BERNARD A. GALLER

Associate Professor of Mathematics and
Research Associate, Computing Center
The University of Michigan

THE LANGUAGE OF COMPUTERS

McGRAW-HILL BOOK COMPANY, INC.

New York San Francisco Toronto London

THE LANGUAGE OF COMPUTERS

 Library of Congress Catalog Card Number 62-13811

II

22730

PREFACE

This book was written for the person who is interested in learning how problems are solved on electronic computers. In order to solve any problem, a *method of solution* must be found. But if a computer is involved, the method must be made very explicit, and it must somehow be communicated to the computer. We are concerned here with the entire process of solving problems. In every case we must ask the same questions: What is the method by which this problem is to be solved? How can we make this method explicit, i.e., how should it be expressed as a set of rules? What sort of *language* can we use to let the computer know what the problem is and what our method of solution is?

We shall study a series of problems in this book. These problems have been chosen from many different areas, such as the simple, everyday computation of making change, the decoding of secret messages, and the solution of simultaneous equations. For each problem we shall devise a method of solution, and we shall make this method quite explicit by means of *flow diagrams*, which indicate the step-by-step solution. From the study of such problems we shall construct the kind of language which we need in order to express each method

of solution as a set of instructions (i.e., a *program*) to a computer. By the time we have finished, our language will be complete enough to allow us to solve even complicated problems.

Our goal, therefore, is twofold: We are interested in the discovery of *algorithms*, or methods of solution of problems; and we are interested in the way a language is designed for the communication of algorithms to computers.

As high schools and universities recognize the relevance of computers and computer-oriented mathematics to their curricula, courses are being organized to introduce students to machines. These courses are usually offered to high school juniors and seniors or university freshmen and sophomores, and it is for these students that this book has been planned. The basic ideas are not difficult. Most of the difficulties that one encounters in using computers arise in the analysis of the problems to be solved. If the problem is statistical, then one needs a good command of statistics. If the problem concerns the translation of Russian into English, then one needs a good background in linguistics and natural languages. But the underlying computer concepts may be studied quite independently of these specialized applications of the computer. The problems we discuss here involve these basic computer concepts without requiring an extensive mathematical background.

The rules for the language which evolves in this book are summarized in Appendix A. Each rule or definition or new kind of statement is discovered because something is needed to express the solution of a problem. Thus, in the first four chapters, while discussing the change problem and the computation of the social security tax, we are led to the definitions of arithmetic and Boolean (i.e., logical) expressions, the arithmetic substitution statement, and the simple conditional and iteration statements. If necessary, these four chapters could be used alone as a quick introduction to the writing of simple computer programs.

The next four chapters cover the decoding of secret messages, simple numerical integration via Monte Carlo methods, several sorting methods and the binary search procedure, and an elementary discussion of the correlation coefficient. While considering these problems, such language concepts as a more general iteration statement, internal and external functions (sometimes called *subroutines*), and

input-output statements are encountered. Again, if necessary, a short course could easily stop at this point.

In Chapters 9 and 10 we take on somewhat more difficult problems. The first involves a program which must itself *write programs.* The automatic generation of programs is a fairly recent development in the computer field, and one which will definitely become more important in the future. Chapter 10 deals with the solution of simultaneous equations, covering all the subtle cases (such as the system of equations with an infinite number of solutions or no solution at all), as well as the general case.

The book might have ended with Chapter 10, but the question inevitably arises, "Is this language which we are developing *really* the language used on computers?" This question, and the related question, "Are there any other languages for computers?" are discussed in Chapters 11 and 12. Readers familiar with computers may recognize the MAD (Michigan Algorithm Decoder) language in our statements. What about other languages, such as FORTRAN and ALGOL? For that matter, what about NELIAC, JOVIAL, IT, GAT, and FLOWMATIC? These languages are similar in many ways, as we point out in Chapter 12, and any one of them might have served as a basis for this book. The particular language (MAD) used here is easy to describe, easy to motivate, easy to use, and it is available on several computers. Those who have access to a computer may wish to introduce some simple input-output procedures before reaching Chapter 8 in order to run examples on the computer. For those who have access to a computer for which FORTRAN is available, but not MAD, Appendix B provides a set of rules which allow translation of programs from the language of this book into FORTRAN. Appendix C provides similar rules for translation to ALGOL. It certainly is not necessary to have access to a computer to be able to use this book, however. There are many exercises in the book, and none requires a computer.

Another question which is sometimes asked is, "If I learn something about this particular language, will I be able to use computer X which uses language Y?" It is well known among computer users that the use of any computer or computer language makes the next one much easier to learn. The fundamental ideas concerning algorithms, loops, the making of decisions, the structure of arithmetic

expressions, and so on, are common to every computer and every language. The answer to the question, then, is most definitely: "Yes, you will be able to use language Y on computer X with a minimum of attention to the new details of that computer and language."

"But what about the hardware? What are magnetic tapes? How are index registers used?" Quite a few excellent books have already been written about the circuitry and the hardware components of computers. This book is concerned with the *procedure-oriented language* and its use in solving problems. After reading this book, the reader may very well wish to find more details on the computer hardware. Two excellent sources of such information are: "Digital Computer Primer," by E. M. McCormick, McGraw-Hill Book Company, Inc., New York, 1959, and "Digital Computer Programming," by D. D. McCracken, John Wiley & Sons, Inc., New York, 1957.

I would like to express my gratitude for the support of the project on the Use of Computers in Engineering Education at the University of Michigan sponsored by the Ford Foundation. I would also like to acknowledge the many suggestions and criticisms offered by my colleagues and friends.

Bernard A. Galler

CONTENTS

INTRODUCTION

THIS BOOK is intended as an introduction to the language of digital computers. We shall not be concerned here with the hardware, such as magnetic drums, high-speed printers, and so on, but with the *software*. This refers to the *language* by means of which we communicate with the computer. By studying typical problems which might be posed to a computer, we shall generate a description of a computer language. When we finish, this language will be complete enough to enable us to describe a great many of our problems to the computer, and it will also be quite natural for everyone to use.

More important than the development of a suitable language, however, is the insight we shall obtain into the structure of many of the problems which we bring to the computer. We shall see that the ability to make decisions is a basic ingredient in the solution of every problem, and we shall need a way to describe the decision to be made as well as the courses of action which are possible as a result of that decision. Another common ingredient is the *loop*, which requires a sequence of steps to be performed over and over until an appropriate decision is reached as to the effectiveness of the procedure or the number of repetitions made.

This view of the structure of problems and the computational procedures leading to their solution does not depend at all on any

particular computer, nor does it depend on a particular language. We shall find it convenient, however, to develop a suitable language for describing procedures. Later, we shall discuss the relationship between this language and other similar languages.

The ideas that we shall deal with are not hard to understand. Many high school students have already been introduced to computers, and they are quite capable of understanding the basic concepts. Why, then, is there the strong emphasis that is always placed on mathematics in any discussion of computers? It happens that most of the problems that have been successfully attacked by means of computers have had mathematical formulations. Mathematicians have been interested in computational procedures for hundreds of years, and it is quite natural that the first problems which were considered feasible for machine computation were mathematical methods already studied and developed in the form of *algorithms*, i.e., sequences of well-defined steps leading to the solution.

In recent years, however, we have become more and more aware of the power of the computer to solve nonmathematical problems, such as the simulation, in a few minutes, of the behavior of a factory over a period of five years or the translation of articles and books from one language to another. Many of the problems we shall consider here will be nonmathematical. In fact, since the main ideas will be quite independent of formal mathematics, none of the examples will demand more than a bare minimum of mathematics, such as the solution of quadratic equations or the law of cosines.

It was indicated that we shall generate a description of a computer language. The reader should understand that the final version of the language toward which we shall aim does in fact exist and is called MAD (the Michigan Algorithm Decoder). It was developed at the University of Michigan by Bruce W. Arden, Robert M. Graham, and the author, and has been in use by students at the University since February, 1960. Nothing that follows, therefore, is hypothetical or fictitious.

CHAPTER ONE

THE CHANGE PROBLEM

AN INTERESTING PROBLEM which all of us solve every day, almost without realizing it, is the "change problem." What happens when you hand the grocer a dollar bill in payment for something which costs 21 cents? He gives you 79 cents in change, of course, but which coins does he use? Actually, there are many different ways to make up 79 cents, such as seven dimes and nine pennies, or five dimes, five nickels, and four pennies, and so on. Suppose we make the problem more specific, then, and ask him to use the *fewest* coins in making change, which is what most grocers do, anyway.

One way, used by many people, is this: Start with the amount being charged (21 cents). Add enough pennies to just come to a multiple of 5. (In this case we use four pennies.) Now build it up to a multiple of 25 by adding two dimes, a dime and a nickel, one dime, one nickel, or nothing at all. (Since we are already at 25 we add nothing.) Now add enough quarters to take the amount to a multiple of 50. Follow this with enough half-dollars to take it up to a dollar. This gives us, in our particular example, four pennies, a quarter, and a half-dollar, which is the best way to do it.

What would happen in the very special case in which the cost of the article we are buying is zero? (This is sometimes called "getting

change for a dollar.") The same rule still works, giving just two half-dollars, which is not the usual way we change a dollar, but which does use the smallest number of coins. In applying the rule in this case, remember that we are starting with zero, and zero is a multiple of any number.[1] Thus, we need to add *no* pennies to make zero a multiple of 5, and so on. It usually pays to test any rule against some very special cases, since these cases actually show up errors on occasion. What would happen here, for example, if the article costs exactly one dollar, so that one should get no change?

To show that the same problem can have more than one rule, or *algorithm*, for its solution, we shall state another rule for the change problem: Subtract the cost of the article from one dollar (giving the amount of change to be made, e.g., 79 cents in the above example). Divide this amount by 50. The quotient indicates how many half-dollars are needed, and the remainder is the amount still to be accounted for. (We find that we need one half-dollar, and we still need to account for 29 cents.) Divide in the same way by 25, so that the quotient indicates how many quarters are needed, and so on. We divide in turn by 50, 25, 10, 5, and 1, each time noting the quotient, and using the remainder for the next division. It is easy to see that this also leads us to a half-dollar, a quarter, and four pennies for the 21-cent article and two half-dollars as change for the dollar.

The second rule is easier to write, but not as easily carried out without writing down some of the numbers. This is probably the reason most people do not use it in their grocery change. A computer, however, can easily save numbers for future reference, since it has a *storage* section (sometimes called its *memory*). Information, such as numbers or strings of letters, may be stored in the storage section of the computer and may later be recalled as often as necessary. The second method would therefore be quite feasible for a computer.

Let us look at this method a little more closely. We note first that if A is the amount of change we must give and q_{50} is the quotient we get when we divide A by 50, then the remainder is $R_{50} = A - 50 \cdot q_{50}$, so that in our earlier example $A = 79$, $q_{50} = 1$, and

[1] Take any number w, and you have $0 \cdot w = 0$; so zero can be written as something times w, making it a multiple of w.

$R_{50} = 79 - 50 \cdot 1 = 29$. A similar statement holds for division by 25, 10, and so on. In order to discuss this more easily, we shall find it convenient to introduce here the *greatest-integer function.* When we write [B] for some number B, we shall mean B itself if B is an integer, and if B is not an integer we shall mean the greatest integer less than B. Thus, $[6] = 6$, $[0] = 0$, $[-3] = -3$, $[2.4] = 2$, and $[-3.2] = -4$.[1] The reason for using this function here is that we can now write

$$\begin{aligned} q_{50} &= [A/50] & R_{50} &= A - 50 \cdot q_{50} \\ q_{25} &= [R_{50}/25] & R_{25} &= R_{50} - 25 \cdot q_{25} \\ q_{10} &= [R_{25}/10] & R_{10} &= R_{25} - 10 \cdot q_{10} \\ q_5 &= [R_{10}/5] & R_5 &= R_{10} - 5 \cdot q_5 \\ q_1 &= [R_5/1] & R_1 &= R_5 - 1 \cdot q_1 \end{aligned}$$

Of course, $q_1 = R_5$ and $R_1 = 0$, but it is interesting to see that they do fit into the same general pattern. In fact, the pattern itself can be used to simplify the description of the rule. In each step there is a divisor, which is the same number used afterward as a multiplier. Also, after we use R_{50} to find q_{25} and R_{25}, we really do not need it anymore. We might as well just remember q_{25} and R_{25} and forget R_{50}. In fact, as soon as we have used any one of the R's to compute the next q and R values, we no longer need it. Computer people are usually on the lookout for such situations, since they are not too anxious to waste part of the storage section of the computer remembering numbers that are no longer needed. (It does not hurt until one runs out of storage, but one might as well develop good habits.) Let us then reserve one place for the R's, called R (without a subscript), and as soon as we compute any new R value, we will put it there, wiping out the previous value. Of course, we will still remember all the q's separately, since we need all of them as the answer to the problem. The second line of the computation outlined above would now appear as follows:

$$q_{25} = [R/25] \qquad R = R - 25 \cdot q_{25}$$

[1] A *function* is a rule by means of which one assigns to each number a unique second number. Thus, the function x^2 assigns to each number chosen as a value of x the square of that number. This function may be represented as a set of pairs (0,0), (1,1), (2,4), (−2,4), etc., where the first number is a value chosen for x, and the second value is the square of that value. For the function [B] used here, typical pairs are (6,6), (−3,−3), (2.4,2), and (−3.2,4).

Note, however, that if we subtract R from both sides of the last equation, we obtain $0 = -25 \cdot q_{25}$, so that $q_{25} = 0$ no matter how much change we need, which is impossible. The trouble is that we are now using the equal sign in a new way. We are not asserting that $R = R - 25 \cdot q_{25}$ and asking for which value of q_{25} this is true, as in ordinary algebra. We are instead saying: Compute the right side, i.e., $R - 25 \cdot q_{25}$, using the current value of R, and put the result in the place whose name is given on the left, e.g., R. This new interpretation of the equal sign is sometimes described by saying that we are using a *command language*, rather than a *descriptive language*. Sometimes an arrow ($\leftarrow$) is used instead of the equal sign, so that the second line of the computation would be written:

$$q_{25} \leftarrow [R/25] \qquad R \leftarrow R - 25 \cdot q_{25}$$

In diagrams, such as Figure 1.1 below, we shall use the arrow in this way, but in the text we shall continue to use the equal sign with this new meaning, since the devices used for input to most computers do not recognize the arrow. The above rule for computing change can now be written in the following way:

$$\begin{aligned} q_{50} &= [A/50] & R &= A - 50 \cdot q_{50} \\ q_{25} &= [R/25] & R &= R - 25 \cdot q_{25} \\ q_{10} &= [R/10] & R &= R - 10 \cdot q_{10} \\ q_{5} &= [R/5] & R &= R - 5 \cdot q_{5} \\ q_{1} &= [R/1] & R &= R - 1 \cdot q_{1} \end{aligned}$$

The first line can be made to look exactly like the others if we set R equal to A to begin with. The first line would then be

$$q_{50} = [R/50] \qquad R = R - 50 \cdot q_{50}$$

The other feature of the general pattern we noted above was the divisor which occurs in each step. If we call the divisor d, all five lines of the rule can be written in the same way:

$$q_d = [R/d] \qquad R = R - dq_d$$

Now we need start with only R = A, and let d take on in turn each of the values in the "coin list": {50,25,10,5,1}. The process terminates when R has been reduced to zero, as it always must. In fact,

since we may ask after each step whether R = 0, occasionally we shall be fortunate enough to have the process end early. For example, if A = 0, so that no change is needed, the termination condition is satisfied as soon as we set R = A. There is one slight complication here, however. If the computation ends on the very first step, what will be the values of $q_{50}, q_{25}, \ldots, q_1$? These are, after all, the answers to the problem. Of course, they should all be zero, and in general, whenever the process ends at some early stage, the q's not yet computed should be zero. The easiest way to accomplish this is to set all the q's to zero at the very beginning. Some (or all) of the q's will receive new values as we proceed, but the remaining q's will still be zero if the process should happen to end early.

Figure 1.1 is a *flow diagram* of the algorithm. We shall see that flow diagrams are among the most useful devices we have for communicating algorithms. Remember that q_d is the number of coins we need of denomination d, A is the total amount to be given in change, and R is the amount remaining at each step of the procedure.

The first three boxes (after the word START) perform the *initialization*. There is then a *loop*, which consists of a *termination condition* (in this case R = 0); a block of computation, sometimes called the *scope* or *body* of the loop (here the substitutions $q_d \leftarrow [R/d]$ and $R \leftarrow R - dq_d$); and a box which changes d each time around, called the *modification box*. We shall use the shape of each box to suggest its role in the algorithm. Rectangular boxes will be used for actual computation and diamond-shaped boxes for the decisions. Since decision boxes always contain assertions which are either *true* or *false*, there will always be two arrows leaving each diamond-shaped box, one labeled *true*, the other *false*.

As an example of the actual application of the algorithm as given in the flow diagram, we shall consider the case in which A = 25 (i.e., the article originally cost 75 cents). Initially, one would set $q_{50} = q_{25} = q_{10} = q_5 = q_1 = 0$, R = 25, and $d = 50$, and because $R \neq 0$, one would compute $q_{50} = [\frac{25}{50}] = [.5] = 0$ and $R = 25 - 50 \cdot 0 = 25$. Setting $d = 25$, we examine the termination condition R = 0. Since it is *false*, another cycle of the loop is performed, yielding $q_{25} = [\frac{25}{25}] = 1$ and $R = 25 - 25 \cdot 1 = 0$. Setting $d = 10$, we again examine the termination condition and find it *true;* so the computation is terminated. Since the values of

q_{10}, q_5, and q_1 did not change, they remain zero, and we conclude that one quarter is needed to give 25 cents in change.

Suppose now that we have the task of determining the specifications for the language in which this algorithm could be communicated to a computer. Although it would be desirable to feed Figure 1.1

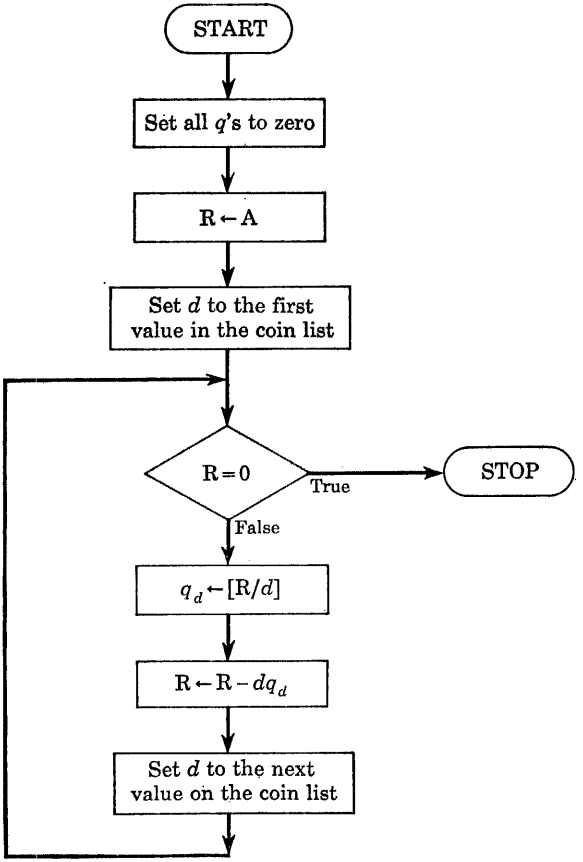


Figure 1.1

into the computer directly, we shall assume that we can feed in only strings of letters and digits. Since we can set the specifications for the language, we must first recognize the need for straight computation. We shall therefore ask for the privilege of writing any

substitution statement we need, that is, any equation in which the equal sign is interpreted as indicated above, e.g., the left side gives the name of the place in storage into which the value of the right side is to be stored. The reason for the name *substitution statement* is that the new value is substituted for the old value in storage.

Note that in the termination condition the equal sign is used in yet a different way. When we wrote R = 0, it was intended to be an expression which could be labeled *true* or *false*, depending on the current value of R. It is *true* for some values of R (in this case only one value) and *false* for others. The equal sign is called a *relation* when it is used in this way. Other relations that are used a great deal are $<$, $\leq$, $>$, $\geq$, and $\neq$, meaning, "less than," "less than or equal," "greater than," "greater than or equal," and "not equal," respectively. Although a simple expression (R = 0) sufficed for the termination condition here, it is not hard to imagine situations arising in other problems in which the termination condition would have to be more complicated, such as

$$M < N \textit{ and } P \neq Q \textit{ or } M \geq N \textit{ and } P < Q + 3$$

This is again an expression which is *true* for some sets of values of M, N, P, and Q and *false* for other sets of values. We will probably want to include in our specifications for the computer language the ability to write very complicated *true-false* expressions for use as termination conditions and, as we shall see, for use in other ways.

We have to be careful, however, in making these specifications. How complicated an expression can the right side of a substitution statement be? How can one determine whether an expression is legitimate or just a meaningless collection of characters, such as Q/T + ((A(.$? We must carefully describe just what shall be considered a legitimate expression in the proposed language. Then we shall be able to build from this the description of our statements, such as the substitution statement, the simple and compound conditional statements that we shall introduce later, and so on.

CHAPTER TWO

EXPRESSIONS

WE MUST NOW take a good look at what we have been calling *expressions* in the preceding chapter. The first thing is to distinguish between the *arithmetic* expression, which has a number as its value, and the *logical* expression, which has *true* or *false* as its value. (Logical expressions are usually called *Boolean* expressions, after the logician George Boole.) We have seen examples of both kinds of expressions in our change problem, e.g., $R - dq_d$ is an arithmetic expression, and $R = 0$ (as used there) is a Boolean expression.

Now, what kinds of things go into making up an arithmetic expression? We have *names of variables*, such as R and q_d; *constants*, such as 50 and 25; and *operations*, such as + and −. Our job, then, is to set forth the rules as to what constitutes a legitimate name for a variable, a legitimate constant, and a legitimate operation. After that we shall decide what combinations of these three ingredients make up acceptable arithmetic expressions.

2.1 NAMES OF VARIABLES

What, then, should be acceptable as the name of a variable (i.e., the symbol used in referring to the variable)? Why is it necessary

to put any restrictions on these symbols at all? It is obviously necessary to have some restrictions; otherwise someone might write A + B, and we could not tell whether he meant the variable represented by the symbol "A + B" or the sum of the two variables A and B. We will probably all agree that the symbol for an operation may not occur within the symbol representing a variable. Similarly, we shall have to rule out parentheses. What about a symbol like "91"? This symbol is commonly understood to represent a constant, i.e., the integer 91, rather than a variable. The rule we choose for describing symbols acceptable as names of *variables* will have to rule out such constants. What about the number of characters in a symbol? Since even the largest computer has a fixed amount of storage, we cannot allow arbitrary lengths for these symbols. We shall therefore set some upper limit (such as six) to the number of characters in each symbol. We could just as well have chosen ten as the upper limit, but we shall use six. Restrictions such as these, which are based on machine considerations, are not particularly desirable, but they greatly facilitate the handling of expressions inside the computer.

Let us agree on a rule, then, for the construction of symbols to represent variables. *A symbol which represents a variable will contain one to six capital letters or digits, the first of which must be a letter.* Examples of acceptable symbols are Q1, BC3A, and R. Examples of unacceptable symbols are 9ED, 1.5, 91, and E1 + 2. We shall also find it convenient to use the following terms in referring to symbols representing variables: the name of the variable, the variable, the variable name. (In using "variable name," there is no implication that the *name* will be changing.)

2.2 CONSTANTS

Without going into a similar discussion with regard to constants, let us simply state the rules for forming acceptable constants. *A numeric constant contains one to eleven digits with or without a decimal point and with or without a sign.* Examples of acceptable numeric constants are 0, −1.0, 1., .0, and 51.246513912.

