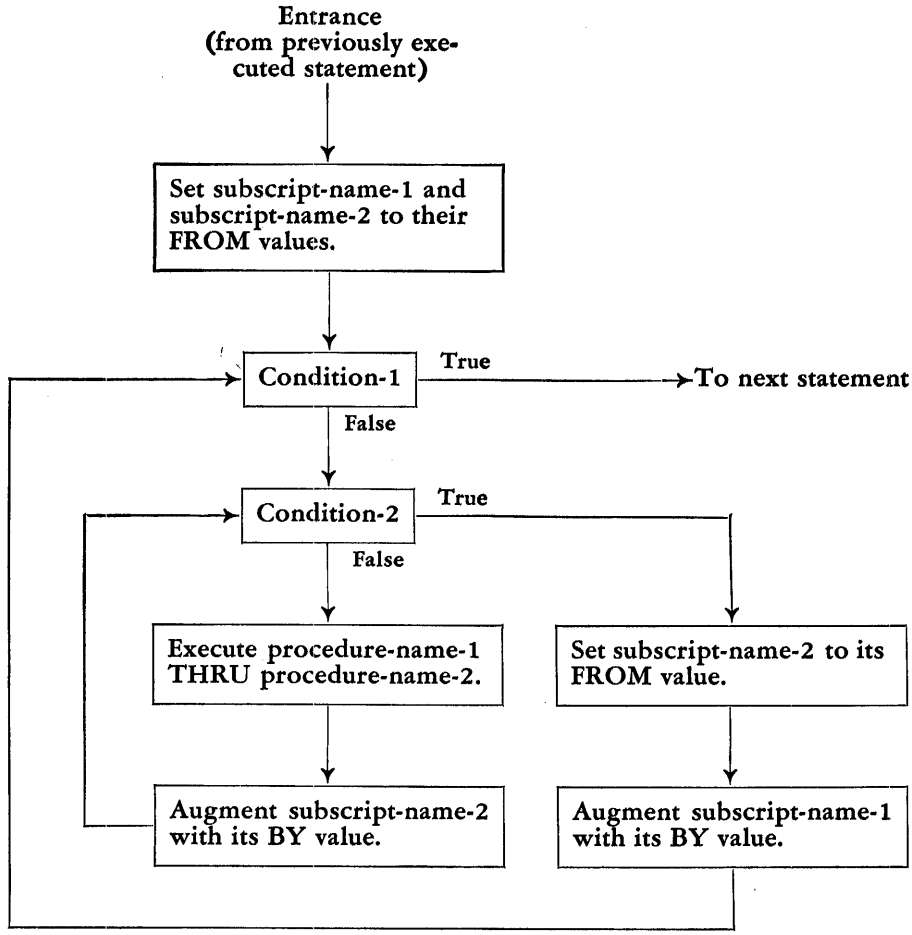


Figure 7-6. Functioning of the PERFORM verb when the VARYING *subscript-name* option employing two subscripts is used.

found to be true. At this point, *subscript-name-2* is reset to its initial value, *subscript-name-1* is increased or decreased by its BY value, and *condition-1* is again evaluated. If *condition-1* remains unsatisfied, the procedure is again executed once for each value of *subscript-name-2* until *condition-2* is true. This process continues until *condition-1* is determined to be true, whereupon the PERFORM is completed and control goes to the statement immediately following the PERFORM statement.

Figure 7-7 shows how this option functions when three subscripts are involved. It operates in the same manner as described above except that a third dimension is added. The subscripts are first set to their initial values. Then, the value of *subscript-name-3* goes through a complete cycle for each value of *subscript-name-2* which in turn goes through a complete cycle for each value of *subscript-name-1*.

It is important to note that, regardless of the number of subscripts involved, a PERFORM statement of this type is complete as soon as *condition-1* is found to be true. As the diagrams indicate, *condition-1* is evaluated before the procedure is executed the first time. Accordingly, if *condition-1* is true when the PERFORM statement is encountered, the procedure will not be executed even though *condition-2* and *condition-3* may not be true. In addition, the following rules apply, regardless of the number of subscripts that are varied:

1. The initial (FROM) value of a subscript must be a positive, non-zero integer.
2. The increment (i.e., the BY value) must be a non-zero integer. (It *can* be negative.)

3. *Subscript-name-1, subscript-name-2* and *subscript-name-3* cannot refer to the same item; i.e., they must not be alternative names for the same data item.

After completion of a PERFORM, the values of *subscript-name-2* and *subscript-name-3* are equal to their respective initial (FROM) values, while *subscript-name-1* has a value exceeding its initial value by one increment (or decrement).

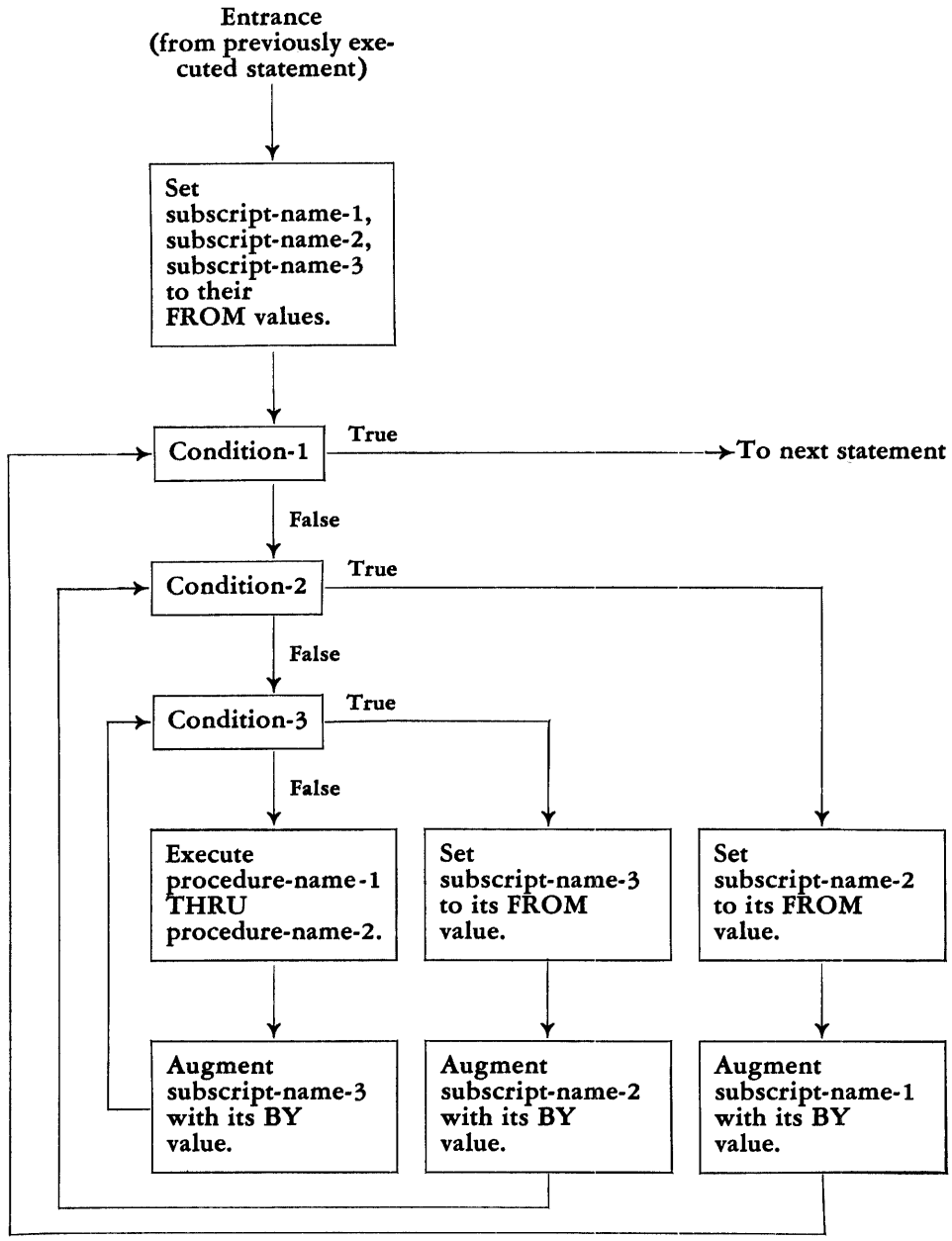


Figure 7-7. Functioning of the PERFORM verb when the VARYING *subscript-name* option employing three subscripts is used.

To help clarify the way in which the VARYING *subscript-name* option is used, suppose that a rate table is employed in a billing procedure and that the table requires periodic updating. This hypothetical rate table is three-dimensional, being divided into five regions, each of which includes ten states, each of which contains the rates for ten cities. It is assumed further that an appropriate rate-updating procedure is available elsewhere in the program. Such a procedure might appear as:

```
RATE-UPDATING. MULTIPLY RATE (REGION, STATE, CITY) BY
    ADJUST-FACTOR GIVING RATE (REGION, STATE, CITY).
    IF RATE (REGION, STATE, CITY) IS GREATER THAN MAX-
    RATE MOVE MAX-RATE TO RATE (REGION, STATE, CITY).
```

It is desired to execute this RATE-UPDATING procedure once for each city of each state in each region, using the current rate for a given city and producing an adjusted rate for that city. Accordingly, the programmer employs a PERFORM statement with the VARYING *subscript-name* option:

```
PERFORM RATE-UPDATING VARYING REGION FROM 1 BY 1
    UNTIL REGION IS GREATER THAN 5 AFTER STATE FROM
    1 BY 1 UNTIL STATE = 11 AFTER CITY FROM 1 BY 1 UNTIL
    CITY IS GREATER THAN 10.
```

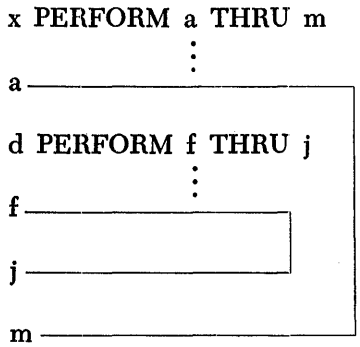
One feature of this example requires amplification. The MAX-RATE item in the RATE-UPDATING procedure is a constant that limits the value of any rate. If this limit is exceeded, the maximum rate is used in lieu of the computed (new) rate.