In Chapter 5 we shall discuss a problem concerned with decoding messages. In such a problem one must be able to ask, for example, whether a particular character is an "A", etc. We shall therefore find it convenient to allow alphabetic constants such as "A". We shall write alphabetic constants in a way which will correspond to the way quotation marks are used in English in writing about the the name "A" of the variable A. *An alphabetic constant contains up to six characters* (*any character except the* $ *itself*), *preceded and followed by dollar signs* (our substitute for quotation marks, since the input devices on most present-day computers do not accept quotation marks). Although we will completely ignore blanks elsewhere, here we will count the blank space as a character, so that examples of alphabetic constants are GO, $GO ON$, $5 + 3$, and $4TE−1$.

2.3 OPERATIONS

What about the operations which can occur in arithmetic expressions? We obviously want to allow +, −, and multiplication and division, and probably exponentiation, i.e., A^B. We cannot omit the multiplication sign as we do in ordinary algebra, however, since then we could not tell if "AB" is a variable name or the product of the variables A and B. (We shall use "*" for multiplication, since the center dot "·" is not acceptable to the input devices on most computers.) Also, since we do not have a way to raise something to the level of an exponent on most computers, we shall invent a symbol for exponentiation. We shall write A .P. B for A^B, where .P. reminds us of "raising to the power." (We cannot just write A P B, since we could not distinguish it from the variable name "APB".) Similarly we shall write .ABS. for *absolute value*.

2.4 ARITHMETIC EXPRESSIONS

We are finally in a position to say what an arithmetic expression is. An arithmetic expression is defined as follows:

1. Numeric and alphabetic constants and variable names are arithmetic expressions.

2. If 𝔄 and 𝔅 are already known to be arithmetic expressions, then so are the combinations 𝔄 + 𝔅, 𝔄 − 𝔅, +𝔄, −𝔄, 𝔄 * 𝔅, 𝔄/𝔅, 𝔄 .P. 𝔅, .ABS. 𝔄, and (𝔄), the last one meaning that any expression may be enclosed in parentheses at any time, if desired.

3. The only arithmetic expressions are those which are generated by (1) and (2).

Note that in this definition, (1) allows us to start with certain very elementary expressions with which we are already familiar. Then (2) allows us to combine the expressions we have from 1 into more complicated expressions. If we apply (2) again to these expressions, we obtain even more complicated expressions, and so on. This process goes on and on, but (3) closes the definition by asserting that there is no other way to obtain an expression. In other words, (1) and (2) provide a way to obtain expressions, and (3) makes this the only way. (We shall see this method of definition again later.)

These rules allow very complicated arithmetic expressions to be formed, such as

$$B .P. (X - Y) + (C - D) .P. (E/F + G)$$

Let us write the same expression without parentheses, however. We then obtain the expression

$$B .P. X - Y + C - D .P. E/F + G$$

If we were to ask several people (who had not seen the original expression) to put in the missing parentheses, we might get the original expression back again, or we might not. We should really ask a more basic question, e.g., what is the reason for having parentheses in an expression? Their job is to indicate the order in which the steps of the computation are to be done. Thus, if we write $6 + (12/2)$ and $(6 + 12)/2$, we obtain two different values, e.g., $6 + 6 = 12$ and $18/2 = 9$. It makes a difference, then, in which order we do things, and parentheses help us understand which particular sequence of computation is intended for each expression.

Why do we ever leave parentheses out at all, then, if we need them to specify the sequence of the computation? Too many parentheses would make any expression very difficult to read. If *all* the parentheses were written for the example above, we would have the

following:

$$(B .P. (X - Y)) + ((C - D) .P. ((E/F) + G))$$

We see that some parentheses were omitted even from the original expression. There is no harm in this, provided that there is some well-known rule, or agreement, which indicates precisely how any missing parentheses are to be inserted. If we were to write $6 + 12/2$, most of us would immediately understand it to mean $6 + (12/2) = 12$. If we had intended it to mean $(6 + 12)/2$, we would be obliged to keep the parentheses there. It is important now that we make explicit the agreement on the order in which operations will be performed when parentheses have been omitted, so that there will be no doubt as to the meaning of any expressions we may encounter.

The reason we decided to do the division first in the expression $6 + 12/2$ is that we have always agreed to do division before addition unless parentheses indicate otherwise. We say that *division takes precedence over addition.* This implies that we have in mind a ranking (i.e., ordering) of the operations, and we always do an operation with a higher rank before an operation with a lower rank. Let us determine what this ranking is. We have already indicated that division ranks higher than addition. Division would rank higher than subtraction as well, and multiplication would also rank higher than addition or subtraction. Unfortunately, we cannot claim that multiplication ranks higher than division, nor that division ranks higher than multiplication. The same holds true for addition and subtraction. For example, if we write $6 - 4 + 2$, we would arrive at a value $(6 - 4) + 2 = 2 + 2 = 4$. From this we might be tempted to say that subtraction, being done first, has a higher rank. However, if we write $6 + 4 - 2$, we would compute $(6 + 4) - 2 = 10 - 2 = 8$, and it would appear that addition has the higher ranking. There is no harm in having some operations (such as these) with the *same* rank, since we will include in our rule a special provision for handling operations of the same rank.

Before stating the rule, however, we must decide how to rank the exponentiation operation .P. and the absolute value operation .ABS., and we should also take into account the fact that a minus in front of an operand (representing *negation*) is quite a different kind of operation from the minus between two operands (representing

subtraction). (An operand is a symbol or expression upon which an operation is performed.) In order to place the operation .P. in our ranking, let us consider a typical expression involving exponentiation, such as A * B .P. C * D, and decide where we would want to put the parentheses if we had to insert them. In ordinary mathematical notation, there would be no need for a decision, since this expression would be written as $A \cdot B^{CD}$, or $A \cdot B^C \cdot D$, or perhaps $(AB)^C D$, or some other way, but there would be no doubt as to the nature of the exponent. We should try to be as close to ordinary mathematical notation as possible, however; so let us note that in evaluating $A \cdot B^C \cdot D$, we would evaluate the factor B^C before doing either of the multiplications. We are thus led to decide that .P. should rank higher than multiplication.

Except for the negation operation and .ABS. mentioned above, we now have the following ranking:

.P.
*, /
+, −

where each operation on a particular line has higher rank than any operation on any lower line. It is important to observe at this point that we could have decided on a different rank for .P. if we had wished. Provided that we were consistent throughout the rest of our work, there would be no lack of communication as a result of such a decision. These rules are simply agreements so that we all do our computations the same way. Since we do not have any compelling reasons for departing from standard mathematical notation, however, we will try to be consistent with it whenever possible, but with the understanding that *we could depart from it if we needed to.*

Now let us rank the absolute value operation .ABS. relative to the other operations. Using the same procedure that we used above, let us examine standard mathematical notation. Unfortunately, we are not helped much here, since absolute value ordinarily indicates very clearly where it fits into the computation by means of beginning and ending vertical lines, such as in the expression $|X + Y \cdot Z| - 4 \cdot X + Y$. This time we must make our own rule without any help from standard mathematical notation, since we do not have a terminating symbol. The question is: Where would

we put parentheses in the expression A * .ABS. B * C? As a general rule, a *unary* operation, i.e., an operation which stands in front of one operand, rather than between two operands, should be applied just to that object in front of which it appears. "That object" might be a variable, a constant, or a compound expression enclosed in parentheses. This would imply that the example above would be interpreted as A * .ABS.(B) * C, and if we wanted $A \cdot |B \cdot C|$, we would write A * .ABS.(B * C). It appears, also, that the unary operation .ABS. should be ranked higher than the *binary* operations +, −, *, /, and .P., which would guarantee that .ABS. is done as soon as possible whenever it occurs. Our ranking now appears as follows:

.ABS.
.P.
*, /
+, −

Continuing in this way, we would argue that the unary minus sign, i.e., negation, should go very high on the list as well, since it is unary and should apply to its immediate successor in the expression. Here we meet a quirk of standard mathematical notation, which will force us to do things slightly differently. Consider the expression −A .P. B. In mathematical notation, we would write it $-A^B$, and normally we would interpret this as $-(A^B)$, rather than $(-A)^B$. We are thus forced to let exponentiation (.P.) rank higher than negation. We therefore end up with the ranking

.ABS.
.P.
− (unary)
*, /
+, − (binary)

All that remains is the rule for handling two operations with the same rank. We may take our clue from the way we evaluated several earlier expressions, such as $6 - 4 + 2 = (6 - 4) + 2 = 2 + 2 = 4$. We simply proceed *from left to right.* Returning now to our original complicated expression without parentheses,

B .P. X − Y + C − D .P. E/F + G

our rules would lead us to insert first the following parentheses:

$$(B .P. X) - Y + C - (D .P. E)/F + G$$

and then

$$((((B .P. X) - Y) + C) - ((D .P. E)/F)) + G$$

This is not the interpretation we wanted when we wrote it the first time with parentheses

$$B .P. (X - Y) + (C - D) .P. (E/F + G)$$

and that is exactly why it was written as it was, with parentheses.

There is one detail that we must consider before leaving arithmetic expressions. We saw in the change problem that it is sometimes very useful to be able to compute [A/B], where A and B are integers. What this really amounts to is the stipulation that we obtain an integer for a quotient when we divide two integers. This suggests that perhaps the arithmetic we do with integers should be different from the arithmetic on nonintegers. (Actually, this arises only in division, since the sum, difference, and product of two integers are always again integers.) Since most computers can do additions and subtractions faster with integers than nonintegers, we usually find it useful, anyway, to spell out which variables have integer values and which do not, and which constants are integers, and so on.

Let us then agree to do integer arithmetic on those variables and constants which are recognized as integers, even to the point of using "truncated division," i.e., A/B means [A/B] if A and B are integers. But how do we recognize integer constants and variables whose values will always be integers in our language? We shall first stipulate that *any numeric constant which contains no decimal point is an integer*. Thus, 5, 0, −3 are integers, and 3.2, 1., −1.0 are not, even though their *values* may in some cases be integral. This gives us the privilege of deciding, by using the decimal point or not, whether a particular number should be considered an integer or not.

In the case of a variable we find that we can no longer tell whether its values will be integers or not merely by looking at it. We shall have to make a special descriptive statement (sometimes called a *declaration*) about those variables whose values are to be considered integers, i.e., those variables which are to be of *integer mode*. We

could write

A IS AN INTEGER, R IS AN INTEGER

and so on for each variable. The clause IS AN INTEGER should not have to be repeated each time, however. We shall write

INTEGER A, R, D, Q50, Q25, Q10, Q5, Q1

and agree that the word INTEGER will apply to the entire list of names.

A statement such as the INTEGER declaration we have just written will not be *executed*, in the sense that a substitution statement is executed. There is no computation resulting from a declaration; it is only descriptive information about the program. We shall, in fact, sometimes refer to such declarations as *nonexecutable statements*. It should be understood that *executable statements*, such as substitution statements, will be executed in the order in which they occur, one statement after another; so it is important to have them in the right order. Since declarations are not executed, they may occur anywhere. During the execution of the other statements, declarations are simply bypassed. Actually, it sometimes happens, as it did here, that *every* variable is to be of integer mode. In this case we shall write

NORMAL MODE IS INTEGER

which will settle it once and for all.

Now for the inevitable complications. What happens if we write B/N, where B has not been declared to be of integer mode, but N has been so declared? The easy way to settle it would be to say that it is illegal to write it at all. A more realistic policy (which we shall adopt) would be to say that for this particular division operation we will treat N as noninteger. Then we get the full quotient, without any harm being done.[1] Also, if we write N = B, where N is of integer mode and B is not, then N must have an integer value, and we shall have to interpret it as N = [B], i.e., the greatest integer

[1] Warning: Many quotients can be represented only by means of an infinite number of digits; e.g., $\frac{7}{3} = 2.33333333\ldots$. Since computers cannot store all these digits, we must stop at some point, e.g., 2.3333333333. This introduces an error into the computation, called *roundoff error*, but a discussion of such errors is beyond the scope of this book.

less than or equal to B. (We could round B, instead of just dropping its fractional part, but it is often more convenient to avoid rounding. Besides, one can always achieve the rounding effect by writing N = B + .5 instead of N = B. Adding the .5 forces a carry into the integer part of B if the fractional part of B is .5 or more. This carry, if it occurs, together with the "greatest-integer" interpretation of the equation N = B + .5, i.e., as N = [B + .5], has the effect of a rounding process. Note that this is not the most common rounding procedure (which rounds to the nearest even integer), since 2.5 rounds to 3 rather than the nearest *even* integer 2. It is one of the easiest rounding procedures to perform, however, and it is therefore the procedure actually built into most computers.)

2.5 BOOLEAN EXPRESSIONS

It was pointed out in Chapter 1 that we would probably want to include in our language the ability to construct quite complicated logical, or Boolean, expressions, i.e., expressions which may only have the values *true* or *false*. The smallest unit in such expressions is the *basic Boolean expression*. A basic Boolean expression consists of one of the relations $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ preceded and followed by any two arithmetic expressions. Examples of basic Boolean expressions are $X + Y \geq 48$, $A < B + C - K$, and $X = Y$. Unfortunately, most present-day computers do not accept or recognize the characters $<$, $\leq$, $>$, $\geq$, and $\neq$. We shall use instead the easily remembered names .L., .LE., .G., .GE., and .NE. for "less than," "less than or equal," "greater than," "greater than or equal," and "not equal," respectively. The one remaining case is the equal sign, which many computers do recognize. Since we are already using the equal sign in the substitution sense, however, we might as well recognize its appearance as a relation as something entirely different. Corresponding to the names given to the other relations, we now shall write .E. for the equal sign *when it is used as a relation* (that is, when "equal to" appears in a Boolean or logical expression). The basic Boolean expressions given as examples above are now written X + Y .GE. 48, A .L. B + C − K, and X .E. Y. Here, for example, X + Y .GE. 48 means that the values of X and Y

will be added, with the result being compared with 48. The expression is *true* if the sum is greater than or equal to 48, *false* otherwise.

Once again we have been omitting parentheses, and now we see that the new relations, such as .GE., must also be included in the rules which were developed in Section 2.4 to indicate the order in which the operations and relations are to be applied in the computation. Fortunately, we need merely place them in the list which indicated the ranking of the operations. The same general rules may then be applied. From the example above,

$$X + Y \text{ .GE. } 48$$

we see that the only meaningful way to insert parentheses is as follows:

$$(X + Y) \text{ .GE. } 48$$

Any other insertion of parentheses, such as

$$X + (Y \text{ .GE. } 48)$$

would lead to a *meaningless expression*, not just a wrong answer.

It should be clear, then, that all the relations .E., .NE., .G., .GE., .L., and .LE. should appear in the ranking *below* all the arithmetic operations. We shall consider below the ranking among the relations, i.e., the ranking of .L. with respect to .G., and so on. It will turn out that we will not need to rank the relations relative to each other at all.

Although basic Boolean expressions are usually enough for us to express the logical conditions which we wish to use in our decisions, we sometimes need to make more complicated decisions. For example, how do we ask whether x is between 4 and 5? The usual algebraic notation is $4 < x < 5$, but this is really a compressed form of the expression

$$4 < x \textit{ and } x < 5$$

We see now that we need to express not only basic Boolean expressions such as $4 < x$, $x < 5$, and so on, but combinations of these using the word *and*, which is sometimes called a *connective*. What happens if we try to say that x is *not* between 4 and 5? It is not hard to see that we would write

$$x \leq 4 \textit{ or } x \geq 5$$

So we need the connective *or*, also. Are there any other connectives which we might need? There are others which are sometimes used, such as *not*, but as long as we can use all the relations $<$, $\leq$, $>$, $\geq$, $=$, and $\neq$, we can get along perfectly well with just *and* and *or*.

We may now give a general definition of a Boolean expression:

1. Basic Boolean expressions and the Boolean constants *true* and *false* are Boolean expressions.

2. If $\mathcal{P}$ and $\mathcal{R}$ are already known to be Boolean expressions, then so are $(\mathcal{P})$, $\mathcal{P}$ .AND. $\mathcal{R}$, and $\mathcal{P}$ .OR. $\mathcal{R}$.

3. The only Boolean expressions are those which are generated by (1) and (2).

The symbols .AND. and .OR. are usually written $\wedge$ and $\vee$ by logicians, but these symbols are not available on present-day computers. Here, as in other places, we find a symbol preceded and followed by periods. Just as in the case of .P., the periods are needed to separate the symbol from other symbols on either side of it.

The connectives .AND. and .OR. are used to combine two Boolean expressions into a new Boolean expression with its own *truth value*, i.e., its own value *true* or *false*. The understanding is that the truth value of the new expression is completely determined just by the truth values of the original expressions and the particular connective used, and not by the meanings of the expressions involved. We shall agree, then, that no matter what expressions $\mathcal{P}$ and $\mathcal{R}$ might

Table 2.1

$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$ .AND. $\mathcal{R}$	$\mathcal{P}$ .OR. $\mathcal{R}$
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

be, the new expression $\mathcal{P}$ .AND. $\mathcal{R}$ is *true* if and only if both $\mathcal{P}$ and $\mathcal{R}$ have the value *true*, while $\mathcal{P}$ .OR. $\mathcal{R}$ is *true* if and only if either $\mathcal{P}$ or $\mathcal{R}$ (or both) has the value *true*. These agreements are usually summarized by means of Table 2.1, in which all combinations of truth values for $\mathcal{P}$ and $\mathcal{R}$ are considered. Such a table is called a *truth table*.

The third line of Table 2.1 says, for example, that if $\mathcal{P}$ is a Boolean expression whose truth value is *true* (such as 2 .L. 3) and $\mathcal{R}$ is an expression whose truth value is *false* (such as 2 + 3 .E. 4), then the expression

$$2 \text{ .L. } 3 \text{ .AND. } 2 + 3 \text{ .E. } 4$$

is to be assigned the truth value *false*, while the statement

$$2 \text{ .L. } 3 \text{ .OR. } 2 + 3 \text{ .E. } 4$$

is to be assigned the truth value *true*. As another example,

$$(X + Y \text{ .L. } 3 \text{ .AND. } I \text{ .E. } 3) \text{ .OR. } X + Y \text{ .GE. } 2$$

would be *false* if X = 0, Y = 0, and I = 2, but it would be *true* if X = 2, Y = 2, and I = 3; if X = 0, Y = 0, and I = 3; or if X = 2, Y = 2, and I = 2.

The rules given above are enough to generate very general Boolean expressions, and although it is not yet apparent, such logical expressions are the most powerful part of any language. They provide the capacity to make decisions, as we have already seen in the change problem. An example which illustrates this point even more clearly is the Social Security problem, to be considered in Chapter 4. Before going on, however, we must again consider the rules for omitting parentheses. In fact, every time we consider adding any new operations or relations to the language, we must immediately place them in the ranking which we have developed. As soon as we have done this, the general rule for missing parentheses will do the rest.

Consider the example used just above:

$$(X + Y \text{ .L. } 3 \text{ .AND. } I \text{ .E. } 3) \text{ .OR. } X + Y \text{ .GE. } 2$$

This expression will have as its value *true* or *false*, and it is clear that we must determine the values of the *subexpressions* X + Y .L. 3, I .E. 3, and X + Y .GE. 2 before being able to ask about the effect of .AND. and .OR. . This implies that the arithmetic operations and the relations must be applied before any logical operations. In other words, just as the relations have been given a lower rank than any of the arithmetic operations, we now must rank the logical operations below the relations.

The next thing to be determined is the relative ranking of the

logical operations. Strangely enough, although these operations have been in use for many years, logicians still have not been able to agree on a standard ranking. (This only emphasizes the statement made earlier that these rankings are only agreements, and they can be changed by an author whenever he has a good enough reason.) Some authors rank .AND. and .OR. on the same line, and others place .AND. higher than .OR. . Since we must make a decision, let us rank .AND. higher than .OR., so that the example used above

$$(X + Y \text{ .L. } 3 \text{ .AND. } I \text{ .E. } 3) \text{ .OR. } X + Y \text{ .GE. } 2$$

really has redundant parentheses and could have been written

$$X + Y \text{ .L. } 3 \text{ .AND. } I \text{ .E. } 3 \text{ .OR. } X + Y \text{ .GE. } 2$$

just as well.

It is clear from this example, also, that there must be at least one logical operation (.AND. or .OR.) between any two relations (.E., .NE., .L., .LE., .G., and .GE.), and therefore we will never be in the position of having to decide which of two relations must be applied first. They will be applied quite separately, without any interaction between them. It follows that we need not rank relations relative to each other at all, and therefore we might just as well place them all on the same line in the ranking. We have finally arrived at a complete ranking of all our operations and relations:

.ABS.
.P.
− (unary)
*, /
+, − (binary)
.E., .NE., .L., .LE., .G., .GE.
.AND.
.OR.

This ranking, together with the rule that operations of the same rank are applied from left to right, completely specifies the meaning of an expression in which the precedence is not determined by parentheses. As a final illustration, the reader should verify that none of the parentheses in the following example is redundant. In

other words, if any of the parentheses were taken out, the value of the expression would be changed.

```
((A + B) .P. -X .L. 3 .OR. I/(J - K) .GE. X/Y)
                                .AND. N .E. I - 4/((L + I) * Z)
```

PROBLEMS

1. In Section 2.1 several unacceptable names of variables were listed, e.g., 9ED, 1.5, 91, and E1 + 2. Why is each of these unacceptable according to our definition?

2. For each of the following, decide whether it would be acceptable or not as the name of a variable: E1, A2B3, A2.B3, 2B3, 2 * B3. Explain your answer.

3. Which of the following would be acceptable as constants in our language? 9ED, 1.5, 91, E1 + 2, 2B3, −91. Explain your answer.

4. Which of the following are acceptable arithmetic expressions? 3. * B * C, 3. * .ABS. B * C, 3. * .ABS. (B * C), 3. * (.ABS. B), 3. * (((.ABS. (B)) * C) + D. Explain your answer.

5. Using the precedence ranking which we have established, evaluate the following expressions: 3 + 4/2 .P. 2 + 1, 3 + 4 * 2 .P. 2 + 1, 3 * 4 + 2 .P. 2 + 1, (3 * 4 + 2) .P. 2 + 1.

6. Determine whether the following Boolean expressions are acceptable or unacceptable. For each acceptable Boolean expression, find the truth value:

a. 4 .L. 5 .AND. 2 .E. 3

b. 4 .L. 5 .AND. 2 .E. 3 .OR. 4 .E. 5

c. 4 .G. 5 .OR. 2 .E. 3 .OR. 4 .E. 5

d. 4 .G. 5 .OR. 4 .LE. 5

e. 4 .G. 5 .OR. (2 .OR. 3 .NE. 4)

CHAPTER THREE

CONDITIONAL STATEMENTS AND ITERATION STATEMENTS

LET US SUMMARIZE the state of our language. We can now write arithmetic and Boolean expressions, and we can construct a substitution statement such as $Z = X + Y - 3$. If we were to try to write the algorithm given in Figure 1.1 for the change problem, however, we would immediately find two glaring defects. First of all, even with our ability to write Boolean expressions, we have as yet no way to *examine* their truth values to make decisions. Secondly, we need a way to describe the *loop* process, in which *d* takes on a whole sequence of values. Actually, if we can solve the first difficulty, we can "get around" the second one, but only in a very clumsy way. We will show the clumsy method below, but only to emphasize the simplicity of the better solution which we will develop afterward.

3.1 THE SIMPLE CONDITIONAL STATEMENT

In order to make decisions we must be able to examine one or more Boolean expressions, and we must be prepared to specify various alternative actions to be undertaken, depending on the truth values

of the expressions we are considering. We saw such a decision in Figure 1.1 where the value *true* for the expression R = 0 (we now write R .E. 0) led us to one action, e.g., we stopped the algorithm, and the value *false* led us into another action, i.e., another trip around the loop.