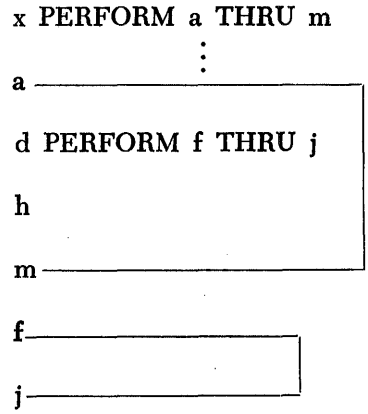
When the PERFORM is executed at object time, the RATE-UPDATING procedure is executed first for the first city of the first state in region 1, then for the next city, and so on. The PERFORM is complete when the procedure has been executed for the tenth city of the tenth state of region 5, by which time the procedure will have been executed 500 times.

"Nested" PERFORM Statements

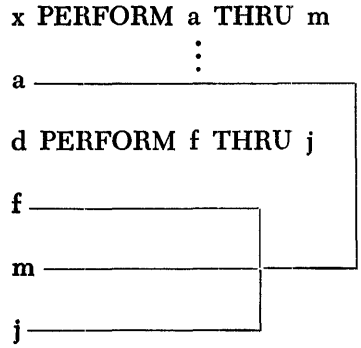
If a procedure referenced by a PERFORM statement includes another PERFORM statement, the procedure associated with the inner PERFORM must be either entirely included in, or entirely excluded from, the procedure related to the outer PERFORM.

For example, the cases shown below are correct:



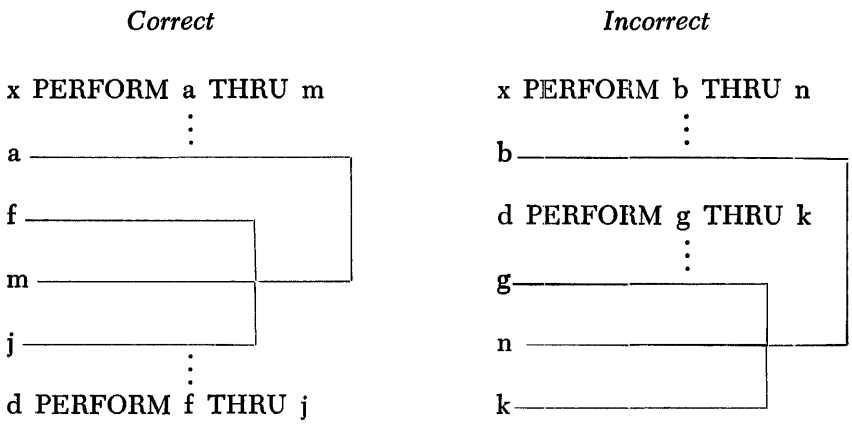


whereas, the following is incorrect:



However, a procedure associated with one PERFORM can overlap or intersect the procedure associated with another PERFORM, provided that neither procedure includes the PERFORM associated with the other procedure.

For example:



These rules and examples apply to all five options of the PERFORM verb.

STOP

The STOP verb permits the programmer to specify a temporary or final halt in the object program. Its format is:

$$\underline{\text{STOP}} \left\{ \begin{matrix} \textit{literal} \\ \underline{\text{RUN}} \end{matrix} \right\}$$

When a literal is used in a STOP statement, the object program will display the literal at the time the STOP occurs. Following execution of a STOP *literal* statement, continuation of the object program begins with the next statement in sequence.

The STOP RUN option is used to indicate an end-of-program halt. Because of its terminal effect, the STOP RUN option can be used only as the final statement of the sequence in which it appears. Otherwise, the succeeding statements would never be executed. The action following execution of a STOP RUN statement depends upon the procedures established for a given installation and/or a particular computer. This subject will be discussed in more detail in the publications dealing with the respective processors.

Some examples of STOP statements are:

```
STOP  0034
STOP  RUN
STOP  9
STOP  'END OF INTERMEDIATE PHASE'
```

When numeric literals are employed in STOP statements, as in the first and third examples above, the usual practice is to specify a different number for each STOP in the source program. These numbers can then be used as keys to a list of the various STOPs and their respective meanings in the program.

Processor Verbs

The COBOL processor verbs are instructions directed to the processor; they cause the processor to take certain specific action. Two of the three processor verbs have an *indirect* effect on the object program. However, the other, the NOTE verb, has no effect whatsoever on the object program.

ENTER

The ENTER verb enables the programmer to use other programming languages in a COBOL source program. This facility makes it possible to incorporate into a COBOL program existing routines written in another language. The format in which ENTER is used is:

ENTER *language-name.*

The *language-name* informs the processor as to what kind of "other language" statements are to be inserted at this point in the procedure. The publications for the respective processors will specify which languages can be entered.

The "other language" statements must be written in-line immediately following the ENTER statement and they must be followed by an ENTER COBOL entry to indicate the point at which the COBOL source language is resumed. Each ENTER statement must constitute a separate paragraph in the source program.

An example of the use of ENTER is given below. Note that each ENTER statement is preceded by its paragraph-name:

```
LEAVE.  ENTER AUTOCODER.
          .
          .
          .
RETURN.  ENTER COBOL.
```

Note: All statements between an ENTER *language-name* statement and the following ENTER COBOL statement must conform to the rules of the named language.

EXIT

The EXIT verb is used when it is necessary to provide an end point for a procedure that is to be executed by means of a PERFORM. While EXIT is classified as a processor verb because it supplies the processor with necessary information and does not produce any coding in the object program, it can be thought of also as a "dummy" program verb. Its format is simply:

EXIT.

As mentioned in the discussion of the PERFORM verb, the logic of a procedure referenced by a PERFORM may involve a conditional exit prior to the last sentence. When this is the case, the EXIT verb must be used to provide an ending point common to all paths. This is illustrated in the following PERFORM statement and associated procedure:

```
. . . PERFORM ANALYSIS-ROUTINE THRU FINISH-ANALYSIS
                .
                .
                .
ANALYSIS-ROUTINE. COMPUTE RETURNS-RATIO = RETURNS
    / (ORDERS-FILLED + BACK-ORDERS - RETURNS). IF RE-
    TURNS-RATIO IS LESS THAN .20 GO TO FINISH-ANALYSIS.
    IF RETURNS-RATIO IS LESS THAN .33 ADD 1 TO HIGH-
    RATIO-COUNTER GO TO FINISH-ANALYSIS. PERFORM
    HIGH-RATIO-REPORT.
FINISH-ANALYSIS. EXIT.
```

In this example there are two points at which execution of the procedure may terminate prior to the last statement; i.e., either of the two conditional sentences can cause the remainder of the procedure to be bypassed. Accordingly, an EXIT is required and all paths lead to it.

As indicated in the example, EXIT must appear in the source program as a one-word paragraph, preceded by a paragraph-name.

NOTE

The NOTE verb is used for inserting statements and comments in the source program to explain or annotate the procedures being defined. A NOTE appears in the program listing but has no effect on the object program. The format of a NOTE statement is:

NOTE *any comment.*

Any combination of characters from the COBOL character set can follow the word NOTE. The combination of characters may constitute a sentence or a paragraph according to these rules:

1. If NOTE is the first word of a paragraph, the entire paragraph must be devoted to the note(s). The paragraph must be named and all other format rules for paragraph structure must be observed.
2. If NOTE is not the first word of a paragraph, the commentary is terminated by a period followed by a space.

Some examples of NOTE are:

```
. . . NOTE END OF FIRST PHASE.
NOTE-1. NOTE THAT THE WORD 'NOTE-1' IS THE PARAGRAPH-
    NAME OF THIS PARAGRAPH. THE VERB, NOTE, IS THE
    FIRST WORD OF THE FIRST SENTENCE.
```

Chapter 8: Environment Division

Introduction

In the COBOL system, all aspects of the total data processing problem that depend on the physical characteristics of a specific computer are segregated in one portion of the source program known as the Environment Division. Thus, the primary functions of the Environment Division are to describe the computer system on which the object program is to run and to establish the necessary links between the other divisions of the source program and the characteristics of the computer.

Since the Environment Division is completely machine-oriented, it follows that it must be rewritten each time the source program is to be processed for a different computer. Re-writing and reprocessing may also be necessary if the program is to be run on a *different configuration* of the computer for which it was written originally.

Organization

The Environment Division of a COBOL source program consists of two sections, each of which has a fixed section-name—CONFIGURATION and INPUT-OUTPUT.

The CONFIGURATION section is concerned with the specifications of computers. It comprises three paragraphs which also have fixed names: The SOURCE-COMPUTER paragraph names the computer on which the COBOL processor is to be run. The OBJECT-COMPUTER paragraph identifies and describes the machine system on which the object program is to be run. The SPECIAL-NAMES paragraph is used to relate the machine names (i.e., the names of machine components and devices) used by a particular processor to the names used by the programmer in his source program.

The INPUT-OUTPUT section deals with the external media of the data to be processed by the object program and with techniques for handling the data. This section consists of two paragraphs, again with fixed paragraph-names: The FILE-CONTROL paragraph names the data files and specifies the external media with which they are to be associated. The I-O-CONTROL paragraph is used to designate special input/output techniques.

The overall structure of the Environment Division of a source program is shown below:

```
ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
        SOURCE-COMPUTER. computer-name.
        OBJECT-COMPUTER. computer-name....
        SPECIAL-NAMES. device-name... switch-name....
    INPUT-OUTPUT SECTION.
        FILE-CONTROL. SELECT....
        I-O-CONTROL. APPLY....
```

Each of the five paragraphs of the Environment Division is discussed in turn in the balance of this chapter. The discussion is in general terms, however. No attempt is made to provide information regarding specific machine systems or the respective processors. Such detailed information will be published separately for each of the IBM data processing systems involved.

Configuration Section

Source-Computer

This paragraph of the Environment Division is used to name the computer on which the COBOL processor is to be run, i.e., on which the source program is to be processed. The format is simply:

SOURCE-COMPUTER. *computer-name.*

The SOURCE-COMPUTER paragraph can be thought of as being essentially a comment or NOTE. The *computer-name* is a fixed name that will be specified in the publications for the respective processors.

Object-Computer

The purpose of this paragraph is to describe the computer on which the object program is to be run and to permit the programmer to specify the input device from which the program will be read at object time. The format of the OBJECT-COMPUTER paragraph is:

Option 1

OBJECT-COMPUTER. COPY *library-name.*

Option 2

OBJECT-COMPUTER. *computer-name*

[ASSIGN OBJECT-PROGRAM TO *device-name-1*]

[MEMORY SIZE { *integer-1* {WORDS / CHARACTERS} / ADDRESS *integer-2* THRU *integer-3* }]

[[*integer-4*] *device-name-2* [[*integer-5*] *device-name-3* . . .]].

The COPY *library-name* option (i.e., Option 1) can be used if the library contains a complete description of the object computer. Otherwise, the object computer must be defined utilizing Option 2.

Fixed *computer-names* and *device-names* will be assigned in the publications for the individual processors. Moreover, *computer-name* will represent a particular configuration of equipment, including memory size, memory addresses, and type and number of devices.

The MEMORY SIZE specified by the programmer may, however, be greater or smaller than that of the implicit configuration. In either case, the processor will adjust MEMORY SIZE accordingly. Should the requirements of the program exceed the specified configuration, the processor will indicate an error condition at process time.

The last clause of the OBJECT-COMPUTER paragraph is used in specifying a configuration of equipment other than the particular configuration implied by *computer-name.* The device-names designated in this clause may include input/output de-

vices and machine features such as index registers, floating-point arithmetic devices, additional machine instructions, etc.

When the ASSIGN OBJECT-PROGRAM clause is employed, the processor assigns the *device-name* unit as the input unit from which the object program will be read at object time. If this clause is omitted, a standard input device (predetermined for each processor) will be assigned for this purpose.

Special-Names

The function of the SPECIAL-NAMES paragraph is to equate mnemonic names with the standard names for actual machine devices or switches, and condition-names with the standard names for the status of actual machine switches. The format is:

Option 1

```
SPECIAL-NAMES.  COPY library-name.
```

Option 2

```
SPECIAL-NAMES.

    [ device-name-1 IS mnemonic-name-1 [ device-name-2 IS
        mnemonic-name-2 . . . ] ].

    [ switch-name-1 [ IS mnemonic-name-3 ] [ ON STATUS IS
        condition-name-1 ] [ OFF STATUS IS condition-name-2 ]
        [ switch-name-2 . . . ] ].
```

The entire SPECIAL-NAMES paragraph can be omitted if no *condition-names* (pertaining to machine switches) or *mnemonic-names* appear in the Procedure Division of the source program. Option 1 is used when the library contains a complete description of all the SPECIAL-NAMES used in the program; otherwise Option 2 must be employed.

Each *mnemonic-name* used in ACCEPT or DISPLAY statements (in the Procedure Division) must have a standard *device-name* assigned to it in the SPECIAL-NAMES paragraph. (The standard *device-names* will be specified in the publications for the respective processors.) It should be noted that *mnemonic-names* cannot be used in a source program except in those verb formats which specifically permit their usage.

A machine switch can be referred to in the Procedure Division either by means of a *mnemonic-name* or by means of one or two *condition-names* associated with the ON/OFF status of the switch. In this event, the SPECIAL-NAMES paragraph is used to assign a standard *switch-name* to each *mnemonic-name* and/or to relate the switch's ON/OFF status to one or two *condition-names*. Thus, as indicated in the paragraph format, a given *switch-name* can have one, two, or all three of the optional clauses associated with it.

In the Procedure Division of the program, the status of a machine switch is interrogated by means of conditional expressions utilizing the *condition-name(s).*

Input-Output Section

File-Control

The FILE-CONTROL paragraph is used to name each file, identify its media, and to assign it to one or more input/output devices. Provision is made for specifying alternate input/output areas. The format of this paragraph is:

Option 1

```
FILE-CONTROL. COPY library-name.
```

Option 2

```
FILE-CONTROL. SELECT [OPTIONAL] file-name-1

    [RENAMING file-name-2] ASSIGN TO [integer-1] device-name-1

    [device-name-2 . . .] [FOR MULTIPLE REEL]

    [RESERVE {integer-2 / NO} ALTERNATE AREA[S]].

    [SELECT . . .].
```

Option 1 is used when the complete description of FILE-CONTROL is available in the library. Otherwise the programmer must employ Option 2 and specify the details.

Each file to be processed by the object program must be named in a SELECT *file-name* entry; the designated name must be unique within the source program. Each file employed in the program must also be assigned to an input or output medium, i.e., to a *device-name.* The *device-names* used in making such assignments are fixed names that will be specified in the publications for the respective processors. The word OPTIONAL must appear immediately following the word SELECT if the file being named is an input file that will not necessarily be present each time the object program is run. (See the discussion of the OPEN, READ, and CLOSE verbs in Chapter 7.)

The *file-name-1* file must be described by a File Description in the Data Division of the source program unless *file-name-1* is RENAMING another file, i.e., *file-name-2,* for which a File Description is given. The RENAMING option is used when the File Description of one file *(file-name-2)* is applicable to another file *(file-name-1),* as in the case of a file that is to be processed as an input file and as an output file in the same program. The RENAMING of a file implies a sharing of a single File Description; it does *not* allow the two names to be used interchangeably in the program.

In assigning a file to an input or output medium, *integer-1* can be used only if *device-name-1* designates magnetic tape as the medium (as opposed to specific tape units). In this case, *integer-1* indicates the number of tape units to be assigned to the file. If desired, however, the programmer can omit *integer-1* and designate specific tape unit assignments using specific *device-names,* i.e., *device-name-1,*

device-name-2, etc., which will be provided for each processor. If *integer-1* is omitted for a tape file and *device-name-1* specifies the tape medium, then the processor will determine the number of tape units to be used, based on the number available (as specified in the OBJECT-COMPUTER paragraph) and the number required for other files. To provide the processor with the necessary information, the MULTIPLE REEL option must be included when:

1. *Integer-1* is not specified and the file may contain more than one reel.
2. *Integer-1* is specified but the file may contain more than that number of reels.

The RESERVE clause of the FILE-CONTROL paragraph makes it possible to modify the standard number of input/output areas allocated for a given file by the processor. The programmer can designate a certain number *(integer-2)* of *additional* alternate areas to be reserved, or he can specify that NO alternate areas are to be reserved.

I-O-Control

The I-O-CONTROL paragraph permits the programmer to specify input/output techniques and to establish rerun, or restart, provisions that are implemented by the input/output control system. The format of the paragraph is:

Option 1

I-O-CONTROL. COPY *library-name.*

Option 2

I-O-CONTROL. [APPLY *input-output technique* ON *file-name*] . . . .

[RERUN [ON {*file-name-1* / *device-name*}] EVERY END OF REEL OF *file-name-2.*]

This paragraph need be included in the Environment Division of a source program only if one or both of its features are desired. The COPY option can be used if the library contains a complete description of I-O-CONTROL; otherwise, Option 2 is employed to specify the details.

The input/output control system for a given computer may provide alternative input/output techniques. The APPLY *input-output technique* clauses in Option 2 are used to select the technique appropriate to each specified file.

Using the second clause in Option 2, RERUN points can be established for each end of reel of a file. Since the ON clause has two alternate forms and is itself optional, there are three ways to specify how the RERUN information is to be treated, i.e., where the contents of memory is to be placed:

1. If the ON clause is omitted, the contents of memory will be written on each reel of the *file-name-2* file, which, in this case, must be an output file. The publication for each processor will specify where on the reel the information will appear.
2. When the ON *file-name-1* option is employed, the contents of memory will be written in *file-name-1* (which must be an output file) each time an end of reel occurs in *file-name-2*. In this case, *file-name-2* can be either an input file or an output file. Again, the respective processor publications will specify where the RERUN information will appear in *file-name-1*.
3. If the ON *device-name* option is used, memory will be written on a separate RERUN tape, i.e., on a tape unit designated by *device-name*, each time an end of reel occurs in the *file-name-2* file.