One very convenient statement we can add to our language in order to make decisions is the *simple conditional statement*. Here we specify a Boolean expression to be examined and a single statement to be executed if the expression is *true*. It will be understood that if the expression is *false*, we will not execute the other statement, but skip it and go on in sequence. An example of this is the statement

WHENEVER J .E. (J/2) * 2, I = I + 1

where I and J have been declared elsewhere to be of integer mode. This statement represents the following decision and action: If J is even, increase I by 1; if J is not even, do not increase I. In either case, go on to the next statement. (It is not completely obvious, perhaps, how we determine that J is even. Remember that if J is an integer, division by the integer 2 is interpreted in the sense of the greatest integer less than or equal to J/2, i.e., [J/2]. Multiplication by 2 will give us J back again if J is even; otherwise it will give us J - 1. For example, $(14\frac{1}{2}) * 2 = 7 * 2 = 14$, but $(13\frac{1}{2}) * 2 = 6 * 2 = 12$.)

3.2 THE TRANSFER STATEMENT

We are considering here collections of statements, which, when executed, will solve a particular problem. Such a collection of statements is called a *program* (sometimes, a *routine*). (Figure 3.1 is an example of a program.) Occasionally, on the basis of some decision, it will be determined that some of the statements should in fact be skipped. Or perhaps after reaching a certain point in the program it is determined that a section of program which occurred earlier is to be executed again. When the normal sequence in which statements are executed is changed, as in these cases, we refer to the change in sequence as a *transfer*, or *jump*, to another part of the

program. Thus, if we call the first executable statement in a particular program START, we may very well end the program with the statement

TRANSFER TO START

Then, after the other statements in the program have been executed, the execution of the transfer statement will change the sequence so that the statement labeled START is the next to be executed.

We see now that it will also be necessary to allow the second statement in a simple conditional to be a TRANSFER TO statement. Then, on the basis of the decision in the first half of the simple conditional, a transfer will or will not be made to some other part of the program. This is a very important addition to the language, since it enables us to determine, *on the basis of the computation thus far carried out*, whether or not to change the order in which later statements of the program are to be carried out. This will be seen more clearly in Figure 3.1. Of course, in order to indicate a transfer to another part of the program, we must be able to *label* the place to which the transfer is to be made. In other words, we need a way to attach a label such as START to a statement. Let us specify that *statement labels* must have the same form as names of variables, i.e., up to six letters or digits, the first of which must be a letter. Then we can write

WHENEVER R .E. 0, TRANSFER TO FINISH

where FINISH is the label of the last statement in the program.

Below is a clumsy but correct program for the change problem, according to the specifications we have made so far for our language. (We do not really need a variable named *d* in this version since each value of *d* is explicitly used.)

(In Figure 3.1 we use the knowledge we have of the algorithm that if it ever gets to the computation of Q1 at all, then Q1 must equal R, and R must become zero at the next step.) The END OF PROGRAM statement must be used on all programs so that the reader can recognize that he is seeing a complete program. As indicated in Figure 3.1, it may be given a label (FINISH), and transferring to it may then serve as a way to end the computation.

```
        NORMAL MODE IS INTEGER
        Q50 = 0
        Q25 = 0
        Q10 = 0
        Q5  = 0
        Q1  = 0
        R = A
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q50 = R/50
        R = R - 50 * Q50
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q25 = R/25
        R = R - 25 * Q25
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q10 = R/10
        R = R - 10 * Q10
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q5 = R/5
        R = R - 5 * Q5
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q1 = R
FINISH  END OF PROGRAM
```

Figure 3.1

3.3 THE ITERATION STATEMENT

The other gap we need to fill now is the need for handling loops. By far the greatest number of computer programs that are written contain loops of one kind or another, and we shall see examples of several kinds of loops as we continue. For the change problem, we already saw the need for indicating that a collection of statements (the *scope* of the loop) is to be repeated again and again (i.e., *iterated*) with some variable (in our case, *d*), taking a different value each time from some specified list of values. We shall write this kind of iteration statement as follows:

```
THROUGH SCOPE, FOR VALUES OF D = 50, 25, 10, 5, 1
```

Here SCOPE will be the label on the last statement in the scope of the loop; i.e., after executing that statement, the entire scope is executed again with the next value of D.

Unfortunately, although we are much better off with this iteration statement than we were before without it, there is still one small part of the flow diagram in Figure 1.1 that we have not accounted for. When we wrote q_d there, d was a *subscript;* that is, because we knew the value of d each time around the loop, we were aware that we were referring to separate storage locations q_{50}, q_{25}, . . . , q_1, and by specifying d we indicated which of these q's we meant. In Figure 3.1, we referred to Q50, Q25, . . . , Q1. Although the numbers 50, 25, etc., appear as part of the name, there is no reason to tie them to some quantity d. In fact, there was no mention of d in Figure 3.1 at all. We would like now to recapture the dependence of these quantities on d, since our entire loop will depend on the current value of d each time around. What we really need is a way to say that we are dealing with several quantities, all named q, but distinguished by means of a subscript d. We shall need to specify, of course, just what constitutes a legal subscript in our language. In some complicated programs, subscripts may be quite complex, such as $a_{i+2*(j-1)}$. We should therefore not restrict the form of subscripts unnecessarily. We can assume that subscripts will be of integer mode, since they were intended to locate a *position* in storage relative to the beginning of the set of numbers being subscripted. Let us designate the beginning element of the set as having subscript zero. We may then specify that any nonnegative integer expression may qualify as a subscript. Examples would be A_i, B_{i+j}, and $A_{i+2*(j-1)}$, provided that in each case the value of the subscript is not negative.

There is now another difficulty. It was mentioned before that even large computers have a fixed amount of storage. In order to allocate properly the available storage, therefore, it will be necessary to specify how much is to be set aside for each subscripted variable. Accompanying any use of subscripts would be the obligation to declare the highest subscript to be used, so that adequate storage can be set aside for this *vector.* (Generally, one refers to a string of quantities, indexed by some subscript, as a *vector;* we do so here, also.) The number of quantities is referred to as the *dimension* of the vector. In the change problem, then, we shall refer to Q(50), Q(25), . . . , Q(1) and make the following declaration as to the highest subscript:

```
DIMENSION Q(50)
```

[To be absolutely precise, since we shall allow the use of zero as a subscript, when 50 is the highest subscript we actually have 51 quantities in the vector, e.g., Q(0), . . . ,Q(50); so the true dimension should be 51. It is more convenient, however, to declare the highest subscript used and still refer to it as the *dimension.*] Note that the expression in parentheses is the subscript, and it is the presence of the parentheses that indicates that it is a subscript.

We are now in a much better position to write a program for the change problem. The flow diagram still looks the same as in Figure 1.1, but now we may even set up the initialization of the Q's as a loop. Notice that the essential structure of the algorithm, e.g., two loops, shows up very clearly in the language we are developing. We shall put the mode and dimension statements last this time to emphasize that *declarations* may appear anywhere in the program (except after the END OF PROGRAM statement).

```
        THROUGH LOOP1, FOR VALUES OF D = 50, 25, 10, 5, 1
LOOP1   Q(D) = 0
        R = A
        THROUGH LOOP2, FOR VALUES OF D = 50, 25, 10, 5, 1
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q(D) = R/D
LOOP2   R = R - D * Q(D)
        NORMAL MODE IS INTEGER
        DIMENSION Q(50)
FINISH  END OF PROGRAM
```

Figure 3.2

In reading this program, remember that the statement labeled LOOP1, which is the entire scope of the first loop, will be executed for each of the indicated values of D before the statement R = A is executed at all.

PROBLEMS

1. Since most people make occasional errors, it is useful to predict the behavior of the computer when an error is made. Sometimes a program will accidentally produce the correct answers in spite of errors, but this does not happen very often. In each of the following programs, Figure 3.2 has

been modified to contain an error. What computation will result in each case?

a.

```
        THROUGH LOOP1, FOR VALUES OF D = 50, 25, 10, 5, 1
        Q(D) = 0
LOOP1   R = A
        THROUGH LOOP2, FOR VALUES OF D = 50, 25, 10, 5, 1
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q(D) = R/D
LOOP2   R = R - D * Q(D)
        NORMAL MODE IS INTEGER
        DIMENSION Q(50)
FINISH  END OF PROGRAM
```

b.

```
        THROUGH LOOP1, FOR VALUES OF D = 50, 25, 10, 5, 1
LOOP1   Q(D) = 0
        R = A
        THROUGH LOOP2, FOR VALUES OF D = 50, 25, 10, 5, 1
        WHENEVER, R .E. 0, TRANSFER TO FINISH
LOOP2   Q(D) = R/D
        R = R - D * Q(D)
        NORMAL MODE IS INTEGER
        DIMENSION Q(50)
FINISH  END OF PROGRAM
```

2. In describing the simple conditional statement in Section 3.1, we apparently allowed any statement to follow the comma after the Boolean expression, although we have had occasion so far to use only a transfer statement. We also saw how a substitution statement might be used, such as in the following:

```
WHENEVER I .E. K, I = I + 1
```

which states that I is to be increased by 1 whenever it is equal to K. We have now seen another kind of statement, however, which *cannot* be used in a simple conditional, e.g., the iteration statement. Thus, we cannot write

```
WHENEVER I .E. K, THROUGH LOOP2, FOR VALUES OF J = 1, 2, 3
```

Why must we rule out this use of the iteration statement? (*Hint:* What happens to the loop if the Boolean expression I .E. K is *false?*)

3. In the following sequence of statements, what is the value of Y when the computation reaches the point indicated by the three asterisks? Answer this question for each of the sets of values listed below.

```
    Y = B
    WHENEVER .ABS. (B - C) .LE. A, TRANSFER TO Q
    Y = Y + C
Q   Y = Y + A
    * * *
```

a. A = 2, B = 5, C = 6
b. A = −11, B = 5, C = 6
c. A = 0, B = 0, C = 0

CHAPTER FOUR

THE SOCIAL SECURITY PROBLEM

A GREAT MANY people in the United States are affected by the federal social security program. While those who are self-employed compute their own taxes, it is usually the employer's responsibility to deduct an employee's tax from his wages along with the withholding tax and perhaps other taxes. The social security tax is based on a tax rate (to be called RATE here) and a threshold amount (THRESH). The employee's income is taxed at the given rate until it reaches the threshold amount, and then there is no further tax during the remainder of the year. Thus, if RATE = .03 and THRESH = $4800, a man earning $6000 would be taxed $144 on the first $4800, and that would in fact be his total social security tax for that year. Let us suppose that this tax computation is to be given to the computer (most likely as part of a much larger complete payroll computation). We will assume that the employee's total wages up to, but not including, this week's earnings are called WAGES. Let this week's earnings be called SALARY. Then a program for the tax computation (which leaves WAGES increased, ready for next week's computation) might be written as in Figure 4.1. (The digit 1 just to the left of the point at which statements are

started indicates that the line so marked is a continuation of the previous line.)

```
  WHENEVER WAGES + SALARY .LE. THRESH, TAX = RATE * SALARY
  WHENEVER  WAGES + SALARY .G. THRESH .AND.
1       THRESH .G. WAGES,  TAX = RATE * (THRESH - WAGES)
  WHENEVER  THRESH .LE. WAGES,  TAX = 0
  WAGES = WAGES + SALARY
  END OF PROGRAM
```

Figure 4.1

4.1 THE COMPOUND CONDITIONAL STATEMENT

Although the program in Figure 4.1 does illustrate the use of more complicated Boolean expressions than we saw in the change problem, it is still not a very good algorithm, in the sense that there is some wasted computation built right into it. As it is now written, whenever one of the Boolean expressions is *true*, we would compute TAX, but then we would go right on to the next statement, even though it means examining expressions which must be *false* (since a previous one was *true*). In this problem, as in many others, we know that the three alternatives are mutually exclusive; so there is really no need to test the other conditions once one of them has the value *true*. It would be useful to have a way to choose that alternative which is *true* and then skip the rest of the possible alternatives.

Let us then add to our language the *compound conditional*. Before we describe it, however, let us see what it will do for the social security problem. Figure 4.2 shows the same program as in Figure 4.1, written to use the compound conditional.

```
WHENEVER WAGES + SALARY .LE. THRESH
    TAX = RATE * SALARY
OR WHENEVER THRESH .G. WAGES
    TAX = RATE * (THRESH - WAGES)
OTHERWISE
    TAX = 0
END OF CONDITIONAL
WAGES = WAGES + SALARY
END OF PROGRAM
```

Figure 4.2

(The indention of the three statements in which TAX is actually computed is entirely for readability. Since blank spaces are ignored, a few blanks at the front end of a statement will not affect the interpretation of the statement.) From Figure 4.2 we see that the program now has the following structure. There is a WHENEVER statement, very much like a simple conditional, except that the Boolean expression is not followed by a comma, nor by another statement written on the same line. Instead, a section of program follows immediately below the WHENEVER statement. (It is only a one-line section of program in this case, but in more complex problems, it could be much more complicated.) Then there is an OR WHENEVER statement which is also followed by a section of program. The initial word "OR" simply serves notice that this is not the first of the statements in this compound conditional. There may be several OR WHENEVER sections (or none at all) in a compound conditional. The last section may be (but need not be) an OTHERWISE section, and the last statement of the entire compound conditional *must* be an END OF CONDITIONAL statement.

We must now describe how such a sequence of statements is executed. We first evaluate the Boolean expression in the WHENEVER statement. If it is *true*, we execute the section of program which follows directly below it. If it is *false*, we skip the section of program which follows it, and we evaluate the Boolean expression in the first OR WHENEVER statement, and so on. The first time we find a Boolean expression which is *true*, we execute its section of program, and *then skip all the remaining statements* in the compound conditional. The END OF CONDITIONAL statement serves very conveniently to let us know where to transfer after executing one of the sections above it. Of course, it may happen that none of the Boolean expressions is *true*. This does not create any difficulties, however, since then we merely skip all that part of the computation. The OTHERWISE statement serves as a trap for any case not covered by the preceding Boolean expressions. In other words, OTHERWISE has exactly the same effect as OR WHENEVER 2 .L. 3 in that we always execute its section of program if no earlier Boolean expression was *true*. The reader should now verify that the program in Figure 4.2 does indeed solve the social security problem.

It is possible to see the difference between the programs in Figures

4.1 and 4.2 even more clearly by drawing their flow diagrams. The diagram for Figure 4.1 is given as Figure 4.3, while the diagram for Figure 4.2 is given as Figure 4.4.

It is interesting to observe that there is a single formula for computing the social security tax; so we could have written just one

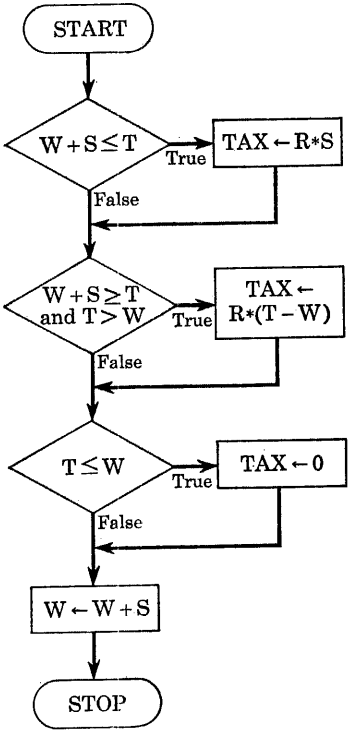


Figure 4.3

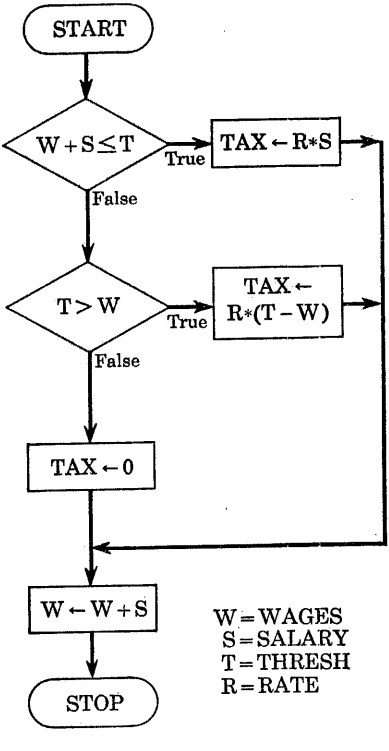


Figure 4.4

substitution statement. Here it is (with T for THRESH, S for SALARY, and W for WAGES, for simplicity):

$$\text{TAX} = \text{RATE} * (\text{S} + \text{T} - \text{W} - .\text{ABS.}\ (\text{S} + \text{W} - \text{T}) + .\text{ABS.}\ (\text{S} + \text{T} - \text{W} - .\text{ABS.}\ (\text{S} + \text{W} - \text{T})))/4$$

The reader should verify that this formula works by trying a few sets of values, such as T = 4800, W = 4000, and S = 200, or T = 4800,

W = 4600, and S = 400. Later, when we are discussing functions (Section 6.5), we shall see how this formula is derived.

If we were to write a program which made use of this formula, we would observe that there are in it some *common subexpressions.* The expression S + T − W occurs in two places, and the expression S + W − T also occurs in two places. Rather than go through the computation for each of these expressions twice, we may write separate statements to compute them as the values of some new variables, say Z1 and Z2, and then use these in the long expression. We would then obtain the following program.

```
Z1 = S + T − W
Z2 = S + W − T
TAX = RATE * (Z1 − .ABS. Z2 + .ABS. (Z1 − .ABS. Z2))/4
W = W + S
END OF PROGRAM
```

We could go one step further to obtain an even better program by noting that the compound expression Z1 − .ABS. Z2 occurs in two places. This would lead to the following program:

```
Z1 = S + T − W
Z2 = S + W − T
Z3 = Z1 − .ABS. Z2
TAX = RATE * (Z3 + .ABS. Z3)/4
W = W + S
END OF PROGRAM
```

Of course, *better* means here *fewer arithmetic operations performed,* even though we may write more statements in the program.

PROBLEMS

Be sure to draw a flow diagram to organize the algorithm before trying to write the program. A solution to the first problem is given after Problem 4. Write your own solution before consulting the solution given there. They will probably differ, since there are usually several ways to organize an algorithm for a particular problem.

1. If P is the day, Q the month, and N the year of some event, then if we compute D, where

$$D = P + 2 * Q + [3 * (Q + 1)/5] + N + [N/4] - [N/100] + [N/400] + 2$$

then the remainder R obtained when D is divided by 7 gives the day of the week on which the event occurred. January and February should be considered the thirteenth and fourteenth months of the preceding year. Here we interpret R = 0 as Saturday, R = 1 as Sunday, and so on. Write a flow diagram and program which will produce the day of the week (as R) for a given date. Assume that a date such as February 14, 1960, is given in the form P = 14, Q = 2, and N = 1960 (i.e., as the actual date), and include statements to test for January and February and adjust the data accordingly. Why does this method work? On which day of the week did July 4, 1776, fall?

2. The formulas for the two solutions of the quadratic equation

$$AX^2 + BX + C = 0$$

are

$$X1 = (-B + \sqrt{B^2 - 4AC})/2A \qquad X2 = (-B - \sqrt{B^2 - 4AC})/2A$$

Since we do not yet have a way to find a square root in our language, let us just write SQRT.(Z) for the square root of Z. Write a flow diagram and program to compute X1 and X2 from A, B, and C. If A = 0, you cannot divide by 2A, so you had better set X1 = −C/B, X2 = 0, and set a count of real roots, say R, to 1. (You may assume then that $B \neq 0$.) If there are no real roots, set X1 = X2 = R = 0. If there are two real roots, compute them as the values of X1 and X2 and set R = 2.

3. In a certain town the water bill is (perhaps) computed as follows: If the number of gallons used is below K1, the rate is RATE1. If the number of gallons used is between K1 and K2, the rate is RATE1 for the first K1 gallons and RATE2 for the number of gallons above K1, and so on. Assuming that there are four thresholds K1, K2, K3, and K4 and five rates RATE1, RATE2, RATE3, RATE4, and RATE5, write a flow diagram and program which will start with an amount of water used, called GALLON, and compute the BILL.

4. Given the compound conditional:

```
WHENEVER C + A .L. 0
    B = A
    A = C
    C = B
END OF CONDITIONAL
* * *
```

What will be the values of A, B, and C when the computation reaches the three asterisks, if A, B, and C start with the following values:

a. A = 2, B = 5, C = 6
b. A = −11, B = 5, C = 6
c. A = 0, B = 0, C = 0

A Solution to Problem 1

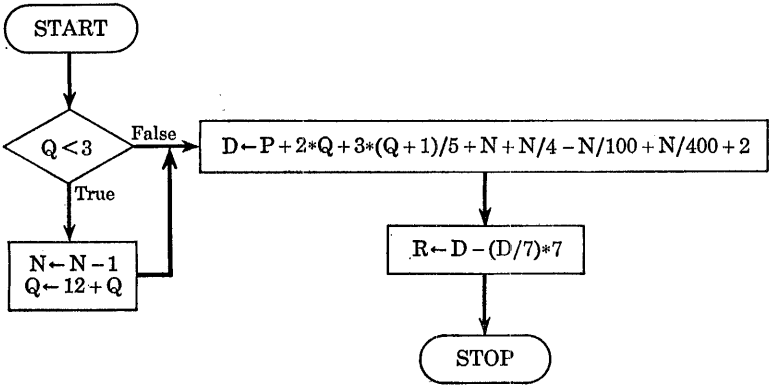


Figure 4.5

The program:

```
NORMAL MODE IS INTEGER
WHENEVER Q .L. 3
    N = N − 1
    Q = 12 + Q
END OF CONDITIONAL
D = P + 2 * Q + 3 * (Q + 1)/5 + N + N/4 − N/100 + N/400 + 2
R = D − (D/7) * 7
END OF PROGRAM
```

Note that the brackets are not needed in the program because of the truncated division which is performed on integers.

CHAPTER FIVE

THE SECRET-CODE PROBLEM

WE HAVE BEEN looking at some small, but typical problems in great detail. From this study we have been able to identify several very important features of problems in general, and these in turn have suggested features which belong in any language whose job it is to express algorithms.

5.1 THE STATEMENT OF THE PROBLEM

Let us consider now the problem of decoding a secret message. There are a great many ways to encode information so that it will be unintelligible to anyone except the person for whom it is intended.[1] A necessary condition, however, is that it be possible for that person to be able to decode the message. This can be described by saying that, no matter what transformation is used to produce the encoded message, there must be an *inverse transformation* which can produce the original message again. Moreover, practical considerations dictate that the *key* be easily transmitted. (The key is the designation

[1] See, for example, Cryptography in an Algebraic Alphabet, by Lester S. Hill, *Amer. Math. Monthly*, **32**:306–312 (1929).

of the particular transformation used, or the designation of the inverse transformation which is needed to decode the message.) The key might be the name of a book containing the code, or it might be some number that specifies a permutation of the alphabet, or it might be any one of a number of other possible devices. We shall use a rather simple code, just to illustrate the ideas, and afterward mention a few of the ways in which the code could be made more complicated.

Let us suppose that the key to this particular code is a simple permutation of the alphabet. To be more specific, we must explain exactly what *alphabet* means here. Our alphabet will consist of the letters A to Z in normal order, followed by the digits 0 to 9, which are in turn followed by the characters +, *, and /, in that order. The *standard alphabet* for this code is thus:

ABCDEFGHIJKLMNØPQRSTUVWXYZ0123456789+*/

(Note that the letter O has a line through it to distinguish it from zero.) We shall assign a *position number* (starting with 0) to each character in the standard alphabet, as shown in Table 5.1.

To encode a message in the code which we are going to use, we shall specify as a key a particular permutation of the alphabet. As an example, we shall use the key (selected at random) shown in Table 5.1.