Chapter 9: Identification Division

The fourth major part of a COBOL source program, the Identification Division, is used to identify or label the program and to provide any other pertinent information concerning the program. As is noted in Chapter 5, the Identification Division precedes the other divisions when the source program is presented to the processor at process time.

The format of the Identification Division is relatively brief and straightforward. The format for the entire division is:

IDENTIFICATION DIVISION.

PROGRAM-ID. *program-name.*

[AUTHOR. *author-name.*]

[INSTALLATION. *any sentence or group of sentences.*]

[DATE-WRITTEN. *any sentence or group of sentences.*]

[DATE-COMPILED. *any sentence or group of sentences.*]

[SECURITY. *any sentence or group of sentences.*]

[REMARKS. *any sentence or group of sentences.*]

The Identification Division is essentially an extended NOTE (see Chapter 7). The information supplied to the processor becomes a part of the program listing, but it has no effect on the object program.

Fixed paragraph-names are used throughout the Identification Division. As the format indicates, only the PROGRAM-ID paragraph is required; the other paragraphs are optional and the programmer can specify any or all of them. Thus, the Identification Division of a source program can consist of from one to seven paragraphs.

The format presented above is largely self-explanatory. It should be noted, however, that the PROGRAM-ID paragraph must always appear as the first paragraph of the Identification Division. The *program-name* designated in this paragraph must be either a name or a literal as defined in Chapter 3. The *program-name* should be used in referring to the source program, the object program, and all associated documentation.

The Identification Division of a typical program might be written as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INVENTORY-MAINTENANCE.
AUTHORS. J. DOE AND R. SMITH.
INSTALLATION. ACCOUNTING DEPT., XYZ MANUFACTURING
    CORP.
DATE-WRITTEN. JUNE 5, 1961.
REMARKS. DESIGNED FOR WEEKLY UPDATING OF INVEN-
    TORY-MASTER FILE. INPUT IS FROM RUN 10 AND OUTPUT
    IS USED IN RUN 17.
```

Appendix A: Supplementary Reference Material

Conditional Expressions

Conditional expressions may contain conditions and the logical operators AND, OR, and NOT. Subexpressions may be contained in parentheses as required.

Conditions

One or more conditions are contained in a conditional expression. The five forms in which conditions may be written are given below:

1. Simple Relational Conditions

$$\left\{\begin{array}{l}\textit{data-name}\\ \textit{literal}\\ \textit{arithmetic expression}\end{array}\right\} \left\{\begin{array}{l}\text{IS [NOT] GREATER THAN}\\ \text{IS [NOT] LESS THAN}\\ \text{IS [NOT] EQUAL TO}\\ =\end{array}\right\} \left\{\begin{array}{l}\textit{data-name}\\ \textit{literal}\\ \textit{arithmetic expression}\end{array}\right\}$$

---

2. Sign Conditions

$$\left\{\begin{array}{l}\textit{arithmetic expression}\\ \textit{data-name}\end{array}\right\} \text{ IS [NOT] } \left\{\begin{array}{l}\text{POSITIVE}\\ \text{NEGATIVE}\\ \text{ZERO}\end{array}\right\}$$

---

3. Class Conditions

$$\textit{data-name} \text{ IS [NOT] } \left\{\begin{array}{l}\text{NUMERIC}\\ \text{ALPHABETIC}\end{array}\right\}$$

---

4. Condition-Names

condition-name

---

5. Switch-Status-Names

switch-status-name

Evaluation of Conditional Expressions

If the i-th conditional expression (such as MARRIED OR PAY IS GREATER THAN 2 * X + Y) is designated by the symbol Ci the following rules may be stated concerning the formation of conditional expressions involving Ci, and the logical operators NOT, AND, and OR:

1.

The Conditional Expression	Is True If
C1	C1 is true
NOT C1	C1 is false
C1 AND C2	Both C1 and C2 are true
C1 OR C2	Either C1 is true, C2 is true, or both are true
NOT (C1 AND C2)	C1 is false, C2 is false, or both are false
NOT (C1 OR C2)	C1 and C2 are both false

2. If C1 and C2 are conditional expressions, then "C1 AND C2" and "C1 OR C2" are conditional expressions, as are similar expressions formed with the use of NOT. Thus, an expression of the form

   C1 AND (C2 OR NOT (C3 OR C4))

   may be successively reduced as follows:

   Let C5 equal "C3 OR C4" → C1 AND (C2 OR NOT C5)
   Let C6 equal "C2 OR NOT C5" → C1 AND C6
   Let C7 equal "C1 AND C6" → C7

   This rule indicates how conditional expressions may be formed from conditional expressions.

3. The conditional expression "C1 OR C2 AND C3" is identical with "C1 OR (C2 AND C3)" but is not the same as "(C1 OR C2) AND C3." In other words, conditional expressions are grouped first according to AND and subsequently by OR. However, the programmer's use of parentheses will affect the order of grouping.

4. The rules for formation of symbol pairs are contained in the table below. The letter P indicates that the specified pair is permissible, while the dash indicates that it is not.

		Second Symbol					
		C	OR	AND	NOT	(	)
First Symbol	C	–	P	P	–	–	P
	OR	P	–	–	P	P	–
	AND	P	–	–	P	P	–
	NOT	P†	–	–	–	P	–
	(	P	–	–	P	P	–
	)	–	P	P	–	–	P

†Permissible only if the condition itself does not contain a NOT.

Simple Relational Conditions with Implied Subjects and Implied Relational Operators

Only simple relational conditions may have implied subjects. If S_1, R_1, and O_1 are the subject, relational operator, and object respectively of the first simple relational condition of a series, and if S_n, R_n, and O_n are the components of the nth simple relational condition of the series; then the following diagram shows the general form for a series of consecutive simple relational conditions with implied subjects:

$$S_1\ R_1\ O_1\ \left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ R_2\ O_2\ \left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ R_3\ O_3\ \ldots\ \left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ R_n\ O_n$$

A relational operator can be implied only when a subject is also implied. The general form for writing consecutive simple relational conditions with implied subjects and relational operators is given below using the notation of the above diagram:

$$S_1\ R_1\ O_1\ \left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ O_2\ \left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ O_3\ \ldots\ \left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ O_n$$

Conditional Statements

The following diagrams show the three ways in which a conditional statement may be written:

Option 1

IF *conditional expression* *statement-1*

Option 2

IF *conditional expression* $\left\{\begin{matrix}\textit{statement-1}\\ \underline{\text{NEXT SENTENCE}}\end{matrix}\right\}$ $\left\{\begin{matrix}\underline{\text{OTHERWISE}}\\ \underline{\text{ELSE}}\end{matrix}\right\}$ $\left\{\begin{matrix}\textit{statement-2}\\ \underline{\text{NEXT SENTENCE}}\end{matrix}\right\}$

Option 3

$\left\{\begin{matrix}\textit{statement-1}\ \text{AT}\ \underline{\text{END}}\\ \textit{statement-2}\ \text{ON}\ \underline{\text{SIZE}}\ \underline{\text{ERROR}}\end{matrix}\right\}$ *statement-3*

Arithmetic Expressions

Arithmetic expressions may contain numeric literals, names of variables, names of constants, arithmetic operators, and the figurative constant ZERO (including ZEROS and ZEROES). Sub-expressions may be enclosed in parentheses as required. The rules for forming arithmetic expressions are:

1. Arithmetic operators must be chosen from the following list:

Addition	+
Subtraction	−
Multiplication	*
Division	/
Exponentiation	**

2. The ways in which symbol pairs may be formed are summarized in the table below. The letter P indicates that the specified pair is permissible, while the dash indicates that it is not.

		Second Symbol				
		Variable	* or / or **	+ or −	(	)
First Symbol	Variable	−	P	P	−	P
	* or / or **	P	−	P	P	−
	+ or −	P	−	−	P	−
	(	P	−	P	P	−
	)	−	P	P	−	P

3. When the hierarchy of operations in an expression is not completely specified by parentheses, the order of operations is assumed to be exponentiation, then multiplication and division, and finally addition and subtraction. Thus the expression A + B / C + D ** E * F − G will be taken to mean A + (B / C) + (D^E * F) − G.

4. When the order of a sequence of consecutive operations on the same hierarchal level (i.e., consecutive multiplications and divisions or consecutive additions and subtractions) is not completely specified by parentheses, the order of operations is assumed to be from left to right. Thus, certain expressions ordinarily considered ambiguous are permitted in COBOL. For example, A / B * C and A / B / C are taken to mean (A / B) * C and (A / B) / C. The more complex expression A * B / C * D is taken to mean ((A * B) / C) * D.

5. The expression A^{B^C} cannot be written as A ** B ** C; it should be written as either (A ** B) ** C or A** (B ** C), whichever is intended.

List of COBOL Verb Forms

The general forms of all of the COBOL verbs are presented below in alphabetical order for reference purposes.

ACCEPT *data-name* [FROM *mnemonic-name*]

ADD {*data-name-1* / *literal-1*} [{*data-name-2* / *literal-2*} ...]
[{TO / GIVING} *data-name-n*] [ROUNDED]
[ON SIZE ERROR *any imperative statement*]

ALTER *procedure-name-1* **TO PROCEED TO** *procedure-name-2*
[*procedure-name-3* **TO PROCEED TO** *procedure-name-4* . . .]

CLOSE *file-name-1* [**REEL**] [**WITH** { **LOCK** / **NO REWIND** }] [*file-name-2* . . .]

COMPUTE *data-name-1* [**ROUNDED**] = *arithmetic expression*
[**ON SIZE ERROR** *any imperative statement*]

DISPLAY { *data-name-1* / *literal-1* } [{ *data-name-2* / *literal-2* } . . .] [**UPON** *mnemonic-name*]

DIVIDE { *data-name-1* / *literal-1* } **INTO** { *data-name-2* / *literal-2* }
[**GIVING** *data-name-3*] [**ROUNDED**]
[**ON SIZE ERROR** *any imperative statement*]

ENTER *language-name.*

EXAMINE *data-name* { **TALLYING** { **ALL** / **LEADING** / **UNTIL FIRST** } *literal-1* [**REPLACING BY** *literal-2*] / **REPLACING** { **ALL** / **LEADING** / [**UNTIL**] **FIRST** } *literal-3* **BY** *literal-4* }

EXIT.

Option 1

GO TO [*procedure-name*]

Option 2

GO TO *procedure-name-1 procedure-name-2* [*procedure-name-3* . . .]
DEPENDING ON *data-name*

Option 1

MOVE {*data-name-1* / *literal*} TO *data-name-2* [*data-name-3* . . .]

Option 2

MOVE CORRESPONDING *data-name-1* TO *data-name-2* [*data-name-3* . . .]

MULTIPLY {*data-name-1* / *literal-1*} BY {*data-name-2* / *literal-2*}
[GIVING *data-name-3*] [ROUNDED]
[ON SIZE ERROR *any imperative statement*]

NOTE *any comment.*

OPEN [INPUT *file-name-1* [*file-name-2* . . .]] [OUTPUT *file-name-3*
[*file-name-4* . . .]]

Option 1

PERFORM *procedure-name-1* [THRU *procedure-name-2*]

Option 2

PERFORM *procedure-name-1* [THRU *procedure-name-2*]

{ *integer-1* / *data-name-1* } TIME[S]

Option 3

PERFORM *procedure-name-1* [THRU *procedure-name-2*]

UNTIL *condition-1*

Option 4

PERFORM *procedure-name-1* [THRU *procedure-name-2*]

VARYING *data-name-1* FROM { *numeric-literal-1* / *data-name-2* }

BY { *numeric-literal-2* / *data-name-3* } UNTIL *condition-1*

Option 5

PERFORM *procedure-name-1* [THRU *procedure-name-2*]

VARYING *subscript-name-1* FROM { *integer-1* / *data-name-1* } BY { *integer-2* / *data-name-2* }

UNTIL *condition-1* [AFTER *subscript-name-2* FROM { *integer-3* / *data-name-3* }

BY { *integer-4* / *data-name-4* } UNTIL *condition-2*] [AFTER *subscript-name-3*

FROM { *integer-5* / *data-name-5* } BY { *integer-6* / *data-name-6* } UNTIL *condition-3*]

READ *file-name* RECORD [INTO *area-name*]

[AT END *any imperative statement*]

STOP { *literal* / RUN }

```
SUBTRACT {data-name-1} [{data-name-2} ...]
         {literal-1  }  {literal-2  }

    FROM {data-name-n} [GIVING data-name-m]
         {literal-n  }

    [ROUNDED] [ON SIZE ERROR any imperative statement]
```

```
WRITE record-name [FROM area-name]
```

Data Division Entry Formats

The complete general forms of the File Description Entry and the Record Description Entry are given below for reference purposes.

The Complete File Description Entry

Option 1

```
FD file-name COPY library-name.
```

Option 2

```
FD file-name [RECORDING MODE IS mode]

    [BLOCK CONTAINS integer-1 {RECORD[S]   }]
                              {CHARACTER[S]}

    [RECORD CONTAINS [integer-2 TO] integer-3 CHARACTER[S]]

    LABEL RECORD[S] {ARE} {STANDARD}
                    {IS } {OMITTED }

    [VALUE OF data-name-1 IS literal [data-name-2 IS ...]]

    DATA RECORD[S] {ARE} data-name-3 [data-name-4 ...].
                   {IS }
```

The Complete Record Description Entry

Option 1

level-number data-name-1 [REDEFINES *data-name-2*]

COPY *data-name-3* [FROM LIBRARY] .

Option 2

level-number { FILLER / *data-name-1* } [REDEFINES *data-name-2*]

[SIZE IS *integer-1* [{ CHARACTER[S] / DIGIT[S] }]]

[USAGE IS { COMPUTATIONAL / DISPLAY }] [OCCURS *integer-2* TIME[S]]

[SIGNED] [SYNCHRONIZED { LEFT / RIGHT }] [POINT LOCATION

IS { LEFT / RIGHT } *integer-3* PLACE[S]] [CLASS IS { ALPHABETIC / NUMERIC / ALPHANUMERIC / AN }]

[PICTURE IS *any allowable combination of characters and symbols as described in Chapter 6*]

[JUSTIFIED { LEFT / RIGHT }] [{ ZERO SUPPRESS / CHECK PROTECT / FLOAT DOLLAR SIGN }

[LEAVING *integer-4* PLACE[S]]] [BLANK WHEN ZERO]

[VALUE IS *literal*] .

List of COBOL Words

Following is a list of words which have pre-assigned meanings in the COBOL language, including both optional and key words. Certain words are key words in one context but are optional in other contexts. However, no COBOL word should be used in any context other than has been prescribed for it in this manual. For this reason, no distinction is made in this list between optional and key words. Certain additional words will be assigned for reference in the Environment Divison. These words will be specified in the publications covering the various processors and, like the words in the list below, should not be used except in the specified contexts.

It may be noted that it has been a general principle in the COBOL language to allow both the singular and plural forms of words in order to improve readability. In most cases, both forms have been shown in the general formats used in this manual. In several other cases, the availability of the optional forms was considered of minor importance and was not indicated in the formats. In all cases, however, both forms have been allowed for in the following list:

ACCEPT
ADD
ADDRESS
ADDRESSES
AFTER
ALL
ALPHABETIC
ALPHANUMERIC
ALTER
ALTERNATE
AN
AND
APPLY
ARE
AREA
AREAS
ASSIGN
AT
AUTHOR
AUTHORS

BLANK
BLOCK
BLOCKS
BY

CHARACTER
CHARACTERS
CHECK
CLASS
CLOSE
COBOL
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONSTANT
CONTAIN
CONTAINS
COPY
CORRESPONDING

DATA
DATE-COMPILED
DATE-WRITTEN
DEPENDING
DIGIT
DIGITS
DISPLAY
DIVIDE
DIVISION
DOLLAR

ELSE
END
ENTER
ENVIRONMENT
EQUAL
ERROR
EVERY
EXAMINE
EXIT

FD
FILE
FILE-CONTROL
FILLER
FIRST
FLOAT
FOR
FROM

GIVING
GO
GREATER
HIGH-VALUE
HIGH-VALUES

IDENTIFICATION
IF
IN
INPUT
INPUT-OUTPUT
INSTALLATION
INTO
I-O-CONTROL
IS

JUSTIFIED

LABEL
LEADING
LEAVING
LEFT
LESS
LIBRARY
LOCATION
LOCK
LOW-VALUE
LOW-VALUES

MEMORY
MODE
MOVE
MULTIPLE
MULTIPLY

NEGATIVE
NEXT
NO
NOT
NOTE
NUMERIC

OBJECT-COMPUTER
OBJECT-PROGRAM
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
OTHERWISE
OUTPUT

PERFORM
PICTURE
PLACE
PLACES
POINT
POSITIVE
PROCEDURE
PROCEED
PROGRAM-ID
PROTECT

QUOTE
QUOTES

READ
RECORD
RECORDING
RECORDS
REDEFINES
REEL
REELS
REMARKS
RENAMING
REPLACING
RERUN
RESERVE
REWIND
RIGHT
ROUNDED
RUN

SECTION
SECURITY
SELECT
SENTENCE
SIGN
SIGNED
SIZE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STATUS
STOP
SUBTRACT
SUPPRESS
SYNCHRONIZED

TALLY
TALLYING
THAN
THEN
THROUGH†
THRU
TIME
TIMES
TO

UNTIL
UPON
USAGE

VALUE
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS

†May be used in lieu of THRU.

Appendix B: Sample Problems

In the following pages are presented three short sample problems intended to show some of the ways in which the COBOL system may be used to solve typical commercial problems. The programs have been written on COBOL Program Sheets in accordance with the rules given in Chapter 5. They are complete, except for the Environment Division entries, which are shown in abbreviated form. The manner in which the programs have been written on the Program Sheets varies from problem to problem with respect to such matters as indentation, spacing, and so on. These variations are intentional; they illustrate the relatively free form of the Reference Format.

Problem 1 — A Table of Salaries

The first problem shows how a computer may be directed to compute a set of values to be listed in a table and then to print the table on a printer with 120 character positions. Specifically, a monthly salary of $500 is to be used to compute the corresponding weekly and annual salaries, and the resulting figures are to be printed in columns headed WEEKLY, MONTHLY, and ANNUAL. The monthly salary figure is then to be increased by $10 and the procedure repeated. This process is to continue until the table shows the corresponding figures for all monthly salaries from $500 to $1,000 at increments of $10.

The format of the printed table is specified as follows: The first 46 columns of each line are to remain blank. Six columns are then left for the heading WEEKLY, and for the figures that will be printed below it. Similarly, seven columns are allowed for the heading MONTHLY, and six for the heading ANNUAL. Three columns are to be left blank between columns, and the 49 spaces remaining at the end of the line are also to remain blank.

IBM

COBOL PROGRAM SHEET

PAGE 1 3 | PROGRAM SALARY TABLE | SYSTEM | SHEET 1 OF 3
010 | PROGRAMMER | DATE | IDENT. 73 SALARIES 80