The encoding rule for this particular code will be as follows: Given a message to encode, we shall assign to each character in the message a number, called its *shift*. The first character will have shift 5, the next 10, the third 15, and so on, with the nth character having a shift of $5n$. To encode any particular character in the message, take its position number in the standard alphabet, add its shift, and look up the key character with that sum as its position number. Blanks will be ignored.

Consider the message:

FENCE TAKEN/PLEASE ADVISE

According to the above rule, we would encode the F as the character 4, since its position number as a character in the standard alphabet is 5, its shift is 5, and the key character with position number 10 is 4.

Table 5.1

Standard alphabet	Position number	Key alphabet
A	0	A
B	1	D
C	2	G
D	3	J
E	4	M
F	5	P
G	6	S
H	7	V
I	8	Y
J	9	1
K	10	4
L	11	7
M	12	+
N	13	B
Θ	14	E
P	15	H
Q	16	K
R	17	N
S	18	Q
T	19	T
U	20	W
V	21	Z
W	22	2
X	23	5
Y	24	8
Z	25	*
0	26	C
1	27	F
2	28	I
3	29	L
4	30	Θ
5	31	R
6	32	U
7	33	X
8	34	0
9	35	3
+	36	6
*	37	9
/	38	/

Similarly, the first E in the message would be encoded as an E, while the second E in the message would be encoded as the character L, since its shift amount is 25.

We soon find, however, that the sum of the position number and the shift exceeds 38, and we need to extend the encoding rule to cover this situation. The simplest way to handle this is to assume that after the last character of the key alphabet we start over again with the first character. Thus, the N in TAKEN has position number 13 and shift 50, so that its key character would have position number 63. This is position 24 in the second listing of the key alphabet, so that N is encoded as the character 8. In general, although the key-character position numbers will become very large, all we need to do is subtract out complete key alphabets, i.e., subtract 39s, until we have left a number between 0 and 38, and then use that as the key-character position number. The reader should verify *before going on* that the entire message then encodes as follows:

4EI2L 43748 H6936 W7+WM 37G

(Standard cryptographic procedure is to write encoded characters in groups of five so that the spacing will be meaningless.) It should be noted that a simpler procedure than repeated subtraction of 39 is dividing by 39 and retaining the remainder. Thus, when we subtracted 39 from 63 in the example above, we could have divided 63 by 39 and used the remainder 24. We shall return to this point later.

In order to draw a flow diagram for the encoding procedure, we shall need some notation. Let us suppose that the original message contains N characters and is stored one character per computer location as LETTER_1, LETTER_2, . . . , LETTER_N, and the standard alphabet consists of the characters stored as STAND_0, STAND_1, . . . , STAND_{38}. Thus $\text{STAND}_0 = \text{A}$, $\text{STAND}_1 = \text{B}$, and so on. We shall refer to the subscript I of STAND_I as its *position number* so that A has position number 0, B has position number 1, and so on. The key alphabet will consist of the characters KEY_0, KEY_1, . . . , KEY_{38}, so that in our example key, $\text{KEY}_0 = \text{A}$, $\text{KEY}_1 = \text{D}$, and so on. Here, also, we shall refer to the subscript J of KEY_J as its position number so that, for example, D has position number 1 here. We shall refer to the current shift amount as S, so

that at the beginning S = 5. Finally, let us store the encoded message as the characters

$$CODE_1, CODE_2, \ldots, CODE_N$$

5.2 ANOTHER ITERATION STATEMENT

If we now begin to plan the flow diagram, we soon see the need for a new kind of iteration statement. We need to be able to say: "When we consider the message characters $LETTER_1$, $LETTER_2$, and so on, we shall want to refer to $LETTER_K$ with K starting at 1 and increasing by 1 until it exceeds N, the number of characters in the message. Moreover, for each value of K, i.e., for each character in turn, we shall want to do the following block of computation." This is very similar to the iteration statement used in the change problem,

THROUGH SCOPE, FOR VALUES OF D = 50, 25, 10, 5, 1

except that now we do not intend to give a definite list of values for K. We wish to give a starting value (K = 1), a modification rule (K = K + 1), and a termination condition (K > N). We cannot, in fact, list explicitly all the values to be used for K, since N will vary from one message to another, and a program written for this encoding procedure should be expected to work for any message. Since the modification of the iteration variable K generally takes the form of adding something to K (i.e., incrementing K), we shall specify, along with an initial value for K, the amount to be added each time the scope is executed. This amount could be the value of some expression, and it could be negative as well. Let LOOP be the label on the last statement of the *scope*, i.e., the last statement in the block of statements to be executed for each value of K. We would now write

THROUGH LOOP, FOR K = 1, 1, K .G. N

(Note that we may distinguish this kind of iteration statement from the explicit list kind by the omission of the words VALUES OF.) This statement is to be interpreted as: Perform the computation written from this point in the program through the statement labeled

LOOP, starting with K = 1. Each time, before carrying out the computation, test to see if the termination condition (K .G. N) is satisfied. If it is satisfied, proceed immediately from the first statement after LOOP. If the termination condition is not satisfied, execute the scope and increment K by 1.

A convenient way to represent such iteration situations in a flow diagram is to use the *iteration box*, shown in Figure 5.1.

In Figure 5.1 we enter the iteration box at the initialization (K ← 1). We then proceed across the upper diagonal line to test the termination condition (K > N). If this condition were *true*, it could happen that the scope (i.e., the computation within the loop) would not be executed at all. (We saw a situation similar to this in Figure 1.1

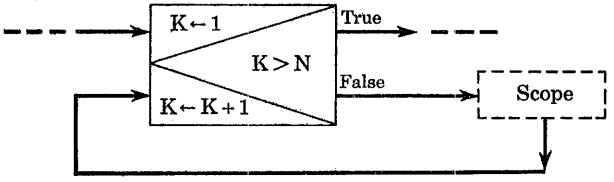
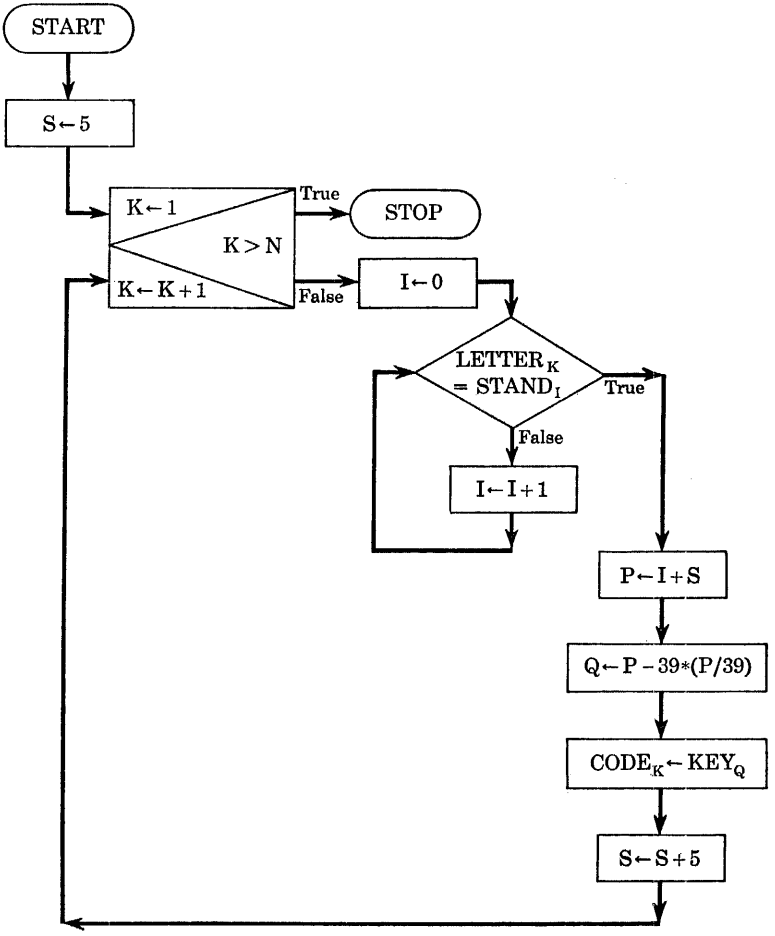


Figure 5.1

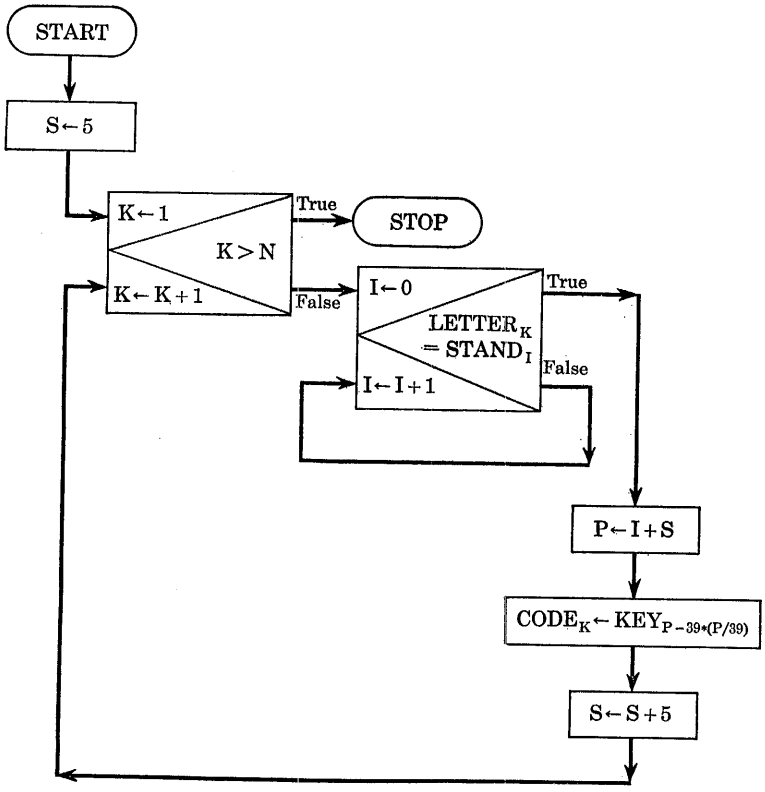
for the change problem, when A = 0.) In the particular problem shown in Figure 5.1 the termination condition would not be *true* (unless N = 0); so we would execute the scope and return to the modification section of the iteration box (K ← K + 1), after which we would cross the lower diagonal to test the termination condition again, and so on.

We are now in a position to construct a flow diagram for the encoding problem. In fact, Figures 5.2*a* and 5.2*b* show two different diagrams. The strategy is the same in the two diagrams, but Figure 5.2*a* is somewhat easier to understand. (Figure 5.2*b* illustrates a more concise way to express the same algorithm.) In Figure 5.2*a*, then, we start by initializing the shift S to 5. Each time we finish encoding a character and are about to move to the next character, we shall increase S by 5, in accordance with our definition of the amount of shift to be associated with each character. After the initialization of S, we move into a large iteration, so large, in fact, that its scope includes the rest of the program. It is an iteration on



S = shift
$STAND_I$ = standard alphabet character with position number I
$LETTER_K$ = Kth character of message to be encoded
P = position number of desired character in key alphabet (before reduction)
Q = reduced position number of desired character in key alphabet
$CODE_K$ = Kth character of encoded message
KEY_Q = character selected from key alphabet according to position number Q

Figure 5.2*a*



S = shift
$STAND_I$ = standard alphabet character with position number I
$LETTER_K$ = Kth character of message to be encoded
P = position number of desired character in key alphabet
$CODE_K$ = Kth character of encoded message
$KEY_{P-39*(P/39)}$ = character selected from key alphabet

Figure 5.2*b*

the value of K, in exactly the form described above in the discussion of Figure 5.1. In this algorithm, K will act as a "pointer," indicating in each iteration which character in the message is currently being encoded. In order to do this, K starts with the value 1, increases by 1 after each iteration, and causes the loop to be terminated when it finally exceeds N.

The computation within the loop accomplishes the encoding of the current (Kth) character of the message. In fact, at the end of the loop we see the substitution

$$\text{CODE}_K \leftarrow \text{KEY}_Q$$

which puts a suitably chosen character from the key alphabet into the Kth position in the region in which the encoded message is stored. This is followed by the substitution

$$S \leftarrow S + 5$$

which increases S, as we indicated above. The earlier part of the scope of the iteration has the job of determining the appropriate character in the key alphabet or, more exactly, determining its *position number*. The first step is to find out where in the standard alphabet the current message character LETTER_K occurs. To do this, we shall compare LETTER_K with each of the letters in the standard alphabet and ask of each one whether it is the same character as LETTER_K. In Figure 5.2*a* there is another "pointer" (the variable I), which will move us along the standard alphabet. We start with I = 0 and ask if LETTER_K is the same as STAND_I (in this case STAND_0, which is A). If the answer is negative, we increase I by 1 and ask the question about STAND_1, and so on. Eventually we will come to the standard alphabet character which is the same as LETTER_K, and the answer to the question will be in the affirmative. At this time, the current value of I will be the position number of the character STAND_I just identified as the one that matches LETTER_K. We may now compute P, the position number of the corresponding key character, by adding the current value of S to the standard position number I. This is done in the substitution

$$P \leftarrow I + S$$

Since we have already seen that P may exceed 39 and must be reduced to a value, say Q, which is between 0 and 38, the substitution

$$Q \leftarrow P - 39 * (P/39)$$

is included. Note that P/39 is an integer, since we are using integer division, and indicates how many multiples of 39 are contained in P. In other words, since $10/39 = 2$, we see that there are two 39s (or

78) in 100. If we subtract 78 from 100, we have a measure of how far we are into the next 39. In terms of our alphabets, P/39 indicates how many complete alphabets can be subtracted out. Multiplying P/39 by 39, then, gives us the actual number of characters in that many alphabets, and if we subtract 39 * (P/39) from P, we have Q, the number between 0 and 38 which represents the extent to which we have gone into the next alphabet. We may then use Q to select the key character. Note that if P is already between 0 and 38, we have P/39 = 0, so that nothing is actually subtracted from P. In this case, it turns out that Q = P.

To illustrate the algorithm in Figure 5.2*a* by using our previous example starting with the word FENCE, we first set S = 5 and K = 1. In other words, we are now considering the first character of the message, LETTER_1 = F. The inner loop on I now searches out the position number of F as a character in the standard alphabet. Thus, when I = 0, LETTER_1 .E. STAND_0 is *false*, since STAND_0 = A, so that I is increased to 1. Continuing in this way, we find that LETTER_1 .E. STAND_5 is *true*, and we leave the inner loop with I = 5. Since S = 5, we compute P = 5 + 5 = 10, Q = 10 − 39 * ($1\frac{0}{39}$) = 10 − 0 = 10, and $\text{CODE}_1 = \text{KEY}_{10} = 4$. Then S is increased to 10, and we return to the iteration box which increases K by 1, thus effectively moving us to the next message character LETTER_2 = E. This time we leave the inner loop with I = 4, so that P = 4 + 10 = 14, Q = 14 − 39 * ($1\frac{4}{39}$) = 14 − 0 = 14, and $\text{CODE}_2 = \text{KEY}_{14}$ = E.

The substitution Q = P − 39 * (P/39) illustrates a very convenient formula for finding a remainder, provided P is an integer and the division is the truncated integer division which was introduced in the discussion of the change problem. This remainder formula is in fact very similar to the formula

$$R = R - d \cdot q_d$$

used in the change problem, since the statement $q_d = [R/d]$ becomes $q_d = R/d$ if integer division is used, and substituting this expression for q_d in the formula for R, we have

$$R = R - d \cdot (R/d)$$

which is now in exactly the same form as the formula used above

for the key-character position number. Since 39 * (P/39) is the largest multiple of 39 less than or equal to P, we see that P − 39 * (P/39) is the remainder.

As a further example, if P = 63, as in our previous illustration, we have

$$\begin{aligned} P - 39 * (P/39) &= 63 - 39 * (63/39) \\ &= 63 - 39 * (1) \\ &= 63 - 39 \\ &= 24 \end{aligned}$$

Also, if P = 128, we would have

$$\begin{aligned} P - 39 * (P/39) &= 128 - 39 * (128/39) \\ &= 128 - 39 * (3) \\ &= 128 - 117 \\ &= 11 \end{aligned}$$

In this case, if the character 2 were in the twentieth message position (shift = 100), it would be encoded as 7. The program for the encoding problem corresponding to the diagram in Figure 5.2*a* is given in Figure 5.3*a*.

```
        NORMAL MODE IS INTEGER
        S = 5
        THROUGH LOOP1, FOR K = 1, 1, K .G. N
        I = 0
LOOP2   WHENEVER LETTER(K) .E. STAND(I), TRANSFER TO FOUND
        I = I + 1
        TRANSFER TO LOOP2
FOUND   P = I + S
        Q = P − 39 * (P/39)
        CODE(K) = KEY(Q)
LOOP1   S = S + 5
        DIMENSION STAND(38), KEY(38), LETTER(1000), CODE(1000)
        END OF PROGRAM
```

Figure 5.3*a*

The DIMENSION statement in this program serves for each of the vectors in the list. Since we do not know how large N might be, i.e., how long the messages will be, we shall arbitrarily set an upper bound of 1000 characters per message.

We now observe that the search of the standard alphabet inside the scope of the K loop really has the complete structure of a loop, except that it does not have any scope. The variable I is initialized and it is modified, and there is a termination condition. This shows that it is quite reasonable to have loops in which there is no scope, i.e., no actual computation within the loop. The task that the loop is supposed to accomplish is performed right in the termination condition. It would be convenient to use the available iteration statement for such a loop as well. All we have to do in the program is put the label which indicates the end of the scope right on the itera-

```
        NORMAL MODE IS INTEGER
        S = 5
        THROUGH LOOP1, FOR K = 1, 1, K .G. N
LOOP2   THROUGH LOOP2, FOR I = 0, 1, LETTER(K) .E. STAND(I)
        P = I + S
        CODE(K) = KEY(P - 39 * (P/39))
LOOP1   S = S + 5
        DIMENSION STAND(38), KEY(38), LETTER(1000), CODE(1000)
        END OF PROGRAM
```

Figure 5.3*b*

tion statement itself. This is illustrated in the flow diagram of Figure 5.2*b* and in the program of Figure 5.3*b*. One other simplification that is made in the more concise diagram and program is that we do not compute Q in one statement and use it in another. The expression P − 39 * (P/39) is used directly as the subscript of KEY.

5.3 THE DECODING PROBLEM

Let us consider now the inverse transformation, i.e., the *decoding* of the secret message. Using the example of the preceding sections

FENCE TAKEN/PLEASE ADVISE

which was encoded as

4EI2L 43748 H6936 W7+WM 37G

we now need a procedure which will start with the first 4 and pro-

duce F, then produce an E from the E, an N from the I, and so on. The obvious solution is to reverse whatever was done in the encoding process. If there were no shift involved, we would simply find the coded character in the key alphabet and read off the corresponding standard alphabet character. Thus, without any shifting, the original message above would be encoded as

PMBGM TA4MB /H7MA QMAJZ YQM

and the decoding process is obvious.

The effect of the shift is to move us down the list before reading off the corresponding key character. To decode, then, we need to reverse the direction of shift. For example, the first character 4 in the encoded message has position number 10 as a character in the *key* alphabet. Subtracting the shift amount 5 yields 5, the position number of F in the standard alphabet. This is a very simple procedure, except that we soon encounter negative position numbers after subtracting larger shift amounts. Again we may move through successive listings of the standard alphabet by *adding* 39 as many times as necessary to produce a number between 0 and 38. Thus, for the sixth character 4 in the encoded message, we have key position number 10, and shift amount 30, leaving a standard position number of $10 - 30 = -20$. If we add 39, we obtain the correct standard position number 19 for the character T, from which it came originally. As another example, the character G in the twenty-third position of the encoded message has key position number 2 and shift 115, leaving a standard position number of -113. Instead of just adding 39 several times, let us again use the earlier formula

$$P - 39 * (P/39)$$

and let $P = -113$. We obtain $-113 - 39 * (-113/39) = -113 - 39 * (-3) = -113 + 117 = 4$, which is the standard position number for the character E. It is clear then that we may use this formula even for negative position numbers.

It is now a simple matter to write the program for decoding messages. We need assume only that the message to be decoded is stored as $\text{LETTER}_1, \ldots, \text{LETTER}_N$, that the decoded message which results will be in $\text{CODE}_1, \ldots, \text{CODE}_N$, and that the *stand-*

ard alphabet is stored as $KEY_0, \ldots, KEY_{38}$. Then the only change necessary in the program we already have in Figure 5.3*b* is that the amount of shift starts with the value -5, and it *decreases* by 5 for each character. Thus, the *source message* (the message to be encoded or decoded) always goes into the vector LETTER, and its alphabet goes to STAND. The alphabet to which we are translating is in KEY, and the message which is produced is in CODE. Let us suppose that the signal as to whether we are encoding or decoding is the value of a variable named SIGNAL, i.e., SIGNAL = 1 if we are encoding and SIGNAL = -1 if we are decoding. Then Figure 5.4 shows one program which will do the entire job. This program should be compared with Figure 5.3*b*.

```
        NORMAL MODE IS INTEGER
        S = 5 * SIGNAL
        THROUGH LOOP1, FOR K = 1, 1, K .G. N
LOOP2   THROUGH LOOP2, FOR I = 0, 1, LETTER(K) .E. STAND(I)
        P = I + S
        CODE(K) = KEY(P - 39 * (P/39))
LOOP1   S = S + 5 * SIGNAL
        DIMENSION STAND(38), KEY(38), LETTER(1000), CODE(1000)
        END OF PROGRAM
```

Figure 5.4

5.4 CONGRUENCE

It is interesting to see the mathematics which was really involved in the encoding-decoding problem. We needed to substitute for an integer which was too large (positive or negative) a number between 0 and 38, and this small positive integer was to differ from the original integer by a multiple of 39. There are obviously many integers differing from some given integer by a multiple of 39, but we needed a particular one, e.g., the one between 0 and 38. If we put into one collection all integers differing from 0 by a multiple of 39 (e.g., 0, 39, 78, -39, etc.) and into another collection all integers differing from 1 by a multiple of 39 (e.g., 1, 40, 79, -38, etc.), and so on, we obtain 39 different, nonoverlapping collections. Moreover, every

integer is in one of these collections. Each collection will contain exactly one integer between 0 and 38, and we shall call this integer the *representative* of the collection.

An interesting property of these collections is that the difference $a - b$ of any two integers a and b in the same collection will be a multiple of 39. To see why this is true, let us suppose that the collection in question has as its representative the integer k $(0 \leq k \leq 38)$. Then

$$a = k + 39r$$
$$b = k + 39s$$

for some integers r and s. Then

$$a - b = (k + 39r) - (k + 39s) = 39(r - s)$$

so that $a - b$ is in fact a multiple of 39. It is also true that if a and c are in different collections, then a and c cannot differ by a multiple of 39. For, suppose the representative of the collection containing c is k' $(0 \leq k' \leq 38)$, with $k' \neq k$. Then, for some integer t,

$$a = k + 39r$$
$$c = k' + 39t$$
$$a - c = (k - k') + 39(r - t)$$

If $a - c$ were a multiple of 39, $k - k'$ would have to be a multiple of 39 also. But $k - k'$ must satisfy the inequality

$$0 \leq |k - k'| \leq 38$$

so $k - k' = 0$ and $k = k'$, a contradiction.