```
SERIAL 4 6  CONT.  A  B
                  8  12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
010   IDENTIFICATION DIVISION.
020
030   PROGRAM-ID. 'SALARIES'.
040
050   ENVIRONMENT DIVISION.
060
070   CONFIGURATION SECTION.
080   SOURCE-COMPUTER.
090   OBJECT-COMPUTER.
100
110   INPUT-OUTPUT SECTION.
120   FILE-CONTROL.
130   I-O-CONTROL.
140
150   DATA DIVISION.
160
170   FILE SECTION.
180   FD SALARY-FILE
190       LABEL RECORDS ARE OMITTED
200       DATA RECORD IS SALARY-RECORD.
210
220   01 SALARY-RECORD SIZE IS 120 ALPHANUMERIC DISPLAY CHARACTERS.
230
```

IBM

COBOL PROGRAM SHEET

PAGE 1 3 | PROGRAM SALARY TABLE | SYSTEM | SHEET 2 OF 3
020 | PROGRAMMER | DATE | IDENT. 73 SALARIES 80

```
SERIAL 4 6  CONT.  A 8   B 12
                  12        16        20        24        28        32        36        40        44        48        52        56        60        64        68        72
010               WORKING-STORAGE SECTION.
020               77  MONTHLY-PAY  PICTURE 9999.
030               01  SALARIES.
040                   02 FILLER    PICTURE A(46)      VALUE SPACE.
050                   02 WEEKLY    PICTURE ZZZ.99.
060                   02 FILLER    PICTURE AAA        VALUE SPACE.
070                   02 MONTHLY   PICTURE ZZZZ.99.
080                   02 FILLER    PICTURE AAA        VALUE SPACE.
090                   02 ANNUAL    PICTURE ZZZZZ.99.
100                   02 FILLER    PICTURE A(47)      VALUE SPACE.
110
120               CONSTANT SECTION.
130                01 HEADING.
140                02 FILLER  SIZE 46 ALPHABETIC CHARACTERS VALUE IS SPACES.
150                02 WEEKLY  SIZE 6  ALPHABETIC          VALUE 'WEEKLY'.
160                02 FILLER  SIZE 3  ALPHABETIC          VALUE SPACES.
170                02 MONTHLY SIZE 7  ALPHABETIC          VALUE 'MONTHLY'.
180                02 FILLER  SIZE 3  ALPHABETIC          VALUE SPACES.
190                02 ANNUAL  SIZE 6  ALPHABETIC          VALUE 'ANNUAL'.
200                02 FILLER  SIZE 49 ALPHABETIC          VALUE SPACES.
210
```

IBM

COBOL PROGRAM SHEET

PAGE 1 3 | PROGRAM SALARY TABLE | SYSTEM | SHEET 3 OF 3
030 | PROGRAMMER | DATE | IDENT. 73 SALARIES 80

```
SERIAL 4 6  CONT.  A 8   B 12
010         PROCEDURE DIVISION.
020
030         START. OPEN OUTPUT SALARY-FILE. WRITE SALARY-RECORD FROM HEADING.
040             PERFORM CALCULATIONS
050             VARYING MONTHLY-PAY
060             FROM 500
070             BY 10
080             UNTIL MONTHLY-PAY IS GREATER THAN 1000.
090             CLOSE SALARY-FILE. STOP RUN.
100         CALCULATIONS.
110             COMPUTE WEEKLY IN SALARIES = 3 * MONTHLY-PAY / 13
120             COMPUTE ANNUAL IN SALARIES = 12 * MONTHLY-PAY
130             MOVE MONTHLY-PAY TO MONTHLY IN SALARIES. WRITE SALARY-RECORD
140             FROM SALARIES.
```

Problem 2 — A File Search

This example illustrates a type of statistical analysis that is often required in marketing research and similar studies. The computer is directed to search a file of data to find all records containing certain specified data; it is then to write each such record in a separate file, simultaneously recording the number of records found and, when finished, displaying this count on a standard display device.

The records to be examined in this case contain the names and physical descriptions of a large number of persons. The records are stored, unblocked, on magnetic tape. Each record consists of 80 characters of information, of which the first 21 are used in this problem. The following information is included:

Character Positions	Item	Codes	Meaning of the Codes
1-15	Name	(none)	
16	Sex	M	Male
		F	Female
17	Age	Y	Less than 20
		M	At least 20, but not over 50
		E	Over 50
18	Height	T	Over six feet
		M	At least five feet, six inches, but not over six feet
		S	Less than five feet, six inches
19	Weight	H	Over 185 pounds
		M	At least 120 pounds, but not over 185 pounds
		L	Less than 120 pounds
20	Eyes	L	Black
		R	Brown
		A	Hazel
		E	Grey
21	Hair	R	Brown
		L	Black
		G	Grey
		D	Bald
22-80		(this information not pertinent to this problem)	

The computer is directed to obtain the records of all persons having the following characteristics:

1. Females under 20 years of age, five feet, six inches, and over in height, from 120 to 185 pounds in weight, with either hazel or brown eyes, and not bald.
2. Males over 50 years of age, over six feet in height, and over 185 pounds in weight.

Each record meeting these requirements is to be written in a separate file. All records found are to be counted, and the count is to be displayed on a standard display device.

IBM

COBOL PROGRAM SHEET

PAGE 1 3 | PROGRAM FILE SEARCH | SYSTEM | SHEET 1 OF 4
010 | PROGRAMMER | DATE | IDENT. 73 SEARCH 80

SERIAL 4 6 | CONT. | A 8 | B 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72

```
010 IDENTIFICATION DIVISION.
020
030 PROGRAM-ID.  A-0120.
040
050 ENVIRONMENT DIVISION.
060
070 CONFIGURATION SECTION.
080 SOURCE-COMPUTER.
090 OBJECT-COMPUTER.
100
110 INPUT-OUTPUT SECTION.
120 FILE-CONTROL.
130 I-O-CONTROL.
140
150 DATA DIVISION.
160
170 FILE SECTION.
180 FD  RESULT-FILE
190     LABEL RECORDS ARE OMITTED DATA RECORD IS RESULT-RECORD.
200 01  RESULT-RECORD PICTURE IS X(80).
210
```

IBM

COBOL PROGRAM SHEET

PAGE 1 3 020 | PROGRAM FILE SEARCH | SYSTEM | SHEET 2 OF 4 | PROGRAMMER | DATE | IDENT. 73 SEARCH 80

```
SERIAL 4 6  CONT.  A 8   B 12
0010        FD     PERSONNEL-FILE
0020               LABEL RECORDS ARE OMITTED
0030               DATA RECORD IS PERSONNEL-RECORD.
0040
0050        01     PERSONNEL-RECORD.
0060        02     NAME                    PICTURE X(15).
0070        02     SEX                     PICTURE X.
0080               88 MALE                 VALUE 'M'.
0090               88 FEMALE               VALUE 'F'.
0100        02     AGE                     PICTURE X.
0110               88 UNDER-20             VALUE 'Y'.
0120               88 20-TO-50             VALUE 'M'.
0130               88 OVER-50              VALUE 'E'.
0140        02     HEIGHT                  PICTURE X.
0150               88 OVER-6               VALUE 'T'.
0160               88 5-AND-A-HALF-TO-6    VALUE 'M'.
0170               88 UNDER-5-AND-A-HALF   VALUE 'S'.
0180        02     WEIGHT                  PICTURE X.
0190               88 OVER-185             VALUE 'H'.
0200               88 185-TO-120           VALUE 'M'.
0210               88 UNDER-120            VALUE 'L'.
0220        02     EYES                    PICTURE X.
0230               88 BLACK                VALUE 'L'.
0240               88 BROWN                VALUE 'R'.
0250               88 HAZEL                VALUE 'A'.
```

IBM

COBOL PROGRAM SHEET

PAGE	PROGRAM	SYSTEM	SHEET	OF
030	FILE SEARCH		3	4
	PROGRAMMER	DATE	IDENT. 73–80	SEARCH

```
SERIAL  A   B
010            88 GREY                      VALUE 'E'.
020        02  HAIR                     PICTURE X.
030            88 BROWN                     VALUE 'R'.
040            88 BLACK                     VALUE 'L'.
050            88 GREY                      VALUE 'G'.
060            88 BALD                      VALUE 'D'.
070        02  FILLER                   PICTURE X(59).
080
090        WORKING-STORAGE SECTION.
100        77 COUNT   PICTURE IS 99999 USAGE IS COMPUTATIONAL.
110
```

IBM

COBOL PROGRAM SHEET

PAGE 1 3 | PROGRAM FILE SEARCH | SYSTEM | SHEET 4 OF 4
040 | PROGRAMMER | DATE | IDENT. 73 SEARCH 80

SERIAL 4 6 | CONT. | A 8 | B 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72