We shall say that *a is congruent to b modulo* 39 if a and b differ by a multiple of 39. In terms of the collections of integers just described, we see that we have simply put into each collection all those integers which are congruent to each other. The usual way to write congruence of two numbers is

$$a \equiv b \bmod 39$$

where the three lines $\equiv$ are intended to suggest the equals sign. In many ways, congruence is very much like equality (although it is obviously not the same as equality). For example, if $a \equiv b \bmod 39$ and $c \equiv d \bmod 39$, then $(a + c) \equiv (b + d) \bmod 39$. To see how

an assertion of this kind may be argued, we translate the congruence statement $a \equiv b \bmod 39$ into the statement that for some integer k, $a - b = 39k$, i.e., that $a - b$ is a multiple of 39. Similarly, for some integer k', $c - d = 39k'$. Then $(a + c) - (b + d) = (a - b) + (c - d) = 39k + 39k' = 39(k + k')$, so that $(a + c) - (b + d)$ is a multiple of 39, and therefore $(a + c) \equiv (b + d) \bmod 39$. Other properties of congruence may be argued in a similar way.

Before we go any further, it should be pointed out that use of the number 39 as the *modulus* (or *base*) of the congruence was a convenient step from the encoding problem. Any other integer could have served as well. Thus, we could talk about congruence modulo 7, for example, and state that $24 \equiv 10 \bmod 7$. With this understanding, we shall continue to use 39 as the modulus, but the reader should be continually checking that no special properties of the number 39 are used.

In the discussion of the encoding problem we noted that, instead of subtracting multiples of 39 from a large positive integer to find its representative between 0 and 38, one could divide by 39 and use the remainder as the representative. To see that this will always yield the correct number as the representative, we may argue as follows: Suppose a is a positive integer and we wish to find its representative. If we divide a by 39, we obtain a quotient q and a remainder r $(0 \leq r \leq 38)$. Thus, $a = 39q + r$. Since $a - r = 39q$, we see that a and r are in the same collection. But r satisfies the condition $0 \leq r \leq 38$; so it must be the representative.

5.5 MORE COMPLEX CODES—RANDOM NUMBERS

There are, of course, many devices which may be used to make it harder to *break* a code (i.e., discover the decoding transformation). Here we shall simply point out some of the directions in which one might go to achieve this desired complexity. It is very important to be sure that each device used has its inverse transformation. A code that could not be translated by anyone would be worse than useless.

We shall assume here that a source of random numbers is available. Not completely random, however, since the inverse transfor-

mation which will be used for decoding it will undoubtedly need to use the same numbers in some inverse way, as the shift amount was used earlier. We need, then, numbers which have as much built-in randomness as possible, but which can be generated in a predictable way by some rule. (One might have several rules and make the choice among them part of the key which describes the decoding transformation.) Numbers which are random in this sense are usually called *pseudorandom* numbers. We shall discuss the generating rules in the next chapter. Let us agree to use the term *random* to mean *pseudorandom*, unless otherwise specified.

One of the ways in which random numbers could have been used to complicate the code is to make the amount of shift random. There is one difficulty here, however. How wide a range of shifts should we allow? What about a shift of 3.75? It is clear that we must be

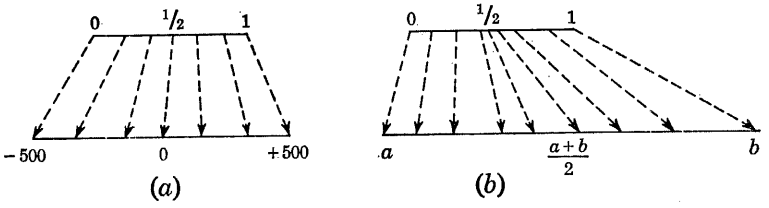


Figure 5.5

able to control these random numbers, and while still allowing random variation, we must somehow interpret this variation in terms of our needs. One very basic decision to be made is the *range* over which these numbers should occur. If we were simulating the economy of the entire country on a computer and wished to include some variation in the amount of income tax to be collected, we might wish for a variation which represented millions of dollars. In our encoding problem, on the other hand, we need various shift amounts, probably ranging from -500 to $+500$. Because these needs are unpredictable, the usual methods of generating random numbers produce numbers between 0 and 1. The user then transforms these numbers to suit himself.

Suppose we have a source of random numbers between 0 and 1, and we need for our application numbers ranging from -500 to $+500$. The following simple formula, inspired by the transforma-

tion illustrated in Figure 5.5*a*, will change any random number x between 0 and 1 into a number y between -500 and $+500$, and the distribution of x's over the range 0 to 1 will be mirrored in the distribution of y's over the range -500 to 500:

$$y = 1000x - 500$$

Thus, when $x = 0$, $y = -500$ and when $x = 1$, $y = +500$. As suggested by Figure 5.5*b*, a more general formula may be obtained which will transform the range 0 to 1 into the range a to b:

$$y = (b - a)x + a$$

[Those who know some analytic geometry may verify that this is the equation of the line passing through the points $(0,a)$ and $(1,b)$. If

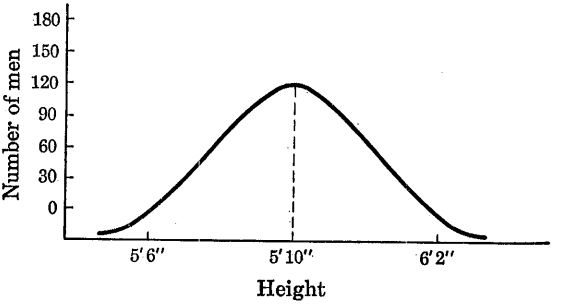


Figure 5.6

a curve other than a line had been used, the distribution of y's would be related to the distribution of the x's in a much more complicated way than we have here. Another interpretation of this transformation is that we are simultaneously changing the scale by multiplying by $b - a$ and shifting the origin by adding a.]

There is still another direction in which our control of the random numbers might go. Many physical phenomena which exhibit randomness still have the property that the values near the *average*, or *mean*, value are much more common than other values. This is true, for example, if we measure the height of various men in some community. Most of the men might be in the range 5 ft 6 in. to 6 ft 2 in., but there will be some men outside this range. One of the distributions which has this property is the *normal distribution*, shown in Figure 5.6 and labeled as if men's heights were being shown.

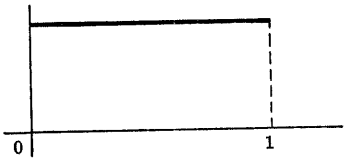
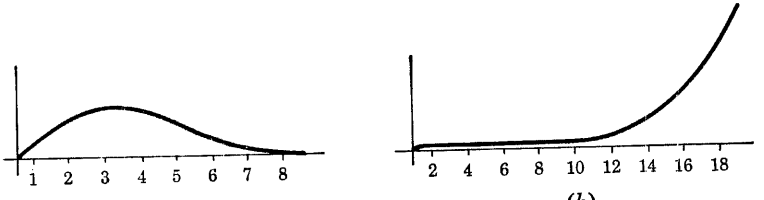


Figure 5.7

On the other hand, if we watch the rain fall on an exposed square of sidewalk, the distribution of drops along the length of the square would not be normal, in the sense of the preceding paragraph. It would be quite *uniform* in that any position would be as likely to receive the same amount of rain as any other position. The uniform distribution is shown in Figure 5.7. There are many other distributions, as well. Two other types are shown in Figure 5.8*a* and *b*. The first distribution (Figure 5.8*a*) might be the interest expressed in dolls by boys of various ages. The second (Figure 5.8*b*) might be the interest expressed in girls by boys of various ages. The user of a source of random numbers would want to specify the kind of distribution of the numbers he would be receiving, e.g., normal, uniform, etc.

Let us assume for the encoding problem that we have a source of *uniformly distributed* numbers. We have already mentioned the possible use of randomly generated amounts of shift in this problem. We might wish to add to this the complication of a random permutation of either the key alphabet or the standard alphabet. Let us go so far as to make even the choice of which alphabet to permute a randomly determined choice. Here we see the need for interpreting a number between 0 and 1 as a *binary switch*, i.e., we want the number to point toward one of two alphabets. If we want to make either alphabet as likely to be chosen as the other, we could say that whenever the number is less than .5, it means the key alphabet; otherwise it means the standard alphabet. If we wished to force the key alphabet to be chosen three times as often, for some reason, we could



(*a*) (*b*)

Figure 5.8

specify that any number less than .75 means the key alphabet, any number greater than or equal to .75 means the standard alphabet. Even the ratio involved here (3:1) could be randomly changed each time. It is obviously not hard to find ways to complicate codes. Unfortunately, some of these methods become time-consuming for human beings. A computer can do this kind of routine work very well, however, and computers are, in fact, being used for such problems.

What about the question raised earlier of a shift of 3.75? In many situations we clearly need randomly generated *integers*. Suppose we wish to obtain integers between 0 and 1000. A simple device, using random numbers between 0 and 1, is to use the greatest integer function which we have already seen:

$$y = [1000x]$$

When $x = 0$, $y = 0$, when $x = 1$, $y = 1000$, so that this transformation does produce integers in the desired range. The graph of this transformation is shown in Figure 5.9. The reader should verify that

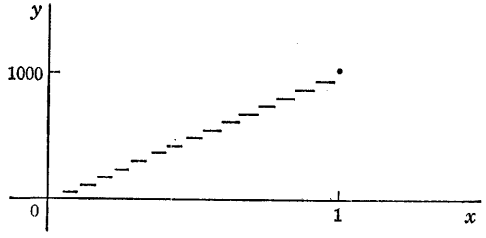


Figure 5.9

the following formula will produce randomly generated *integers* in the range -500 to $+500$.

$$y = [1000x] - 500$$

PROBLEMS

1. In Problem 2, Chapter 4, we wrote SQRT.(Z) for the square root of Z. Let us now write RAND.(0) for a *uniformly distributed* random number between 0 and 1. In other words, every time we execute RAND.(0)

we obtain a new random number. Thus, a program which contains the statements

$$Q = Q + \text{RAND.}(0)$$
$$R = R - \text{RAND.}(0)$$

would expect to have one random number added to Q and a different random number subtracted from R. [You may assume that each time the program is executed from the beginning, the sequence of numbers obtained by using RAND.(0) is the same.] Now write a program which builds some additional complexity into the encoding-decoding program shown in Figure 5.4.

2. Write a section of a program (presumably to be embedded in a larger program) which will determine whether two integers X and Y are congruent modulo 39 or not. Better still, write it to determine whether two numbers are congruent modulo M or not, for any positive integer M. What would happen here if M = 0?

3. At the end of Chapter 3 is an exercise which considers the effect on the computation of an error in the program. Now let us examine the corresponding problem for an error in the flow diagram. What computation would result in each of the following cases:

a. In Figure 5.2*b*, the arrow coming out of the box containing the substitution

$$S \leftarrow S + 5$$

goes into the wrong iteration box; i.e., it enters the I iteration at the point

$$I \leftarrow I + 1$$

instead of the K iteration at the point

$$K \leftarrow K + 1$$

b. In Figure 4.4, each of the arrows marked *True* is incorrectly marked *False*, and each of the arrows marked *False* is incorrectly marked *True*.

c. In Figure 1.1, the arrow from the bottom of the diagram back up into the earlier part of the diagram enters just below the box containing

$$R = 0$$

instead of just above it.

CHAPTER SIX

MONTE CARLO METHODS

6.1 COMPUTING AN AMOUNT OF WORK

IN THE PRECEDING chapter, we noted that there were several methods of generating pseudorandom numbers, and we saw one possible application of random numbers in making the decoding of messages more difficult. Much more common, however, is the use of random numbers to build into an algorithm some definite probability of the occurrence of an event, thus allowing a great deal of very powerful and interesting mathematics to be used. (Methods based on the use of random numbers and probability in this way are called *Monte Carlo methods*, for obvious reasons.) We shall illustrate this by studying in some detail a simple physical problem, e.g., the calculation of the amount of work done in applying a known (but not necessarily constant) force to some object along a path.

We know from elementary physics that the work done in pushing an object a distance S with a constant force F is the product $F \cdot S$. If we plot on a graph the force $f(x)$ as a function of the distance x, we see (Figure 6.1) that the work in this case is represented as the area within the rectangle, since the dimensions of the rectangle are $F \times S$. If the force is not constant, but varies with the distance x,

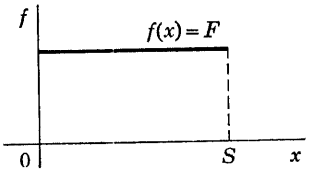


Figure 6.1

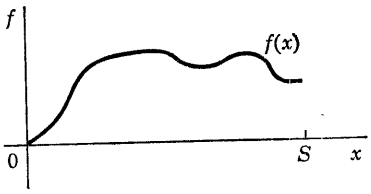


Figure 6.2

we may still plot the force function, but it will not produce a rectangle, i.e., it may look something like Figure 6.2. Here, too, however, the work may be interpreted as being the *area under the curve* $f(x)$. The problem now centers on how one can compute the area, when the function f is known, but perhaps complicated.

One approach, familiar to those who know some calculus, is to approximate the area in question by rectangles or trapezoids. Thus, in Figure 6.3, we see the use of two trapezoids. The area of the trapezoids will be a fair approximation to the area under the curve. If we subdivide the horizontal axis even more finely, as in Figure 6.4, we may even improve the approximation. Since the area of a trapezoid is computed by multiplying the altitude by the average of the bases, we see that the area of the trapezoid from x_2 to x_3, for example, is

$$A_{23} = \tfrac{1}{2}[(f(x_2) + f(x_3))(x_3 - x_2)$$

[Here the altitude is $x_3 - x_2$ and the lengths of the bases are $f(x_2)$ and $f(x_3)$, respectively.] We may express the total area (interpreted as work, in this problem) as

$$A = \tfrac{1}{2}[(f(x_0) + f(x_1))(x_1 - x_0) + \cdots + (f(x_3) + f(x_4))(x_4 - x_3)]$$

This may be simplified somewhat by assuming that the points x_0,

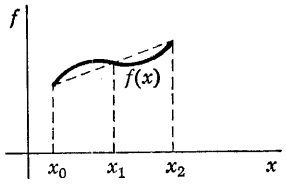


Figure 6.3

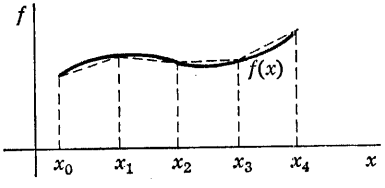


Figure 6.4

$x_1, \ldots, x_4$ are equally spaced, so that $x_{i+1} - x_i = h$ for some fixed h. Then we have

$$\begin{aligned} A &= \tfrac{1}{2}[(f(x_0) + f(x_1))h + \cdots + (f(x_3) + f(x_4))h] \\ &= \tfrac{1}{2}h[(f(x_0) + f(x_1)) + (f(x_1) + f(x_2)) + \cdots + (f(x_3) + f(x_4))] \\ &= \tfrac{1}{2}h[f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + f(x_4)] \end{aligned}$$

In general, if we have $n + 1$ equally spaced points $x_0, \ldots, x_n$, we have

$$A = \tfrac{1}{2}h[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

This is called the *trapezoidal rule*, and if h is small and the function f not too wild, this rule will give quite a good approximation to the area. (There are other formulas for numerical integration[1] which furnish better approximations but which sometimes require more computation. These may be found in any book on numerical analysis.[2])

We shall consider now a quite different method, based on the use of uniformly distributed random numbers (see Chapter 5). Let us construct a rectangle large enough to contain the area we wish to measure. The rectangle can easily be constructed by drawing a horizontal line of height M, where M is some number greater than or equal to the maximum value of the function f. (We shall assume that f has a maximum value.) Then the area which we wish to measure is less than or equal to the area $M \cdot S$ of the rectangle. Now we shall assume that we can obtain two (uniformly distributed) random numbers from the interval from 0 to 1, and by suitable transformations (as outlined in Chapter 5), we may arrange it so the first number will be uniformly distributed over the interval 0 to S and the second over the interval 0 to M. (It is very easy to slip into calling a single number "uniformly distributed." This is incorrect, since this is a property that only *collections of numbers* can enjoy. Although we may occasionally use the phrase "uniformly distributed number,"

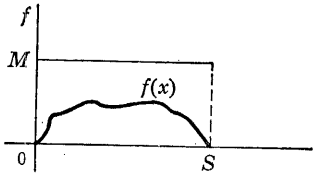


Figure 6.5

[1] Computation of the area under a curve is called *integration* in calculus.

[2] For example, see "Numerical Methods for Science and Engineering," by R. G. Stanton, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1961.

it should always be understood to mean "a number drawn from a uniformly distributed collection." Similar remarks hold for normally distributed collections.) Taking these numbers as pairs of coordinates, then, we now have a uniformly distributed collection of *points of the rectangle.* Using the notation suggested in Problem 1, Chapter 5, we could write statements for these coordinates as follows:

$$\mathrm{X} = \mathrm{S} * \mathrm{RAND.}(0)$$
$$\mathrm{Y} = \mathrm{M} * \mathrm{RAND.}(0)$$

We shall call a point chosen randomly in this way a *success* if it falls under the curve $f(x)$ and a *failure* if it falls outside the area of interest to us. (It must always fall in the rectangle, however.) We then observe that the ratio of successes to the total number of points generated will be approximately the same as the ratio of the desired area to the area of the entire rectangle. In other words, if we try N points and $\mathrm{N_S}$ of these are successes, we have

$$\frac{\mathrm{N_S}}{\mathrm{N}} \approx \frac{\mathrm{A}}{\mathrm{M} \cdot \mathrm{S}}$$

where A is the desired area, and therefore

$$\mathrm{A} \approx \frac{\mathrm{N_S} \cdot \mathrm{M} \cdot \mathrm{S}}{\mathrm{N}}$$

(The symbol "$\approx$" is used instead of the equal sign to remind us that this is an approximation.) It can be shown, using straightforward mathematics, that the approximation gets better and better as the number of points gets very large. It is clear now why computers can be used to advantage here and, in fact, why such methods were rarely used before the advent of the computer.

When we plan to write a program for some particular force function, say $f(x) = 6x^3 + 5x^2 + x + 7$, we should first observe that there is an easy way to test whether or not a point (x_0,y_0) is in the area under the curve. We need test only whether or not y_0 is less than the height $f(x_0)$ of the curve at x_0, i.e., whether $y_0 \leq f(x_0) = 6x_0^3 + 5x_0^2 + x_0 + 7$. Figure 6.6 exhibits a program for the problem we have been considering. Here we have written NS for $\mathrm{N_S}$, the number of successes. In the loop, which starts with I = 1 and increments I by 1 each time until I exceeds N, random coordinates

X and Y are computed, scaled, and tested against the values of the function. Whenever a success occurs [i.e., $Y \leq f(X)$], NS is increased by 1; whenever there is a failure, NS is not increased, and the loop is completed. After N such tests, the iteration ends, and the area A is computed.

```
      NS = 0
      THROUGH LOOP, FOR I = 1, 1, I .G. N
      X = S * RAND.(0)
      Y = M * RAND.(0)
LOOP  WHENEVER Y .LE. ((6. * X + 5.) * X + 1.) * X + 7., NS = NS + 1
      A = NS * M * S/N
      INTEGER NS, N, I
      END OF PROGRAM
```

Figure 6.6

Note that in the statement labeled LOOP we wrote the polynomial $6x^3 + 5x^2 + x + 7$ in a form called *nested multiplication.* In this form both computation time and accuracy are improved by using fewer multiplications.

6.2 EXTERNAL FUNCTIONS

Now that we have written a program which computes the area under the curve representing the particular function $f(x) = 6x^3 + 5x^2 + x + 7$, one might very well ask whether the program could be easily modified to handle some other function, such as $g(x) = (3x^3 - 4)/(4x^5 + 1)$. This could be done very easily by changing the formula for the value of the function in the statement labeled LOOP. An even better question, however, would be whether the program could be written in some general form so as to handle *any* function on any interval, given only the "name" (or other description) of the function. We might find it very convenient, for example, to say something like this: "Compute the area under the curve $f(x)$, on the interval from S_1 to S_2, using N trials, assuming that the maximum value of $f(x)$ on the interval $[S_1,S_2]$ does not exceed M." The explicit form of the function f would be given somewhere else.

Let us assign a name to the general integration program we are

considering. Again, as in Chapter 2, we need some conventions on the types of names we may choose for *programs*. We might just as well use the same rules that we have for variables, i.e., up to six capital letters or digits, the first of which must be a letter. There ought to be a way to tell program names from variable names, however, since they are really different kinds of objects. Let us follow the name of a program by a period. (This is quite natural, since many of the names of programs will be abbreviations, anyway.) We have already seen examples of such names as SQRT. and RAND., which were suggested with these conventions in mind. Although we usually think of the square root as a *function* which assigns to each nonnegative number its (principal) square root, we are suggesting here that we can regard it also as a *program*. This is what actually happens, anyway, since we use a program to compute a square root for us each time we need one. There is no harm, therefore, in referring to SQRT. as either a *function* or a *program*. Each of these names implies that, when presented with a value for its argument, SQRT. will come up with a second value, i.e., the square root of the argument (see footnote on page 5).

We shall find it very useful to distinguish carefully between the name of a program and the value (or values) that may be computed by the program. In the same way, the mathematician is very careful to refer to the *function* f as being different from the *value* $f(x)$. Now that we have names of functions, such as SQRT., how shall we refer to their values? In each case let us write the name, followed by the list of *parameters* (i.e., those variables, function names, etc., which may change from one computation to another) just as we write the arguments of ordinary functions in mathematics. Thus, we have already seen the use of SQRT.(Z) to indicate the (principal) value of the square root of Z. What we really mean here is the value computed by a special program which accepts a parameter Z and returns the value of the square root of Z.

Returning to our integration program, then, we need a name for the program, say INTEG., and a list of parameters. The obvious parameters are S_1, S_2, N, M, and the name f of the function which defines the area. Thus we might very well write

AREA = INTEG.(0,10,1000,4.,SQRT.)

if we wish to compute the area under the curve of the square-root function over the interval [0,10]. This also specifies that 1000 trials should be used and that the maximum value of the SQRT. function is not expected to exceed 4. Note that we are saying here that the *value* of the function INTEG. (i.e., the value computed by the program called INTEG.) should be stored as the new value of AREA. We may even find it convenient to use values of functions in more complicated expressions, such as

$$X = (-B + SQRT.(B * B - 4. * A * C))/(2. * A)$$

which would occur in the solution of a quadratic equation.

We have seen that in order to compute the value of a function, such as SQRT., we generally need a program. This program, in fact, *defines* the function by providing the rule by which the values are determined, and we will call it the *definition program.* Each computation in some other program which involves the value of a function is referred to as a *call* for that function. Thus, the quadratic-equation example just above illustrates a call for the square-root function. A call for a function must occur in some program other than the program which defines the function, and one should be careful to distinguish between the *calling program* and the *definition program.*

The next question is: How does a program become a definition program? In order to answer this question, let us consider making the program in Figure 6.6 a definition program for the function INTEG., i.e., the program which could be called upon whenever we needed to obtain a value of INTEG. . We must first determine the properties that a definition program must have. (1) It must have a name. (In this case, INTEG. has been chosen to be the name.) (2) It must have a list of parameters, i.e., for each *call* we must be able to specify which values or function names we wish to use as arguments for that call. (We have here S_1, S_2, N, M, and *f*.) (3) We must have a definite value, computed by the program, which can be designated as *the value of the function.* (In Figure 6.6, the value is A.)

Since this program will be separate from any program which calls it, we should expect to put into it such declarations as INTEGER,

etc., as usual. We will need to identify the program as a function definition program, however; so let us put as the first line the following statement.

EXTERNAL FUNCTION (S1,S2,N,M,F.)

The word *external* implies that this program is to be completely independent of any calling program. (We shall consider *internal* functions later.) In parentheses, we have listed the arguments to the function. Whenever another program calls on this program for a value of INTEG., the calling program will use a specific set of values as arguments. As we saw above, a call might be

AREA = INTEG. (0,10,1000,4.,SQRT.)