```
010 PROCEDURE DIVISION.
020
030 BEGIN.
040     OPEN INPUT PERSONNEL-FILE OUTPUT RESULT-FILE. MOVE ZERO TO C
050-    OUNT.
060 READ-CARDS. READ PERSONNEL-FILE AT END GO TO FINISH.
070     IF (FEMALE
080     AND UNDER-20 IN AGE
090     AND NOT UNDER-5-AND-A-HALF IN HEIGHT
100     AND 185-TO-120 IN WEIGHT
110     AND (HAZEL OR BROWN IN EYES)
120     AND NOT BALD)
130     OR (MALE
140     AND OVER-50
150     AND OVER-6 AND OVER-185)
160     THEN ADD 1 TO COUNT WRITE RESULT-RECORD FROM PERSONNEL-REC
170-    ORD.
180     GO TO READ-CARDS.
190 FINISH.
200     DISPLAY 'COUNT IS ', COUNT.
210     CLOSE PERSONNEL-FILE AND RESULT-FILE.
220     STOP RUN.
```

Problem 3—A Work Card Study

This problem illustrates a use of the subscripting principle. In this case, the computer is directed to read a series of records showing the hours worked by employees and then to compute the daily average for each employee. The records are stored, unblocked, on magnetic tape. Each record contains 80 character positions, of which the first 48 are pertinent to this problem. The information in these positions includes a statement of the number of hours worked by an employee on each of five working days in a week. The times are shown decimally as hours and hundredths of hours. The character positions in each record are used as follows:

Character Positions	Item
1-15	NAME
16-20	DEPT
21-24	MON-TIME
25-28	TUES-TIME
29-32	WED-TIME
33-36	THUR-TIME
37-40	FRI-TIME
41-44	TOTAL-TIME
45-48	AVERAGE-TIME
49-80	FILLER

The computer is directed to compute TOTAL-TIME by summing the times for Monday through Friday. As shown in the Data Division for this program, these five times are grouped under the data-name TIME, and the PICTURE associated with it is repeated five times, using an OCCURS clause. In the actual processing, a subscript, represented by the data-name N, is used to obtain each time value in turn. When the five values have been totaled, the computer is directed to divide them by five to obtain the average, and it is then instructed to create a new file consisting of the same records as updated with total and average times.

IBM

COBOL PROGRAM SHEET

PAGE 1 3 | PROGRAM WORK-CARD-PROGRAM | SYSTEM | SHEET 1 OF 2
010 | PROGRAMMER | DATE | IDENT. 73 12345600 80

SERIAL 4 6	CONT.	A 8 / B 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72
010		IDENTIFICATION DIVISION.
020		PROGRAM-ID. 12345600.
030		ENVIRONMENT DIVISION.
040		CONFIGURATION SECTION.
050		SOURCE-COMPUTER.
060		OBJECT-COMPUTER.
070		INPUT-OUTPUT SECTION.
080		FILE-CONTROL.
090		I-O-CONTROL.
100		DATA DIVISION.
110		FILE SECTION.
120		FD WORK-FILE
130		LABEL RECORDS ARE OMITTED
140		DATA RECORD IS WORK-CARD.
150		01 WORK-CARD
160		02 NAME PICTURE X(15).
170		02 DEPT PICTURE X(5).
180		02 TIME PICTURE 99V99 OCCURS 5 TIMES.
190		02 TOTAL-TIME PICTURE 99V99.
200		02 AVERAGE-TIME PICTURE 99V99.
210		02 FILLER PICTURE X(32).
220		FD NEW-WORK-FILE LABEL RECORDS ARE OMITTED DATA RECORD IS NEW-
230	-	WORK-CARD.
240		01 NEW-WORK-CARD PICTURE IS X(80).
250		WORKING-STORAGE SECTION.

IBM

COBOL PROGRAM SHEET

PAGE	PROGRAM	SYSTEM	SHEET	OF
020	WORK-CARD-PROGRAM		2	2
	PROGRAMMER	DATE	IDENT. 73–80	12345600

```
SERIAL CONT. A   B
4    6       8   12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72