In writing the definition program, however, we shall use the names S1, S2, N, M, and F., with the understanding that these names will be replaced by the appropriate values or names for each call. Figure 6.7 shows the program in Figure 6.6 converted to the form of an external-function definition and modified slightly to use the general interval S_1 to S_2 (see Section 5.5). Note that we have changed the END OF PROGRAM statement to END OF FUNCTION to be consistent. Also, note that we have clearly designated the value to be returned to the calling program by the use of a FUNCTION RETURN statement. The one remaining property that this definition program needs to have explicitly stated is its name. This is supplied by the ENTRY TO statement.

```
      EXTERNAL FUNCTION (S1,S2,N,M,F.)
      ENTRY TO INTEG.
      NS = 0
      THROUGH LOOP, FOR I = 1, 1, I .G. N
      X = (S2 - S1) * RAND.(0) + S1
      Y = M * RAND.(0)
LOOP  WHENEVER Y .LE. F.(X), NS = NS + 1
      FUNCTION RETURN NS * M * (S2 - S1)/N
      INTEGER NS, N, I
      END OF FUNCTION
```

Figure 6.7

Table 6.1

I	NS	NS/I
100	32	.320
200	63	.315
300	92	.307
400	119	.297
500	154	.308
600	191	.318
700	220	.314
800	253	.316
900	286	.318
1000	319	.319
1100	347	.315
1200	379	.316
1300	425	.327
1400	450	.321
1500	486	.324
1600	515	.322
1700	548	.322
1800	576	.320
1900	614	.323
2000	651	.325
2100	682	.325
2200	719	.327
2300	751	.327
2400	786	.327
2500	816	.326
2600	848	.326
2700	878	.325
2800	908	.324
2900	934	.322
3000	960	.320
3100	997	.322
3200	1029	.322
3300	1068	.324
3400	1091	.321
3500	1126	.322
3600	1162	.323
3700	1209	.327
3800	1232	.324
3900	1262	.324
4000	1303	.326

Table 6.1. (*Continued*)

I	NS	NS/I
4100	1342	.327
4200	1376	.328
4300	1416	.329
4400	1451	.330
4500	1486	.330
4600	1524	.331
4700	1552	.330
4800	1580	.329
4900	1618	.330
5000	1648	.330
5100	1679	.329
5200	1722	.331
5300	1760	.332
5400	1785	.331
5500	1818	.331
5600	1851	.331
5700	1882	.330
5800	1918	.331
5900	1952	.331
6000	1993	.332
6100	2031	.333
6200	2061	.332
6300	2092	.332
6400	2128	.332
6500	2153	.331
6600	2186	.331
6700	2214	.330
6800	2245	.330
6900	2278	.330
7000	2309	.330
7100	2335	.329
7200	2376	.330
7300	2403	.329
7400	2433	.329
7500	2469	.329
7600	2503	.329
7700	2535	.329
7800	2570	.329
7900	2612	.331
8000	2649	.331

Table 6.1. (*Continued*)

I	NS	NS/I
8100	2672	.330
8200	2702	.330
8300	2739	.330
8400	2773	.330
8500	2812	.331
8600	2846	.331
8700	2886	.332
8800	2923	.332
8900	2964	.333
9000	3004	.334
9100	3022	.332
9200	3047	.331
9300	3078	.331
9400	3107	.331
9500	3142	.331
9600	3173	.331
9700	3198	.330
9800	3224	.329
9900	3257	.329
10000	3290	.329

As an illustration of the behavior of this external function, an actual run was made on the IBM 704 computer. In this case, the following values were used: S1 = 0, S2 = 1, N = 10,000, M = 1, and F.(X) = X .P. 2. This represents the area under the curve of the function $f(x) = x^2$ over the interval [0,1], using a rectangle of height 1 to contain the area. The true value of $\frac{1}{3}$ for the magnitude of this area is easily obtained using elementary calculus. The area and the rectangle are shown in Figure 6.8. The computation took 50 seconds on this moderately fast computer, and values of I, NS, and the approximation to the area NS/I were printed out after each 100 trials. The table of values (Table 6.1) shows the pattern of the approximation.

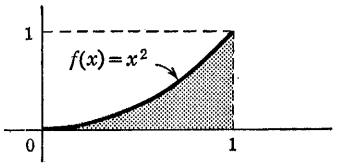


Figure 6.8

6.3 RANDOM-NUMBER GENERATORS

We have been using the name RAND. for some time now to represent a random-number generating function. It might be of interest to digress here and see how an external-function program bearing the name RAND. might be written.

There are many ways to generate numbers which behave quite randomly. One of the early methods for generating uniformly distributed numbers (called the *center-squaring method*) was to start with a large number, such as 3^{15}, and each time another number was needed the current number would be squared and one-fourth of the digits in the result would be deleted from each end, leaving a number with the same number of digits as the original one. This number would be the desired random number as well as the starting number for the generation of the next number. The assumption here (supported by quite rigorous statistical tests) is that any digit is as likely to result in this way as any other, and therefore the numbers obtained in this way will be uniformly distributed. To illustrate this method with manageable numbers, let us start with 2^6 (i.e., 64) instead of 3^{15}. To obtain the next number, we form $(64)^2 = 4096$ and drop one digit off each end to obtain 09 as the next number. The next number after that would be 08 (from 0081), and then in turn, 06, 03, 00. Unfortunately, at this point the sequence degenerates into a succession of zeros, which is a very nonrandom sequence. Although this method was used on many computers, its tendency to degenerate into short cycles in which the same number (or sequence of numbers) is repeated caused it to fall into disrepute. It was then necessary to find ways to obtain uniformly distributed numbers by methods which did not give rise to short cycles.

One class of methods which has been investigated and is gaining in popularity is the class of *Fibonacci methods*, which are based on addition procedures rather than the multiplicative kind illustrated by the center-squaring method above. A simple example of such a method[1] employs a sequence of numbers called the *Fibonacci numbers*. This sequence starts with the numbers 1, 1, 2, 3, 5, 8, 13 and continues indefinitely according to the rule that each number (after the

[1] "Fibonacci Series Modulo m," by D. D. Wall, *Amer. Math. Monthly*, vol. 67 (1960), pp. 525–532.

first two in the sequence) is the sum of the two numbers which precede it. Thus, a general formula for obtaining Fibonacci numbers is the following:

$$u_1 = 1$$
$$u_2 = 1$$
$$u_i = u_{i-1} + u_{i-2} \qquad \text{for } i \geq 3$$

This sequence of numbers appears to have some amazing properties in that many natural phenomena seem to be organized according to the relationships which exist between successive Fibonacci numbers. For example, the spirals along the exterior of many seashells follow this sequence in their spacing. Other examples of such phenomena are often found at science fairs and in mathematical journals.[1] The use of Fibonacci numbers for random-number generation is based on the apparent random behavior of the right-most digits when the numbers get large enough. What makes these numbers very attractive to computer users is that the generation of each number involves a simple addition of the two numbers most recently generated. Moreover, one need store only these two numbers in order to be able to continue. It may be that this method is not so popular as the next method to be described simply because the extent to which the numbers thus generated behave randomly has not been well enough established.

Before going on to the next method, we should examine how one obtains the "right-most" digits of a large number, as in the Fibonacci method just described. Since most present-day computers (but not all) represent numbers internally as *words* (i.e., sequences of digits) which have a fixed number of digits—such as 10 decimal digits with a plus or minus sign—the usual method is to start with a number large enough to fill all these digits. Thus, in the Fibonacci sequence, for example, one would start far enough along the sequence so that the numbers have enough digits. Then, each time an addition is performed, one simply ignores any carries generated off the left end. Using a two-digit decimal word as an example, we might start with the Fibonacci sequence at 13. Then, ignoring carries off the left

[1] Leonardo Pisano (Fibonacci) in 1202 used this series in a problem on the number of offspring of a pair of rabbits. See L. E. Dickson, "History of the Theory of Numbers," vol. 1, p. 393, Washington D.C., 1919.

end of our two-digit word, we would obtain the numbers 13, 21, 34, 55, 89, 44, 33, 77, 10, and so on. Again we obtain a highly non-random sequence after a while, but we would expect that this would not happen with very large numbers.

There is a convenient mathematical formulation of the use of the right-most digits. Another way to describe what we are doing in the two-digit example above is to say that we are dividing the result by 100 and using the remainder. In other words, using the congruence relation introduced in Chapter 5, we may describe the sequence of random numbers for the two-digit machine as follows:

$$\begin{aligned} r_1 &= 13 \\ r_2 &= 21 \\ r_i &\equiv (r_{i-1} + r_{i-2}) \bmod 10^2 \qquad \text{for } i \geq 3 \end{aligned}$$

where it will be understood that r_i should be chosen to be the *representative* of the congruence class containing $r_{i-1} + r_{i-2}$. In other words, r_i is that number between 0 and 99 which is congruent to $r_{i-1} + r_{i-2}$. The number r_i is sometimes called the *residue* of $(r_{i-1} + r_{i-2}) \bmod 10^2$.

A simple extension of this reasoning leads to the following formulas for the 10-digit and n-digit machines, respectively, where u_j is far enough along the Fibonacci sequence to fill the word:

$$\begin{aligned} r_1 &= u_j \\ r_2 &= u_{j+1} \\ r_i &\equiv (r_{i-1} + r_{i-2}) \bmod 10^{10} \qquad \text{for } i \geq 3 \end{aligned}$$

$$\begin{aligned} r_1 &= u_j \\ r_2 &= u_{j+1} \\ r_i &\equiv (r_{i-1} + r_{i-2}) \bmod 10^n \qquad \text{for } i \geq 3 \end{aligned}$$

We should also point out that not every computer represents numbers internally in decimal form. Many of the larger machines use a *binary* representation, i.e., numbers are represented as strings of ones and zeros, using 2 as the base of the number system instead of 10. Thus, the number 39 (decimal) would have the binary representation 100111, since $39 = \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{1} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0$. Note that each digit of a binary number contains much less information about the size of the number than a decimal digit; so we would

expect to need more digits in the binary representation. In fact, a 10-digit decimal number needs a 34-digit binary representation. Machines with binary internal representations usually have words containing 36 to 64 binary digits (sometimes called *bits*). A random-number scheme based on Fibonacci numbers for a binary computer with 36-bit words would then start with some u_j large enough to fill the word and use the following scheme:

$$r_1 = u_j$$
$$r_2 = u_{j+1}$$
$$r_i \equiv (r_{i-1} + r_{i-2}) \bmod 2^{36} \qquad \text{for } i \geq 3$$

It was indicated above that there is yet another method of generating uniformly distributed random numbers which is far more popular than any of the procedures described so far. Now that we have developed most of the ideas and notation which we need, the description of this third method (called the *power-residue* method) is quite straightforward. One starts with a number r_1 large enough to fill up the computer word and a carefully selected number P. Then we use the formula

$$r_i = \mathrm{P}r_{i-1} \bmod m^n \qquad \text{for } i \geq 2$$

where m is the base of the internal number system (2 for binary, 10 for decimal, etc.) and n is again the number of digits in the computer word. The effect of this formula is to start with an initial number r_1 and repeatedly multiply it by P, each time retaining only the rightmost n digits. Illustrating again with a two-digit decimal machine, we may start with $r_1 = 37$ and $\mathrm{P} = 71$. Then $r_2 \equiv \mathrm{P}r_1 \bmod 10^2$, so that $r_2 \equiv (71)(37) \bmod 10^2$. Then $r_2 \equiv 2627 \bmod 10^2$, and finally $r_2 = 27$. In this way, we generate the following sequence: 37, 27, 17, 07, 97, 87, 77, 67, 57, 47, 37, 27, etc. Note that not only is this sequence nonrandom, but it begins to repeat itself after a short time. The sequences 64, 09, 08, 06, 03, 00, 00, . . . and 13, 21, 34, 55, 89, 44, 33, 77, 10, . . . , which we generated while discussing the center-squaring and Fibonacci methods, respectively, actually repeat, also, if carried far enough. ("Repeat" means that although some initial terms may not occur again, there will be a block of numbers that repeats indefinitely. The size of the block is called the *period* of the sequence.)

It is important to observe that in a two-digit decimal word, it is possible to represent only 10^2 different numbers. (In general, then, one can represent only m^n different numbers.) As soon as one of these numbers appears for a second time (such as 37 above), the number which follows its first occurrence (27, in the example above) must follow again, thus starting the repetition. One must therefore try to choose r_1 and P very well, so as to make the period (which cannot exceed m^n) as close to m^n as possible. Using some very interesting number theory, mathematicians such as M. L. Juncosa[1] have shown that r_1 may be any number not divisible by m and that P must be determined separately for each m and n. For example, for $m = 2$ and $n = 36$, one of the best choices is $P = 5^{15}$ and $r_1 = 2^{35} - 1$. With this choice, the period of the generated sequence is 2^{33}. This means that one can generate 2^{33} (i.e., 8,589,934,592) numbers in this way without repeating.

Now let us turn for a short time to the *normally* distributed numbers. As we indicated earlier, such a distribution has a graph such as in Figure 6.9. The point on the x axis labeled $\bar{x}$ (pronounced "x bar") represents the average of the values in the graph and is called

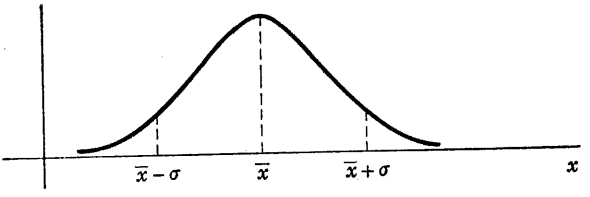


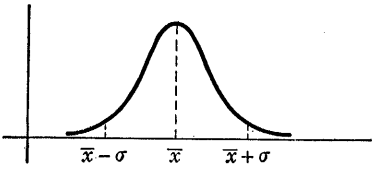
Figure 6.9

the *mean*. The points labeled $\bar{x} + \sigma$ and $\bar{x} - \sigma$ are each at a distance σ from the mean $\bar{x}$ and indicate the spread of the curve. This measure σ of the spread is called the *standard deviation* and is traditionally represented by the Greek letter *sigma*. If σ is small, we obtain a narrow graph, such as in Figure 6.10*a*; if σ is large, we obtain a

[1] Random Number Generation on the BRL High-Speed Computing Machines, *Rept.* 855, Ballistics Research Laboratory, Aberdeen, Md. See also, Random Number Generation and Testing, IBM *Ref. Manual* C20-8011; and "Microanalysis of Socioeconomic Systems," p. 356, by G. H. Orcutt, M. Greenberger, J. Korbel, and A. M. Rivlen, Harper & Brothers, New York, 1961.

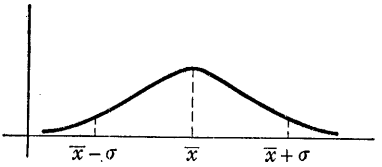
graph such as in Figure 6.10*b*. A complete discussion of the standard deviation can be found in any book on probability.[1] (Also, see Problem 6 at the end of this chapter.)

Thus, if one were to ask for a program to produce normally distributed random numbers, it would be a desirable feature to be able to specify in the call the shape of the graph, i.e., the desired mean and standard deviation. Let us therefore revise the method of calling on our external function RAND. so that a call with *two* arguments,



(*a*)

Figure 6.10*a*



(*b*)

Figure 6.10*b*

the first of which is zero [i.e., RAND.(0,R1)], will still produce a *uniformly* distributed number between 0 and 1, and a call with *four* arguments, of which the first is 1 [i.e., RAND.(1,R1,XBAR, SIGMA)], will produce a random number drawn from a collection with a *normal* distribution having mean XBAR and standard deviation SIGMA. (The names XBAR and SIGMA are used because the characters $\bar{x}$ and σ are not part of the available alphabet in our language.) The argument R1 will be needed to produce the uniformly distributed number needed in either case, as we will see.

One very convenient way to obtain normally distributed numbers with specified $\bar{x}$ and σ is to generate a collection of uniformly distributed numbers by one of the methods described above and transform it into a collection that represents a normal distribution. One such transformation is the following, where r represents a number in the uniformly distributed collection:[2]

$$N = \bar{x} + \sigma \left\{ \text{sign}(r - .5) \left[v - \frac{a_0 + a_1 v + a_2 v^2}{1 + b_1 v + b_2 v^2 + b_3 v^3} \right] \right\}$$

[1] See, for example, "Probability: An Introduction," by Samuel Goldberg, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1960.

[2] Juncosa, *op. cit.*

where

$$v = \sqrt{-2 \log_e .5(1 - |1 - 2r|)}$$

$$\text{sign}(t) = +1 \quad \text{if } t \geq 0 \quad \text{and} \quad \text{sign}(t) = -1 \quad \text{if } t < 0$$

and

$$a_0 = 2.515517 \quad a_1 = .802853 \quad a_2 = .010328$$
$$b_1 = 1.432788 \quad b_2 = .189269 \quad b_3 = .001308$$

6.4 INTERNAL FUNCTIONS

Let us denote the first argument for RAND. by the name DIST, since it determines which type of distribution is needed. Then a rough flow diagram for RAND. might be written as in Figure 6.11.

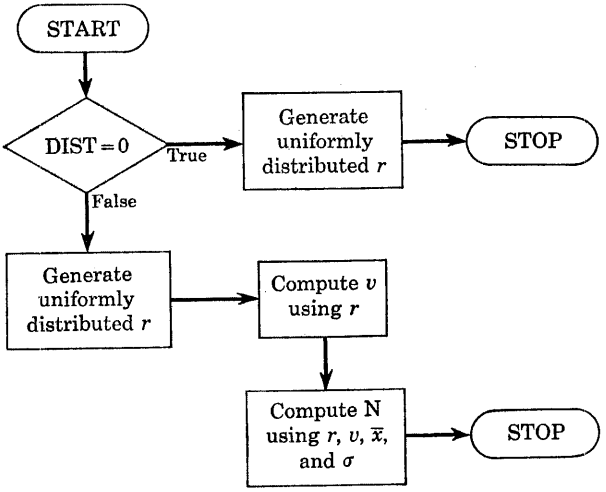


Figure 6.11

From this diagram it is clear that the part which generates the uniformly distributed number r either must be included twice, or it must be something like an external-function definition program, i.e., a separate program which can be called on more than once. It would be very convenient to be able to write it as an external-function definition, but still keep it *within the program* which will call on it.

One benefit which could arise from keeping such a function definition internal to the calling program is that only those variables which are expected to change from one call to another need be specified as arguments. All other variables would be known simultaneously to the calling program and the internal-function definition program. This is not the case with the external-function definitions which we considered earlier, since these are entirely separate programs which can communicate with the calling program *only via the argument list.*

Let us agree, then, to allow a function definition program to occur within a calling program. We shall write such a function definition program in exactly the same way as an external-function definition program, except that the word EXTERNAL in the first line shall be replaced by the word INTERNAL (see Figure 6.7). It will be agreed, also, that any variables not explicitly listed as arguments in the internal-function definition program will be available to both the calling program and the function definition program. For reasons which would lead us too far afield, let us agree that internal-function definitions may occur anywhere in the calling program except within other internal-function definitions. *Calls* for internal functions or external functions may occur anywhere.

There is one question remaining with regard to internal functions. In the preceding paragraph we said that variables not appearing as arguments in the definition program would be common to the definition program and the calling program. What about a variable which occurs in the calling program and also as an argument in the definition program? Is there (or should there be) any connection between the occurrence of a name in the calling program and its use as an argument in the definition program? Because of the possibility of conflicting modes (i.e., one integer, the other noninteger), it is better to specify that names occurring *as arguments in internal-function definitions* should not occur elsewhere in the calling program (i.e., outside the internal-function definition).

Figure 6.12 exhibits a complete program for RAND. as we have discussed it. Let us proceed through the program one statement at a time, bypassing remarks, which are recognized by the letter R at the left. The first two lines declare this to be an external function with four arguments, the first two of which are of integer mode. We then find the definitions of three internal functions SIGN.,

```
   EXTERNAL FUNCTION (DIST,R1,XBAR,SIGMA)
   INTEGER DIST, R1
R      THE NEXT 8 STATEMENTS DEFINE THE FUNCTION SIGN.
   INTERNAL FUNCTION (X)
   ENTRY TO SIGN.
   WHENEVER X .GE. 0.
      FUNCTION RETURN 1.
   OTHERWISE
      FUNCTION RETURN -1.
   END OF CONDITIONAL
   END OF FUNCTION
R      THE NEXT STATEMENT DEFINES THE FUNCTION MODULO.
   INTERNAL FUNCTION MODULO.(Y,Z) = Y - (Y/Z) * Z
   INTEGER Y, Z, MODULO.
R      THE NEXT 5 STATEMENTS DEFINE THE FUNCTION UNIF.
R      WHICH GENERATES UNIFORMLY DISTRIBUTED NUMBERS.
   INTERNAL FUNCTION
   ENTRY TO UNIF.
   R1 = MODULO.(5 .P. 15 * R1, 2 .P. 35)
   FUNCTION RETURN R1/(2. .P. 35)
   END OF FUNCTION
R      THE NEXT STATEMENT IS THE ACTUAL ENTRY
R      TO THE EXTERNAL FUNCTION RAND.
   ENTRY TO RAND.
R      THE NEXT STATEMENT RETURNS A UNIFORMLY
R      DISTRIBUTED NUMBER IF DIST IS ZERO BY
R      CALLING ON THE INTERNAL FUNCTION UNIF. DEFINED ABOVE.
   WHENEVER DIST .E. 0, FUNCTION RETURN UNIF.(0)
R      THE NEXT STATEMENTS ARE EXECUTED IF
R      DIST IS NOT ZERO.  THEY CALL ON UNIF.
R      FOR A UNIFORMLY DISTRIBUTED NUMBER AND
R      TRANSFORM IT INTO A NORMALLY DISTRIBUTED
R      NUMBER, WHICH IS RETURNED AS THE VALUE OF RAND.
   R = UNIF. (0)
   V = SQRT.(-2. * ELOG.(.5 * (1. - .ABS.(1. - 2. * R))))
   FUNCTION RETURN XBAR + SIGMA * (SIGN.(R - .5)*
1        (V - ((.010328 * V + .802853) * V + 2.515517)/
2     (((.001308 * V + .189269) * V + 1.432788) * V + 1.)))
   END OF FUNCTION
```

Figure 6.12

MODULO., and UNIF. which are called upon for values at various times in the program. The function SIGN. has the value +1. if its argument is nonnegative and −1. otherwise. (Except for the fact that the argument might be zero, we could have defined SIGN. by the value of X/.ABS. X. The definition used here does not need to make a special case of a zero argument.)

The single statement which defines MODULO. illustrates a short form of definition program which we can use for simple functions which can be defined by a single expression (see Section 6.5). It is also introduced by the words INTERNAL FUNCTION, as are all internal-function definitions. The function MODULO. computes the integer Y modulo Z (i.e., the residue, or the remainder on dividing Y by Z) by subtracting from Y the largest multiple of Z less than or equal to Y. Thus, MODULO.(17,5) = 17 − (17/5) * 5 = 17 − 3 * 5 = 2. (Note that we are implicitly assuming here that Z will never be zero and that the division used is the truncated integer division.)

The next five statements define the uniformly distributed random-number generator UNIF., which is based on the power-residue method described above for generating uniformly distributed numbers. This internal function uses the current value of the argument R1, multiplies it by $P = 5^{15}$ (we are assuming here a 36-bit binary computer), and reduces the product modulo 2^{35}. This reduction is done by calling on MODULO. (with $Z = 2^{35}$) according to the formula

$$r_i \equiv Pr_{i-1} \bmod m^n \qquad i \geq 2$$

which was introduced earlier. (Since the new value is stored again in R1, it is already in position for a subsequent call on RAND. .) The new value of R1 is scaled down to a number between 0 and 1 and is simultaneously converted to noninteger form by dividing it by 2^{35} (in noninteger form because of the period after the 2). Noninteger form is necessary for the value returned as the value of the function because it is between 0 and 1, and it therefore cannot be an integer. Note that R1 does not need to be an argument in the call for UNIF., since it is available to both the calling program and the *internal function* UNIF. . If UNIF. had been an *external function*, we would have been forced to make R1 an argument.