010   77  N SIZE IS 1 COMPUTATIONAL NUMERIC DIGIT.
020   PROCEDURE DIVISION.
030   1. OPEN INPUT WORK-FILE OUTPUT NEW-WORK-FILE.
040   2. READ WORK-FILE RECORD AT END GO TO 3.
050          MOVE ZEROS TO TOTAL-TIME  PERFORM 4 VARYING N FROM 1 BY 1 UN
060 -        TIL N GREATER 5 DIVIDE 5 INTO TOTAL-TIME GIVING AVERAGE-TIME
070          ROUNDED  WRITE NEW-WORK-CARD FROM WORK-CARD GO TO 2.
080   3.     CLOSE WORK-FILE AND NEW-WORK-FILE                      STOP RUN.
090   4.     ADD TIME (N) TO TOTAL-TIME.
```

Appendix C: Glossary

Many of the important terms used in this manual are defined in the following pages. In studying them, the reader should recognize that the field of electronic data processing has grown so rapidly, and has developed so many special needs, that it has been difficult to agree on standardized terminology. There are many terms that are used widely throughout the field, but they tend to acquire special meanings when used in connection with particular systems. The COBOL system, however, is designed to permit a high degree of programming compatibility among machine systems, and its vocabulary reflects a common usage. Accordingly, the terms given below have been defined in a manner to emphasize those aspects and meanings which refer in particular to the COBOL system and in general to the art of electronic data processing as it is most widely practiced. The definitions are not intended to be comprehensive, and the reader should consult the text of the manual if he wishes further details.

ACTUAL DECIMAL POINT	A decimal point used for "display" purposes. E.g., when a numeric value is listed on a printed report, the decimal point will often appear as an actual printed character. When specified for data to be used within a computer, it requires an actual space in storage. (Cf. ASSUMED DECIMAL POINT.)
ALPHABETIC	With respect to data, consisting of one or more of the letters of the alphabet and/or one or more spaces. As used in the COBOL system, the term does *not* include other non-numeric characters.
ALPHAMERIC	Alphanumeric.
ALPHANUMERIC	With respect to data, consisting of any of the characters of a computer's character set. As used in this manual, the term includes the characters of the COBOL character set and, in addition, any other characters which may be used within a particular machine system through the use of a COBOL processor.
ARITHMETIC EXPRESSION	An expression containing any combination of data-names, numeric literals, and named constants, joined by one or more arithmetic operators in such a way that the expression as a whole can be reduced to a single numeric value. (See the discussion of arithmetic expressions in Chapter 3 and in Appendix A; see also the arithmetic verbs, as described in Chapter 7.)
ASSUMED DECIMAL POINT	The point within a numeric item at which the decimal point is assumed to be located. When a numeric item is to be used within a computer, the location of the assumed decimal point is considered to be at the right unless otherwise specified in the appropriate Record Description entry. It will not occupy an actual space in storage, but it will be used by the computer to align the value properly for calculation.
BLOCK	In the COBOL system, a group of characters or records which may be treated as an entity for movement into and out of a computer.
BLOCKING	The process of combining two or more data records to form a block.
CHARACTER	One of a set of elementary symbols which may be arranged in ordered groups to express information. These symbols may include the decimal digits 0 through 9,

	the letters A through Z, punctuation symbols, special input and output symbols, and any other symbols which may be accepted by a data processing system.
COBOL WORD	One of a group of words having pre-assigned meanings in the COBOL system. A list of COBOL words will be found in Appendix A of this manual.
COLLATING SEQUENCE	A sequence of characters as arranged in the order of their relative precedence. The collating sequence of a particular computer is determined as part of its design; each character acceptable to the computer has a pre-assigned place in this sequence. A collating sequence is used primarily in comparing operations.
CONDITION	In the COBOL system: 1. One of a set of specified values that a data item can assume. 2. The status of a switch as specified in the SPECIAL-NAMES paragraph of the Environment Division. 3. A simple conditional expression. (See CONDITIONAL EXPRESSION.)
CONDITIONAL EXPRESSION	In the COBOL language, an expression which has the particular characteristic that, taken as a whole, it may be either true or false, in accordance with the rules given in Chapter 3 of this manual.
CONDITION-NAME	A name assigned by the programmer to a value representing one of several conditions which may be assumed by a data item, in accordance with the rules given in Chapter 3 of this manual.
CONSTANT	A value which is to be used in a program without alteration, in accordance with the rules given in Chapters 3 and 6 of this manual. (Cf. FIGURATIVE CONSTANT, LITERAL.)
CONSTANT AREA	A location within the computer in which one or more constants may be stored for reference.
DATA DESCRIPTION	The entry or entries in the Data Division used to describe the characteristics of a data item, in accordance with the rules given in Chapters 4 and 6 of this manual.
DATA DIVISION	That division of a COBOL program which consists of entries used to define the nature and characteristics of the data to be processed by the object program.
DATA ITEM	In the COBOL system, a unit of recorded information which may be identified by a name or by a combination of names and subscripts.
DATA-NAME	A name assigned by the programmer to a data item for use in a COBOL program, in accordance with the rules given in Chapters 3 and 6 of this manual.
DATA RECORD	A record containing data to be processed by a program.
DIGIT	One of the numerals from 0 through 9. In the COBOL system, the term is not used with reference to any other symbols.
DISPLAY	The representation of a data item in visible form, as on a printed report, or in lights or indicators on a machine console or other device. Also, a mode of recording data for ultimate display. (See the discussion of USAGE in Chapter 6.)
EDITING	The process of arranging data for such purposes as improving readability or protecting it against unauthorized alteration; it involves an alteration of format and/or punctuation, together with the suppression of certain characters and/or the addition of others. (See the discussion of PICTURE in Chapter 6.)

ELEMENTARY ITEM	In the COBOL system, a data item containing no subordinate items.
ENVIRONMENT DIVISION	That division of a COBOL program in which the programmer specifies the equipment and equipment features to be used in a program, such as the nature of the input/output equipment, the size and nature of the storage area available, and so on. This subject is further discussed in the publications covering the COBOL processors for the various machine systems.
FIGURATIVE CONSTANT	One of several constants which have been "pre-named" and "pre-defined" in a COBOL processor so that they can be written in the program without having to be described in the Data Division. A list of figurative constants will be found in Chapter 3 of this manual.
FILE (noun)	A set of related data records and/or label records organized for use in a data processing system.
FORMAT	A predetermined arrangement of the types of characters of which a data item is composed. The format of each item is specified by the programmer in the Data Division of a program.
IDENTIFICATION DIVISION	That portion of a COBOL program in which the programmer provides certain information necessary to identify the source and object programs. (See Chapter 9 of this manual.)
IMPERATIVE STATEMENT	A statement consisting of a verb and its operand(s); also, a series of such statements. A statement expresses a complete unit of procedure.
INTEGER	A whole number; e.g., 26 is an integer, while 26.7 is not.
JUSTIFICATION	1. In printing or listing, the alignment of a margin. 2. In the COBOL system, the alignment of characters with respect to the left or right boundaries of data items, as explained in the discussion of the JUSTIFIED clause in Chapter 6 of this manual. (Cf. SYNCHRONIZATION.)
KEY WORD	In the COBOL language, a word which is essential to the meaning and structure of a COBOL statement. In this manual, key words are indicated in the basic formats of verbs and statements by underscoring. A list of key words will be found in Appendix A.
LABEL RECORD	A record used to identify the contents of a file or reel of magnetic tape.
LEVEL	In the COBOL system, the status of one data item relative to another, showing whether one is to be treated as part of the other or whether they are unrelated, as specified in the rules governing level-numbers in Chapters 4 and 6 of this manual.
LEVEL INDICATOR	In the COBOL system, a symbol or level-number used in a Data Division entry to indicate level. For example, FD is a level indicator. (See the discussion of levels in Chapter 4 of this manual.)
LEVEL-NUMBER	A numeric level indicator.
LITERAL	A character, or group of characters, used in a program to represent the value "literally" expressed. Thus, the literal 7 represents the value 7, whereas SEVEN is a *name* that could be used to represent the value 7. (See the rules governing literals in Chapter 3 of this manual.)

LOCK (verb)	To terminate the processing of a magnetic tape in such a way that its contents are no longer accessible, as explained in the discussion of the CLOSE verb in Chapter 7 of this manual.
LOOP	A sequence of procedures intended to be repeated under program control, usually with some modification of at least one of the procedures and/or of the data being operated upon.
MACHINE LANGUAGE	The system of codes by which instructions and data are represented internally within a particular data processing system.
MACHINE WORD	(See WORD.)
MACHINE-INDEPENDENT	An adjective used to indicate that a procedure or a program is conceived, organized, or oriented without specific reference to the operating characteristics of any one data processing system. Use of this adjective usually implies that the procedure or program is oriented or organized in terms of the logical nature of the problem, rather than in terms of the characteristics of the machine used in solving it.
MEMORY	Main storage. (See STORAGE.)
MODE	(See RECORDING MODE.)
NON-NUMERIC	In the COBOL system, not having a numeric value. In this manual, the term is considered equivalent to ALPHANUMERIC. (Cf. NUMERIC.)
NUMERIC	In the COBOL system, having a numeric value. In this manual, the term is used to refer to a value, rather than to the characters used to represent it.
OBJECT PROGRAM	In the COBOL system, a program in machine language resulting from the translation of a source program by a processor.
OBJECT TIME	The time at which an object program is executed, as opposed to the time at which a source program is translated into machine language to create an object program.
OFF-LINE	Not under direct computer control; the term generally refers to the operation of input/output devices.
ON-LINE	Under the direct control of a computer program; the term generally refers to the operation of input/output devices.
OPERAND	In the COBOL language, the "object" of a verb or an operator—i.e., the data or equipment governed, or operated on, by a verb or an operator.
OPERATOR	In the COBOL system, a word or symbol, other than a verb, which directs the data processing system to take some action; e.g., the arithmetic operator + instructs the system to perform an addition, and the conditional operator IF directs it to test a conditional expression.
OUTPUT AREA	A portion of storage in which processed data is assembled in proper sequence and form for release as output from a program.
PROCEDURE	In the COBOL system, a set of one or more statements which direct the computer to perform some operation. A routine.

PROCEDURE DIVISION	That portion of a COBOL program which consists of statements directing the data processing system to take specified actions at object time.
PROCESS TIME	The time at which a source program is translated into an object program through the action of a processor.
PROCESSOR	A specialized program used to translate a source program into an object program.
PROCESSOR VERBS	Verbs which specify to the processor the procedures by which a source program is to be translated into an object program. Such verbs do not cause action at object time. (Cf. PROGRAM VERBS.)
PROCESSOR-DIRECTING VERBS	Processor verbs.
PROGRAM	A complete set of instructions directing a computer to perform a data processing task. The term implies an extended sequence incorporating all of the detailed steps and procedures required to complete a job. (See also OBJECT PROGRAM and SOURCE PROGRAM.)
PROGRAM VERBS	Verbs which cause the processor to generate machine instructions which will be executed by the object program. (Cf. PROCESSOR VERBS.)
QUALIFICATION	With reference to COBOL names, the technique of modifying a name by the addition of another name in order to make it unique, in accordance with the rules given in Chapter 3 of this manual. The name to be qualified is followed by either the word OF or the word IN and then by the qualifying name.
RECORD	A set of one or more related data items grouped for handling by an input/output system. (Cf. DATA RECORD, FILE, LABEL RECORD.)
RECORDING MODE	In the COBOL system, the representation in external media of data associated with a data processing system.
REDEFINE	In the COBOL system, to use the same portion of storage for different data items at different times during the running of a program.
RELATIONAL EXPRESSION	In the COBOL language, an expression that describes a relationship between two terms. For example, A IS LESS THAN B.
RERUN	A repetition of all or part of a program, either from the beginning or from some designated reference point. Rerun procedures are often included in a program so that, in case of an interruption, the program can be restarted from the nearest preceding reference point, thus avoiding reprocessing of the entire program.
ROUND	In the COBOL system, to shorten a number, increasing the least significant remaining digit by 1 when the most significant digit of the part removed is greater than or equal to 5. (Cf. TRUNCATION.)
ROUTINE	A set of one or more statements used in a program to cause a computer to perform some operation or series of related operations.
SECTION	In the COBOL system, a sequence of one or more paragraphs defined in accordance with the rules given in Chapter 3. Also, one of the portions of the program defined

	as a section in the rules governing the format of a COBOL program; e.g., the File Section and the Constant Section of the Data Division.
SENTENCE	In the COBOL language, a complete sequence consisting of one or more statements specifying one or more operations, in accordance with the rules given in Chapter 3 of this manual. A sentence must be terminated by a period.
SOURCE LANGUAGE	As used in this manual, the COBOL language, or some other language made available by means of the ENTER verb, as explained in Chapter 7.
SOURCE PROGRAM	As used in this manual, a program written in the source language. (Cf. SOURCE LANGUAGE.)
STATEMENT	In the COBOL language, a group of words (including symbols where appropriate) which expresses a command, in accordance with the rules given in Chapter 3 of this manual; it may also include a condition to be tested. A statement consists of one or more verbs and their associated operands.
STORAGE	A medium in which data may be retained. Storage may be internal or external. *Main storage*—the principal internal area in which data and program instructions are retained for active use within a data processing system. *Auxiliary storage*—a supplementary storage medium, less active in use than main storage, in which data may be retained; data in auxiliary storage can be used directly by the system, but access is generally slower than to main storage.
STORED PROGRAM	A data processing program which is stored internally within a data processing system. The program itself occupies storage in the same manner as the data used in the program and can be treated as if it were such data.
SUBSCRIPT	An integer used to identify a particular item in a list or table, in accordance with the rules specified in Chapter 4 of this manual. It may be written in a COBOL program as a numeric literal or a data-name.
SWITCH	1. A point in a program from which a program may proceed to one of several possible courses of action, depending on conditions established by the programmer; conditional statements are often used to establish switches of this kind; a branch point. 2. A mechanical, electromechanical, or electronic device, built into a unit of equipment, which can be interrogated in order to select a course of action.
SYNCHRONIZATION	In the COBOL system, the alignment of data with respect to the left or right boundaries of machine words, as explained in the SYNCHRONIZED clause in Chapter 6 of this manual. (Cf. JUSTIFICATION.)
TAPE ALTERNATION	A selection, usually controlled automatically by a program, of first one tape unit and then another, normally during input or output operations, which permits successive reels of a file to be mounted and removed without interrupting the program.
TRUNCATION	The process of dropping one or more digits of a number, either at the left or the right, without altering any of the remaining digits. For example, in most operations the number 3847.39 would become 3847.3 when truncated one place at the right, while it would become 3847.4 when rounded correspondingly. (Cf. ROUND.)
UNBLOCKING	The process of separating and obtaining one or more records from a block. (See BLOCKING.)

VALUE	In the COBOL system, the information represented by a data item, arithmetic expression, or conditional expression.
VARIABLE	In the COBOL system, a named data item in storage which may assume different values at different times during the running of the object program.
VERB	In the COBOL language, one of a selected list of words that specify one or more operations to be performed by a data processing system. (See Chapter 7 of this manual.)
WORD	In the COBOL language, a basic unit of language, serving the same general purposes as words in other languages. *Machine word*—a subdivision of storage having a fixed size.
WORK AREA	A portion of storage in which a data item may be processed or temporarily stored. The term often refers to a place in storage used to retain intermediate results of calculation, especially those results which will not appear directly as output from the program.

Index

Printed in U.S.A. F28-8053-2

IBM®
International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N. Y. 10601