The actual entry to RAND. occurs after the function definition programs. (The definition programs could have been placed at the end just as well.) If DIST is 0, the function UNIF. is called upon for a value, and this is returned as the (uniformly distributed) value of RAND. . (Note that we use an argument of 0 even though UNIF. does not need an argument, since this is the way we indicate that we mean the *value* and not the *name* of the function UNIF. .) If DIST is not zero, however, we call upon UNIF. for a uniformly distributed number, but we then transform it as indicated above into a normally distributed number. Note that in the computation of V we call upon two *external* functions SQRT. and ELOG., presumably available from some standard collection of function definition programs.

6.5 THE ONE-LINE INTERNAL FUNCTION

In the preceding section we had an occasion to use a short form of the internal-function definition, e.g., in the definition of the function MODULO. . As another illustration of the use of this form of internal-function definition, let us return to the social security tax calculation which was discussed in Chapter 4. We saw there a single expression which could be used to calculate the tax, e.g.,

$$\text{TAX} = \text{RATE} * (\text{S} + \text{T} - \text{W} - .\text{ABS.}(\text{S} + \text{W} - \text{T}) + .\text{ABS.}(\text{S} + \text{T} - \text{W} - .\text{ABS.}(\text{S} + \text{W} - \text{T})))/4$$

where S = SALARY (earned this week), W = WAGES (accumulated up to but not including this week), T = THRESH (the threshold amount), and .ABS. denotes the absolute-value operator. In order to see where this expression comes from, let us introduce a rather useful function EXCESS.(X,Y), which has the value $X - Y$ if $X \geq Y$ and the value 0 if $X < Y$. In other words, it represents the excess of X over Y, if there is an excess. It is possible to give an expression whose value is EXCESS.(X,Y) as follows:

$$\text{EXCESS.(X,Y)} = \frac{\text{X} - \text{Y} + .\text{ABS.}(\text{X} - \text{Y})}{2}$$

The excess of S + W over T, if there is any [i.e., EXCESS.(S +

W,T)], represents that part of the total wages S + W (including this week's salary) that is not to be taxed. If we compute the excess, if any, of this week's salary S over the amount just mentioned as the amount not to be taxed [i.e., EXCESS.(S,EXCESS.(S + W,T))], we obtain the amount of S which should be taxed. In other words, suppose $S + W \leq T$, so that total wages to date, including this week, do not yet exceed the threshold. Then the amount to be taxed should be all of S. Using the procedure just described, we would find that the excess of S + W over T is zero, and the excess of S over this zero amount is S. The reader should analyze each of the cases which might arise, such as $W < T$ and $W + S > T$ (i.e., the case in which this week's salary carries the total wages across the threshold), and so on, to understand this method of computing the tax.

Using the one-line definition program to define EXCESS., we have the following program for the tax:

```
INTERNAL FUNCTION EXCESS.(X,Y) = (X - Y + .ABS.(X - Y))/2
TAX = RATE * EXCESS.(S,EXCESS.(S + W,T))
END OF FUNCTION
```

The complicated single line for the computation of the tax was obtained by substituting for the two occurrences of EXCESS. the expression used to define EXCESS. as an internal function. In other words, we have

$$\text{EXCESS.}(S + W,T) = \frac{S + W - T + \text{.ABS.}(S + W - T)}{2}$$

and so on.

PROBLEMS

1. Write an external-function definition program called FIBNO. which will produce as its value the next Fibonacci number each time it is called. The arguments should be U1 and U2, the two most recent Fibonacci numbers generated. On the return from the function, U2 and U1 should be updated, i.e., they should have as values the new number and the previous

value of U2, respectively. (Since these numbers will become quite large, provision should be made for reducing the new value modulo 2^{35}.)

2. Write an external-function definition program called FIB. which will contain the program produced in Problem 1 (converted to an internal-function definition) and which will produce either uniformly or normally distributed random numbers, just as RAND. does.

3. Write a program which will call on FIB. (from Problem 2) for a sequence of 10,000 uniformly distributed numbers, and examine this sequence (as described below) to see if it is really uniformly distributed. Let us imagine the interval from 0 to 1 to be divided into 100 equal parts by the numbers .00, .01, .02, . . . , .99. If we generate a large collection of uniformly distributed random numbers, then we expect that these numbers would fall into each of the smaller intervals with approximately the same frequency. The strategy should be to set up 100 locations V(0), . . . , V(99) in which counts are kept of the number of times a random number falls in the corresponding interval. To do this quite simply, you should set up a vector V of dimension 99, i.e., V(0), . . . , V(99). Then, for each random number generated, separate the left-most two digits and store them as an integer I between 0 and 99. Using this integer I as a subscript for V, increase the count associated with that integer [V(I) = V(I) + 1]. The final values of V(0), . . . , V(99) will then represent the number of occurrences of the integers 0 to 99 and should be approximately equal. To separate the left-most two digits, multiply the number which is between 0 and 1 by 100, then simply store the result in I. (Storing in an integer location will truncate the number down to its integer part.) Thus, if one generates the number .14329467, multiplication by 100 yields 14.329467, and storing it as an integer I truncates it to 14. Then V(14) would be increased by 1 to record an occurrence of 14.

Hint: The statement which generates the number, multiplies it by 100, and truncates it, all at once, is

```
I = FIB.(0) * 100.
```

Of course, I must be declared to be of integer mode.

4. In the remark following Figure 6.6 it was stated that there were fewer multiplications if we wrote $((6x + 5)x + 1)x + 7$ instead of $6x^3 + 5x^2 + x + 7$. Verify that there are fewer multiplications but the same number of additions and that the values of the function (except for roundoff) will be the same no matter how we compute them.

5. By applying the algebraic formulation of EXCESS. to the expression

```
RATE * EXCESS.(S,EXCESS.(S + W,T))
```

derive the longer expression for TAX which appeared in Section 4.1:

```
TAX = RATE * (S + T - W - .ABS.(S + W - T)
                + .ABS.(S + T - W - .ABS.(S + W - T)))/4
```

6. In our discussion of normally distributed random numbers, we mentioned the *mean* $\bar{x}$ and the *standard deviation* σ of the set of numbers. We can actually calculate a mean and a standard deviation for any collection of numbers (or measurements, such as men's heights, shown in Figure 5.6). The formulas used in statistics for $\bar{x}$ and σ are

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

The symbol Σ is used to denote *summation*, so that $\sum_{i=1}^{n} x_i = x_1 + x_2 + x_3 + \cdots + x_n$ and $\sum_{i=1}^{n} (x_i - \bar{x})^2 = (x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2$.

Note that the formula for $\bar{x}$ shows that the mean is just the average of the numbers $x_1, \ldots, x_n$. In a normal distribution the mean happens to occur at the highest point in the graph. Write a flow diagram and a program to compute $\bar{x}$ and σ, given n and $x_1, \ldots, x_n$.

7. Using the notation of Problem 6, show that

$$\sum_{i=1}^{n} (x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - \frac{1}{n} \left(\sum_{i=1}^{n} x_i \right)^2$$

and use this result to simplify the flow diagram and program which you produced for Problem 6.

CHAPTER SEVEN

A SORTING PROBLEM

7.1 THE ALGORITHM

IN HANDLING SETS of numbers, letters, or symbols, we very often find that we have to arrange them in some particular order. This is very desirable if we have to search through a set of numbers or letters many times, since we can then make use of efficient searching procedures. For example, since we often have a name and need the corresponding telephone number, telephone directories are sorted into alphabetical order by name. Occasionally, however, it is very important to find out which name corresponds to a particular telephone number, and in some cities the telephone company maintains its version of the telephone directory sorted by number. In some types of problems it is found to be worth the effort to sort the same set of data several different ways and keep all the sorted lists around during the computation. Another area where sorting becomes extremely important is the processing of payroll or inventory files by commercial firms. The people who maintain these files must be able to make thousands of corrections every day, because of address changes, new tax rates, and changes in the number of dependents, in the case of the payroll. For large inventory files, each sale of each

item must be recorded, incoming shipments added to the amounts on hand, and so on. If the changes were to be made in the order in which they arrive at the computer, even the fastest computer could not find its way through such large files of information fast enough to handle the changes. If the changes for a given day are sorted into the same order as the information in the file (such as by employee number or part number), then the computer can work its way through the file once, always having the proper information as to where the next change is to be made. We actually do a lot of sorting in our day-to-day living, also. Arranging the volumes of an encyclopedia on a shelf, the cards of a bridge hand, letters in a file (by date or by author), bills to be paid, and so on, all require sorting which we do manually. Most of the methods we use in these situations would break down completely if we needed to sort 40,000 items instead of a dozen, but the changes to a large inventory file might easily number 40,000 in a single day.

There are many algorithms for efficient sorting methods. Some use very little storage space in the computer while sorting, but are rather slow. Others are very fast, but use more storage space. Still others are quite complicated but, by taking advantage of special properties of some particular computer, turn out to be very fast as well as economical in their storage requirements. We shall discuss one algorithm which is quite easy to understand and does not use much storage, but it is not so fast in actual computation time as some other, more complicated procedures. Then we shall consider another procedure which allows one to take advantage of prior knowledge that the numbers to be ordered are almost in order to begin with.

The problem we shall consider may be stated as follows: Let A(1), . . . , A(N) be N numbers. Sort (i.e., rearrange) these numbers into increasing order. Since this is probably a small part of a larger problem, and since it may actually need to be done several times in a large problem, we should prepare a program for sorting numbers which will be in the form of an *external function* with input arguments N and A. Before proceeding to the flow diagrams and the programs, however, we should discuss the strategy of the algorithm.

One way to arrange numbers A(1), . . . , A(N) in increasing order is to set aside another area in storage as large as the area in which the numbers are presently stored. Then we could start by moving

A(1) into the new area. If A(2) is greater than A(1), we could put it after A(1) in the new area; otherwise we could move A(1) over one place and put A(2) in front of it. Now we could determine where A(3) belongs and insert it in the proper place, moving the others over, if necessary. Eventually, we would have placed all the numbers into the new area. This method is used very often by card players who are dealt a hand for bridge or hearts, for example, and arrange their cards in order by picking up one card at a time and inserting it into the proper place in the hand. This method is quite efficient for human beings because their eyes can very rapidly scan the cards which they already hold and because they can easily shift some of the cards to make room for the newcomer. In a computer, however, the frequent shifting of many numbers would take a much longer time than other methods would require, and the use of a second storage area as large as the original would be wasteful.

Our procedure would be improved if we would search for the smallest number among A(1), . . . , A(N) and move it to the new storage area, then search for the next smallest number, and so on. This would eliminate the shifting of numbers but would still require the second storage area. One small point should be noted before we leave the second method. After a number has been selected as being less than the remaining numbers, how do we remove it from further "competition"? We could delete it by closing up the remaining numbers, but this would introduce the shifting of numbers that we are trying to avoid. A more commonly used method is to replace the selected number by a very large number, usually the largest number allowed in the language (in our case, 99999999999). This guarantees that it will not be chosen again as the smallest number.

Everyone would probably agree that the second method is better than the first method because the shifting has been eliminated. This raises an important question, however. What are the criteria by which one decides which method is better? The usual criterion—other things being equal—is to minimize the number of comparisons that have to be made among pairs of numbers. This is a measure of the amount of computation that is needed and, therefore, a rough measure of the speed of the method. The other things that are assumed equal, however, are the shifting around of numbers in storage (which is really prohibitive in the first method) and the

amount of extra storage needed. In the second method we have to compare each number with each other number; so there are N^2 comparisons. The next method which we shall discuss needs $N(N-1)/2$ comparisons, and it requires only one additional storage location, rather than a second area as large as the first. Other methods have been developed, however, which require only $N \log_2 N$ comparisons. As N gets large, $N \log_2 N$ is much smaller than $N(N-1)/2$. In fact, even for $N = 8$, we have $N \log_2 N = 8 \cdot 3 = 24$ and $N(N-1)/2 = (8 \cdot 7)/2 = 28$.

In the $N(N-1)/2$ method we start with the first number A(1) and compare it with each of the others in turn. As long as it is less than or equal to the other number, it deserves to remain in the first position. As soon as another number is smaller than A(1), however, it becomes a better candidate for the "smallest-number" position, and it should be placed in position A(1) immediately. Since the original value of A(1) must be stored somewhere else now, the easiest way to handle it is to interchange the two numbers. If we now continue to compare the new A(1) with the rest of the other numbers, interchanging whenever necessary, we find that we end up with the smallest value *in the entire set* as the value of A(1). We may completely ignore the value of A(1) from now on, since it is in its final position already. What we need to do now is find the smallest of the remaining numbers, starting with A(2) and comparing with A(3), A(4), and so on. Each time we finish a string of comparisons, the number of comparisons to be made the next time decreases by 1. The first time when each of the other $(N-1)$ numbers was compared with A(1), there were $N-1$ comparisons. The second time there would be $N-2$ comparisons, and so on. The total number of comparisons is $(N-1) + (N-2) + (N-3) + \cdots + 1$, which is easily seen to have the value of $N(N-1)/2$, since it is an arithmetic progression with $N-1$ terms.

How does one interchange two numbers? If A(I) and A(J) are to be interchanged, one might try the statements

```
A(I) = A(J)
A(J) = A(I)
```

but, unfortunately, after the first statement is executed the value of A(I) will have been destroyed, and the second statement will not

accomplish its purpose. It is necessary to preserve the value of A(I) while moving A(J) there. If we let X be some available variable whose value is not currently needed, we may write the interchange as follows:

```
X = A(I)
A(I) = A(J)
A(J) = X
```

This additional variable X is the only extra storage needed in this method since the same variable X may be used for all the interchanges.

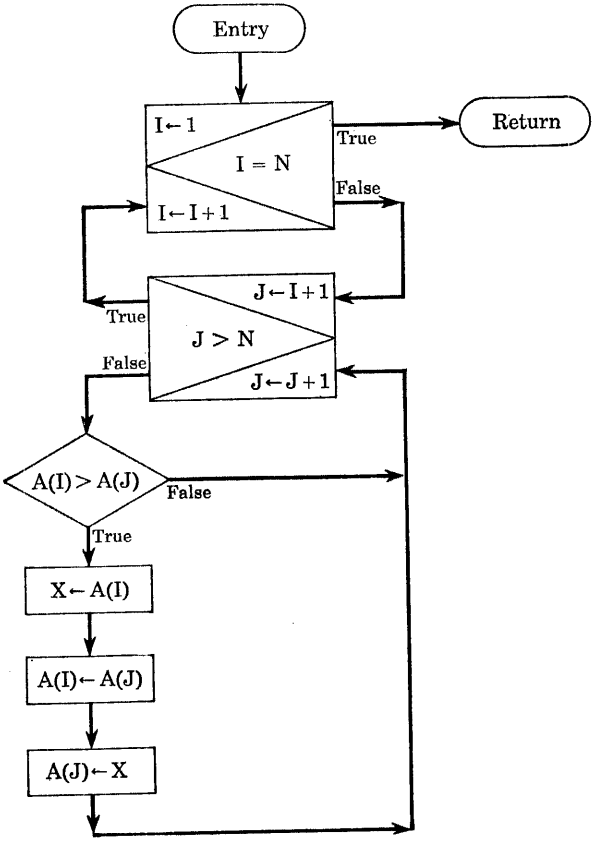


Figure 7.1*a*

We see in this algorithm a loop which starts with the designation of A(1) as a *pivot value* against which comparisons are made. After A(1) is dealt with properly, we move to A(2) as pivot value, and so on. For each choice of pivot value, we have another loop, which consists of moving through the remaining values, interchanging when necessary. The set of numbers to be compared with the current pivot value A(I) always starts with A(I + 1) and goes through A(N). Since there is nothing with which to compare A(N) when it becomes the pivot value, there is no need to use A(N) as the pivot value at all.

7.2 THE EXECUTE STATEMENT

We have just exhibited a flow diagram (Figure 7.1*a*) for an external function (let us call it SORT.), which will sort the numbers A(1), . . . , A(N) into increasing order. In every way but one, SORT. behaves just like the external functions discussed in Chapter 6. It may be entered many times from different places in the calling program; there are two arguments to specify on each call (N and A), and so on. In one respect, however, it differs: it does not produce a number as its value. It is meaningless (not just *wrong*) to write

$$Y = I + J + SORT.(N,A)$$

What we need is a way to call on SORT. without being forced into embedding it into an expression.

One way, which is at best a compromise, is to write

$$Z = SORT.(N,A)$$

where Z is some variable whose values will never be used for any other purpose. We could then let the SORT. definition program return some number as its value [after doing its real work of sorting A(1), . . . , A(N) into increasing order]. This number would then be stored as the value of Z, and we would have accomplished our purpose. This is not a very good solution, however, because it disguises what is really happening, and there is some unnecessary computation built in as well, e.g., the return of an unwanted number to be stored into an unwanted place.

A better way to handle this external function is to provide a new kind of statement in the language, a statement which says simply, "Execute the function SORT.(N,A), and then go on." Let us introduce such a statement into our language by writing

EXECUTE SORT.(N,A)

We shall always understand that after the SORT. function has done its job, we go on to the next statement in the program. What happens to the FUNCTION RETURN statement in this case? Since there is no value to return, we may simply omit the expression whose value is returned as the value of the function. It will be understood that a function whose definition program contains the statement

FUNCTION RETURN

will be called upon by means of the EXECUTE statement, and such a function will not be expected to return a value. We are now in a position to exhibit the program for SORT. (Figure 7.1*b*). Note that the new EXECUTE statement does not occur in this program. It would occur in the *calling program*.

```
      EXTERNAL FUNCTION (N,A)
      INTEGER N, I, J
      ENTRY TO SORT.
      THROUGH B, FOR I = 1, 1, I .E. N
      THROUGH B, FOR J = I + 1, 1, J .G. N
      WHENEVER A(I) .G. A(J)
          X = A(I)
          A(I) = A(J)
          A(J) = X
B     END OF CONDITIONAL
      FUNCTION RETURN
      END OF FUNCTION
```

Figure 7.1*b*

There is one new feature in this program which has not yet been discussed. There are two loops terminating at the same place, i.e., at the statement labeled B. Although the flow diagram shows that the J loop is executed completely for each value of I, this information was lost when we wrote the program. In order to preserve the

original relationship between the loops, let us adopt the following convention:

Whenever two or more loops terminate at the same statement, the scope of the last loop begun is the first to be completed. If the termination condition for that loop is not satisfied, the other loop variables remain constant while the scope of that loop is executed again. In other words, the behavior shall be the same as if the last loop that was begun actually terminated at an earlier statement than the others.

7.3 ANOTHER SORTING ALGORITHM

The SORT. program which was described in Section 7.2 is a *general-purpose* program for sorting in that it makes no assumptions about how well A(1), . . . , A(N) may be ordered before the sorting begins. This allows it to work equally well for all sets of numbers. On the other hand, there is no provision for taking advantage of any knowledge which we might have about A(1), . . . , A(N). Quite often one knows that a set of numbers which was already sorted has been only slightly rearranged (with a few new numbers added at the end, for example). There are several algorithms which take advantage of this, and we shall describe one of them.

In this method, which we shall call SORT1., we compare A(1) with A(2), and if A(2) is less than A(1), we interchange them; otherwise we do not. Then we compare A(2) with A(3) and A(3) with A(4), and so on. We shall call one such scan of the set of numbers a *pass* over the set. When we finish the last comparison and possible interchange [of A(N − 1) with A(N)], the numbers will be closer to being in order. In fact, if the numbers had been in order except that two of them had been interchanged, they would now be in order. There is only one way to know that they are now ordered, however, and that is to make a complete pass through them again and find that no further interchanges are necessary. As long as there are any interchanges at all, another pass will be needed.

As an example, Table 7.1 shows a set of seven numbers after each pass until they are sorted. During the last pass no interchanges were needed; so the algorithm stopped.

There is a way to shorten the work somewhat even in this method. As it was described above, the method calls for each pass to start with the comparison of A(1) and A(2). It may be, however, that some of the numbers near the beginning of the set were in order already, such as in the case in which a few new numbers were added at the

Table 7.1

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)
1	2	4	6	3	−1	5
1	2	4	3	−1	5	6
1	2	3	−1	4	5	6
1	2	−1	3	4	5	6
1	−1	2	3	4	5	6
−1	1	2	3	4	5	6
−1	1	2	3	4	5	6

end of a set already in order. It would be unnecessary, then, to do all those initial comparisons again. The first point at which a comparison *must* be made is at the position most recently involved in an actual interchange. In Table 7.1, for instance, we need not compare the 1 and 2 or the 2 and 4 the second time around. But because A(4) was involved in an interchange, we must be sure to begin our comparisons with A(4) on the next pass. In fact, the next pass must start with a comparison of A(3) and A(4), since the new value of A(4) might require an interchange with A(3).

Table 7.2

Pass begins	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
	1	3	5	7	9	11	13	15	17	8
A(1)	1	3	5	7	9	11	13	15	8	17
A(8)	1	3	5	7	9	11	13	8	15	17
A(7)	1	3	5	7	9	11	8	13	15	17
A(6)	1	3	5	7	9	8	11	13	15	17
A(5)	1	3	5	7	8	9	11	13	15	17
A(4)	1	3	5	7	8	9	11	13	15	17

Table 7.2 shows how a new number could be sorted into a set of nine numbers already in order, using the work-shortening feature. The column headed "Pass begins" shows where the first comparison needs to be made. The first row shows the initial positions of the numbers. In this example, we made 9 comparisons on the first pass, 2 comparisons on the second pass, and so on, amounting to 29 comparisons. If we had started with A(1) on each pass, we would have had 9 comparisons on each pass, or 54 all together. It is now clear that we must set an indicator (i.e., the value of some variable) during each pass when the first interchange is made. This indicator will show where the comparisons are to begin on the next pass. If it turns out that the indicator does not get set at all during an entire pass, this is the signal that the sorting has been completed. We shall therefore set a variable CATOR (short for *indicator*) to zero at the beginning of each pass. If A(I) and A(I + 1) are the first two numbers to be interchanged, and if CATOR is still zero, we shall set CATOR to I to remember the position of the first interchange. (If CATOR is not zero, it is not the *first* interchange.) The next pass will begin the comparisons with A(I − 1) and A(I) as we noted above with A(3) and A(4). (The value of I with which we begin the comparisons is called FIRST I in the flow diagram shown in Figure 7.2*a*.) If the value of CATOR is 1, we shall have to start with A(1) and A(2), however, since there is no A(0) in this problem. The program corresponding to this flow diagram would look as shown in Figure 7.2*b*.

One new kind of statement has been used in this program, i.e., the CONTINUE statement which bears the label PASS. The situation arose in this program (and it is quite common) in which we wished to transfer to the end of the scope of an iteration and bypass all the remaining computation in that scope. In the sorting algorithm, if A(I) is less than or equal to A(I + 1), we needed to go directly on to the next comparison, and we did not wish to execute any part of the interchange computation. It was necessary to separate the label PASS, which represented the end of the scope, from any of the computation in the loop. The statement CONTINUE acts as a place to hang a label, but introduces no additional computation. If there were no CONTINUE statement, and we had put the label PASS

on the statement

WHENEVER CATOR .E. 0, CATOR = I

what would have happened? Since CATOR is set to zero by the statement labeled ONE, the first time through the loop, *whether there was an interchange or not*, CATOR would be set to I. The CONTINUE statement allows us to jump past the conditional statement.

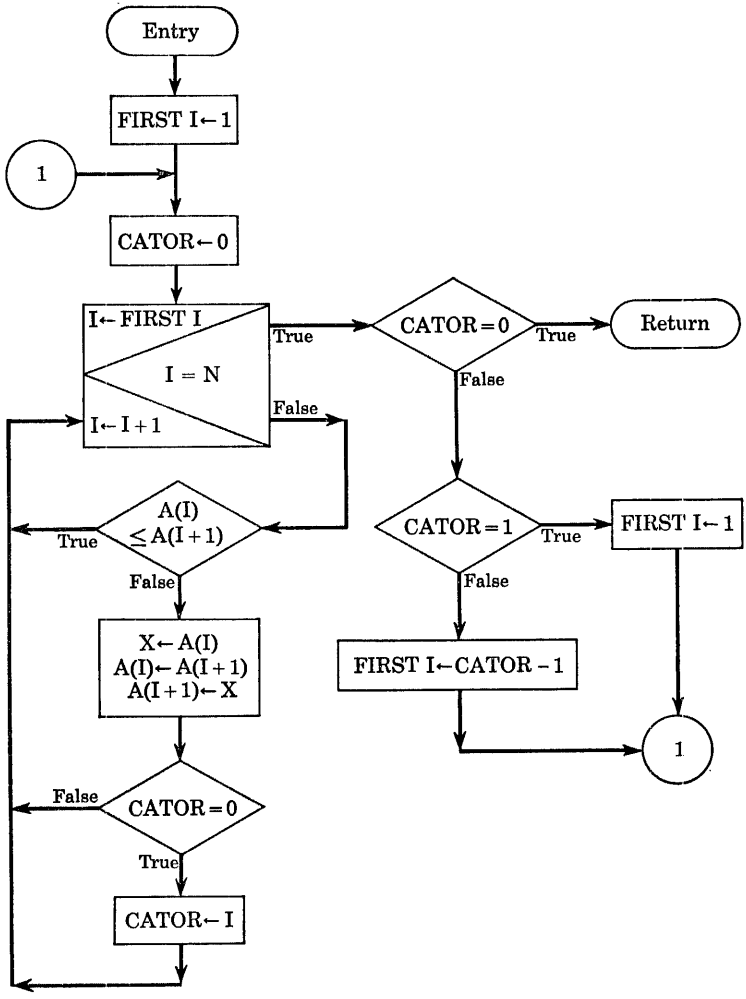


Figure 7.2*a*

```
      EXTERNAL FUNCTION (N,A)
      ENTRY TO SORT1.
      INTEGER N, FIRST I, CATOR, I
      FIRST I = 1
ONE   CATOR = 0
      THROUGH PASS, FOR I = FIRST I, 1, I .E. N
      WHENEVER A(I) .LE. A(I + 1), TRANSFER TO PASS
      X = A(I)
      A(I) = A(I + 1)
      A(I + 1) = X
      WHENEVER CATOR .E. 0, CATOR = I
PASS  CONTINUE
      WHENEVER CATOR .E. 0, FUNCTION RETURN
      WHENEVER CATOR .E. 1,
          FIRST I = 1
      OTHERWISE
          FIRST I = CATOR - 1
      END OF CONDITIONAL
      TRANSFER TO ONE
      END OF FUNCTION
```

Figure 7.2b

7.4 A SEARCH ALGORITHM

Now that we know how to order a set of numbers, we must find ways to *search* for a particular number in the set in such a way as to take advantage of the order. One straightforward way to search through a set of numbers is to start at one end and see if the *searching argument* (i.e., the number which we are trying to find in the table or set of numbers) is there by comparing it with each number in turn. We did this in Chapter 5, in fact, when looking for the position number of a character in the standard alphabet. (It was this search that led us to the case of a loop with an empty scope, since the entire computation was contained in the termination condition of the loop.) Such a straightforward *table look-up*, as it is called, is very useful if the table is ordered in such a way that the most frequently encountered searching arguments occur at the beginning of the table—regardless of the size—so that the search ends quickly. A specialist in cryptography might have ordered the standard alphabet with the letter *e* first, for example, since *e* is the most commonly used letter in Eng-

lish. For a given language, the relative frequencies of occurrence of the various letters are easily obtained. In Chapter 5 we did not have this "high-frequency ordering," but we used the table-look-up procedure because we had no additional information to warrant other search procedures.

Now we shall assume that the table A(1), . . . , A(N) has been sorted into increasing order, and we shall look at a searching method which takes very good advantage of this. The method we shall describe is sometimes called the "binary search," since at each step it cuts in half the part of the table which is still to be searched. Because we shall be dividing repeatedly by 2, we shall want the table size to be a power of 2. We cannot assume that every table we need to search has this property, however, but we can think of every table as being embedded in a larger table with a number of elements which is a power of 2. How this enlargement is handled will become clear as we proceed. The first step in the search makes sure that the argument is somewhere in the range of the numbers in the table. This is done by asking if ARG $<$ A(1) or ARG $>$ A(N). In either of these cases, the search is over, and ARG is *not* in the table. We shall continue the discussion, then, assuming that A(1) $\leq$ ARG $\leq$ A(N). Knowing the size of the original table, say N, we first compute the smallest power of 2 which is greater than or equal to N. For example, if N = 5, we would choose 8. This can be done using one statement in our language:

```
B    THROUGH B, FOR X = 1, X, X .GE. N
```

The reader should verify that this statement does compute as the value of X the desired power of 2. (The variable X is repeatedly incremented by the value of X, hence doubled. When the loop terminates, X will have the desired power of 2 as its value.) Having computed this power of 2, we shall treat the table to be searched as if it were really this large, being careful to avoid using any numbers not actually in the original table. It will be clear from the discussion that follows that if the power of 2 that we use is 2^n, then at most $n + 1$ comparisons will be made in the search.

We now set Y = X, and if ARG = A(Y) *and* Y $\leq$ N (so that we are still in the original table), we are done, i.e., we have found ARG

in the table. This condition, i.e., ARG = A(Y) *and* Y $\leq$ N, is the condition which must be satisfied if we are to make the claim at any time that ARG has been found in the table. We will see later that this condition will serve as the termination condition for the iteration which we will set up. If, at any stage of the search, ARG > A(Y) *or* Y > N, we must clearly move farther down into the table. We may do this the first time, for example, by jumping to the middle of the table. (The way to jump to the middle of the table is to subtract from Y half of X. We then cut X in half to prepare for the next jump.) If ARG is greater than A(Y) at the middle of the table, we next restrict our attention to the upper half of the table; otherwise we restrict ourselves to the lower half. If it is the upper half, we *add* to Y half of the current value of X and again set X = X/2, and we are now considering the three-quarter point in the table. If it is the lower half, we *subtract* from Y half of X and set X = X/2, and we are now considering the one-quarter point in the table. We simply repeat this procedure until X, by repeatedly being cut in half, is reduced to 1, or we find at some point that A(Y) = ARG. If we do have X = 1, ARG is not in the table.

Consider Table 7.3 where, for example, A(9) = 31.

Table 7.3

I	1	2	3	4	5	6	7	8	9	10
A(I)	2	3	4	10	11	13	17	23	31	40

If we search with ARG = 31, we obtain the sequence of values of X and Y shown in Table 7.4. [The final value of Y should be 9, indicating that 31 has been found as the value of A(9).] The initial value of X is 16, since this is the smallest power of 2 greater than 10.

Table 7.4

Y	X	Action to be taken (X = X/2 in every case)
16	16	Y > 10, so Y = Y − X/2
8	8	ARG > A(8), so Y = Y + X/2
12	4	Y > 10, so Y = Y − X/2
10	2	A(10) > ARG, so Y = Y − X/2
9	1	A(9) = ARG and Y $\leq$ 10, so the search ends

If ARG = 20, so that it is not in the table at all, we would have the sequence shown in Table 7.5.

Table 7.5

Y	X	Action to be taken (X = X/2 in every case)
16	16	Y > 10, so Y = Y − X/2
8	8	ARG < A(8), so Y = Y − X/2
4	4	ARG > A(4), so Y = Y + X/2
6	2	ARG > A(6), so Y = Y + X/2
7	1	ARG > A(7), but X = 1, so search ends negatively

Table 7.4 suggests that whenever ARG is in the table, Y will be the index of its position in the table. Examination of a few cases will show that this must always be true. On the other hand, Table 7.5 would indicate that whenever ARG is not in the table, Y will point to the number in the table which is just below the position ARG would have occupied if it had been there. In that example, Y had the value 7, so that A(Y) = 17, which is just below ARG = 20. Unfortunately, Y will not always point to the number *just below* ARG. Consider Table 7.6, where ARG = 30.

Table 7.6

Y	X	Action to be taken (X = X/2 in each case)
16	16	Y > 10, so Y = Y − X/2
8	8	ARG > A(8), so Y = Y + X/2
12	4	Y > 10, so Y = Y − X/2
10	2	ARG < A(10), so Y = Y − X/2
9	1	ARG < A(9), but X = 1, so search ends negatively

In this case ARG is not in the table, but the final value for Y does not point to the number just below ARG in size. It points instead to the number *just above* ARG. This time it actually points to the position ARG should occupy if it is now to be inserted into the table. It is unfortunate that sometimes Y will point to the position just below the appropriate position for inserting ARG. Let us agree to

watch for this situation and increase Y by 1 if A(Y) < ARG, so that we will always find Y pointing to the proper position in the table for inserting ARG. In the extreme cases which we caught earlier, e.g., ARG < A(1) or ARG > A(N), we may automatically set Y to 1 and N + 1, respectively, so that Y will still point to the position where ARG should be inserted.

Let us set up the flow diagram and program for this algorithm (which we shall call SEARCH.) as an external function, since this is likely to be called upon by other, more complicated programs. The arguments in the call for SEARCH. should be N, A, ARG, and Y. We should also provide a way to recognize the case in which ARG is not in the table at all. If this should happen, we would probably like to transfer to another part of the program in order to do something about it. We shall add one more argument to the call for SEARCH., e.g., the label of the statement to which the transfer is to be made in case ARG is not in the table. It will be understood that if we transfer to this statement, Y will have the meaning indicated above; i.e., it will point to the position in which ARG should have appeared. We shall now exhibit a flow diagram for SEARCH. (Figure 7.3*a*), followed by the corresponding program (Figure 7.3*b*). The arguments to this external function are N, A, ARG, Y, and NOT IN, where NOT IN is the label of the statement to which we transfer if ARG is not in the table. The first two questions asked in the flow diagram determine whether ARG lies in the table at all. The small loop after that computes the appropriate power of 2, as described above. The basic search is contained in the larger loop that follows. Notice that the lower iteration box has for its initialization X ← X. This illustrates the occasional situation in which the variable of the iteration already has its initial value, and we do not need to do anything to it. Another interesting point about this iteration is that the termination condition does not involve X at all. The change in Y depends on X, but the termination condition does not mention X explicitly.

In order to understand this algorithm thoroughly, the flow diagram and program should be simulated manually for at least the examples given above. One new statement type was needed in this program. Since the variable NOT IN is an argument to this external function, there is no way to tell that this variable is a statement

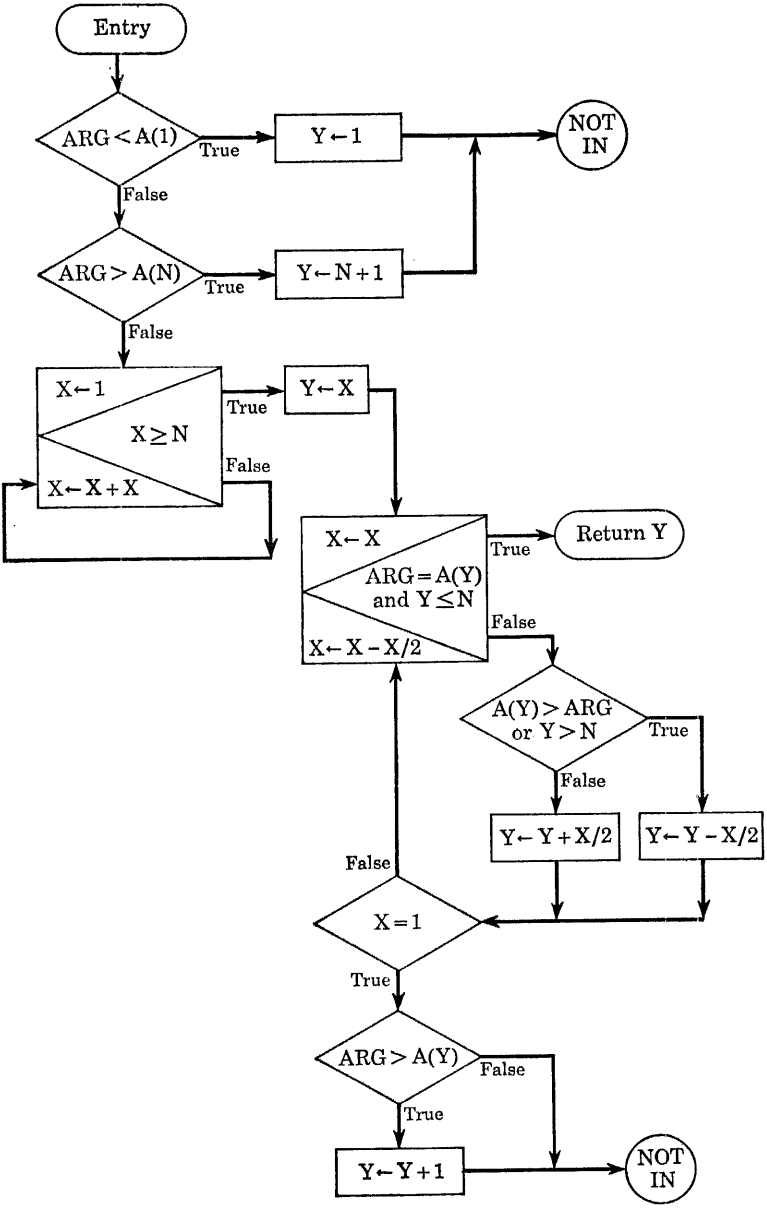

Entry
ARG < A(1)
True
Y ← 1
NOT IN
False
ARG > A(N)
True
Y ← N + 1
False
X ← 1
X ≥ N
X ← X + X
True
Y ← X
False
X ← X
ARG = A(Y) and Y ≤ N
X ← X - X/2
True
Return Y
False
A(Y) > ARG or Y > N
True
False
Y ← Y + X/2
Y ← Y - X/2
False
X = 1
True
ARG > A(Y)
False
True
Y ← Y + 1
NOT IN

Figure 7.3*a*

```
    EXTERNAL FUNCTION (N,A,ARG,Y,NOT IN)
    INTEGER N, Y, X
    ENTRY TO SEARCH.
    WHENEVER ARG .L. A(1)
        Y = 1
        TRANSFER TO NOT IN
    OR WHENEVER ARG .G. A(N)
        Y = N + 1
        TRANSFER TO NOT IN
    END OF CONDITIONAL
B   THROUGH B, FOR X = 1, X, X .GE. N
    Y = X
    THROUGH C, FOR X = X, -X/2, ARG .E. A(Y) .AND. Y .LE. N
    WHENEVER ARG .L. A(Y) .OR. Y .G. N
        Y = Y - X/2
    OTHERWISE
        Y = Y + X/2
    END OF CONDITIONAL
C   WHENEVER X .E. 1, TRANSFER TO D
    FUNCTION RETURN Y
D   WHENEVER ARG .G. A(Y), Y = Y + 1
    TRANSFER TO NOT IN
    STATEMENT LABEL NOT IN
    END OF FUNCTION
```

Figure 7.3b

label in the calling program and that it thus may legitimately appear in a TRANSFER TO statement. We therefore include a declaration

```
STATEMENT LABEL NOT IN
```

to indicate that NOT IN is of statement-label mode, just as we indicate that other variables are of integer mode. This is not needed for variables such as B and C in Figure 7.3*b*, which are obviously of statement-label mode. If we had not provided NOT IN as an argument, and therefore as a link to the calling program, we would have had to provide some other action in case the argument is not in the table.

One further point may be made about this program. The question may be raised as to why Y must be an argument to the function when its value is being returned as the value of the function. It is true that in the case of a successful search, the functional value is the

value of Y. In the case of an unsuccessful search, however, we merely transfer to NOT IN, and there is no value returned. To make use of the value of Y as an indication of the place ARG should have occupied in the table, we have made Y an argument, and it is therefore available to the calling program. If we had returned to the calling program by means of the

FUNCTION RETURN Y

statement each time, whether ARG was in the table or not, we would have no way of knowing each time whether the search was successful.

PROBLEMS

1. Modify the SORT. definition program to sort the set of numbers A(1), . . . , A(N) into decreasing order. (*Hint:* Only one letter need be changed in the program.)

2. Write a new SORT. program and flow diagram which will have a third argument T to indicate whether A(1), . . . , A(N) should be sorted into increasing or decreasing order depending on whether T = 1 or T = −1, respectively, and which will do either kind of sorting on request via this third argument.

3. Suppose that A(1), . . . , A(N) were a set of telephone numbers and D(1), . . . , D(N) were the names of the users of these telephones. Assume that D(I) is the name of the user of the telephone whose number is A(I), i.e., that the two lists correspond. Whenever the numbers A(1), . . . , A(N) are rearranged, the names D(1), . . . , D(N) must be simultaneously rearranged; or the correspondence will be lost. Modify your solution to Problem 2 to include a fourth argument D and to rearrange D(1), . . . , D(N) while sorting A(1), . . . , A(N) into either increasing or decreasing order.

4. Write a flow diagram and program to sort a bridge hand into order, as follows. Suppose that V(1), . . . , V(13) are the values of the cards in the hand, with the jack, queen, and king having values 11, 12, and 13, respectively. Suppose, also, that S(1), . . . , S(13) are the suits of the 13 cards, with spades, diamonds, clubs, and hearts represented by the alphabetic constants $SPADE$, $DIAMND$, $CLUB$, and $HEART$, respectively. (Inside the computer alphabetic constants are represented by integers; so it is convenient to consider them to be of integer mode. With-

out asking which integers are used for their representation, we may still sort them into either increasing or decreasing order. All the spades will then end up together, and so on.) Using the two-vector external function SORT., which was developed in Problem 3, write your program to call on SORT. to rearrange the cards first into groups of the same suit. Then, after finding out how many there are of each suit, call on SORT. again to rearrange the cards *by value* within each suit. (The program should be written to handle N cards, rather than just 13, so that one could also use it for games other than bridge.)

5. Sometimes a table contains two entries, such as name and telephone number, stored in consecutive locations. For example, we might have A(1) be a name and A(2) the corresponding telephone number. Then A(3) would be another name, and A(4) would be the number which corresponded to that name, and so on. Write a flow diagram and program for an external function called SORT2. which sorts such a table according to increasing telephone number, using the algorithm of SORT1. (Figure 7.2*a*).

6. Write a flow diagram and program which uses the algorithm of SEARCH. on a *two-entry* table such as that described in Problem 5. Assume that ARG will be a telephone number and that we wish to find the name of the subscriber with that telephone number.

7. The algorithm for SORT1. included a work-shortening feature which remembered the earliest interchange for each pass over the set being sorted. The next pass could begin at this point (or one entry earlier), thus saving repeated scanning of entries already in order. An additional saving in time can be achieved by remembering as well the *last* interchange on each pass, so that the next pass can often be stopped before all the entries are scanned. Modify the flow diagram and program for SORT1. to include this additional feature.

8. It is often necessary to insert a new number into a table if it is not already there. Since SEARCH. returns an indication as to where the argument should have been if it could not find it, it should be possible to move the rest of the table over and insert the new number at that point. Construct a flow diagram and program for an external function INSERT., with arguments N, A, ARG, Y (corresponding to the arguments in SEARCH.), which will insert ARG into the table A at the point indicated by Y and increase N by 1 to reflect the increased table size.

CHAPTER EIGHT

THE CORRELATION COEFFICIENT

8.1 THE PROGRAM

LET US CONSIDER now one of the simplest of all prediction devices, the *correlation coefficient.* This is a number which is attached to several sets of data to indicate that their behavior is "similar" in some sense. Thus the weather in the Midwestern part of the United States would be highly correlated (i.e., have a high correlation coefficient) with the weather which occurred three days earlier in the western part of the country. On the other hand, one would not expect to find much correlation between the number of runs scored in the American League each day and the price of milk.

The more highly correlated two phenomena are, the more confidently can we use one to predict the other. Thus, we do use the weather in one part of the country to predict the weather for other areas, but we would not expect to predict baseball scores from the price of milk. The correlation coefficient gives a measure, then, of the degree of confidence we may have when using one of two phenomena to predict the other. There are other, more sophisticated prediction devices in use, such as multiple correlation coefficients, regression equations, and so on, but the simple correlation coefficient will suffice here.

In order to obtain some intuitive feeling for the correlation coefficient (often designated r), let us consider a very simple example of two observations of each of two phenomena. We may label the observations of the first phenomenon x_1 and y_1 and the observations of the second phenomenon x_2 and y_2. Plotting these numbers as coordinates of points (x_1,y_1) and (x_2,y_2) on a graph, we may then draw line segments from the origin of the graph to the points, as in Figure 8.1.[1] One question arises immediately: Does "similar behavior" mean that the two points should be very near to each other? It is entirely possible that the actual numbers involved might differ by a great deal, and in this case the points would be far apart. Their *variation* could be quite similar, however, and this is what is important to us. Thus, if the two phenomena we are observing are highly correlated, we would expect that whenever $y_1 = 2x_1$, then $y_2 = 2x_2$, at least approximately. In other words, we would expect to find that

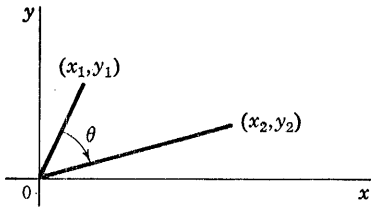


Figure 8.1

$$\frac{y_1}{y_2} = \frac{x_1}{x_2}$$

or, rewriting it in a different way,

$$\frac{y_1}{x_1} = \frac{y_2}{x_2}$$

The first equation suggests that the data for one phenomenon should be proportional to the data for the other. The second equation may be interpreted as implying that the two line segments in Figure 8.1 have the *same slope*, i.e., the lines containing these segments coincide. In other words, the two *points* do not have to be very near each other to be highly correlated, but the *lines* they determine through the origin must be as near coinciding as possible.

If we wish to measure the extent to which the lines do *not* coincide,

[1] The usual approach to the correlation coefficient in statistics does not use this geometrical point of view, but the result is the same.

we might use the angle θ between the lines (or some function of the angle θ), so that the most highly correlated case would have $\theta = 0°$. The measure actually used in statistical work is not the angle θ itself, but the cosine of the angle θ. One of the reasons for using the cosine of the angle rather than the angle itself is that there is a simple formula for the cosine in terms of the original data. This formula comes directly from the law of cosines in trigonometry. Applied to the triangle in Figure 8.2, which has sides of lengths a, b, and c, respectively, the law of cosines states that

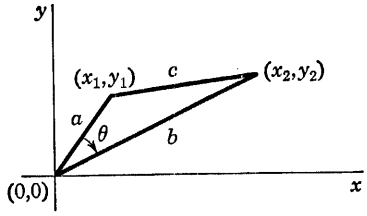


Figure 8.2

$$\cos \theta = \frac{a^2 + b^2 - c^2}{2ab}$$

We may obtain expressions for a, b, and c by applying the standard formula for the distance between any two points whose coordinates are known. For example, to compute c, the distance between (x_1,y_1) and (x_2,y_2), we have

$$c^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Therefore, since $a = \sqrt{x_1^2 + y_1^2}$, $b = \sqrt{x_2^2 + y_2^2}$, we have

$$\cos \theta = \frac{(x_1^2 + y_1^2) + (x_2^2 + y_2^2) - [(x_2 - x_1)^2 + (y_2 - y_1)^2]}{2\sqrt{x_1^2 + y_1^2}\sqrt{x_2^2 + y_2^2}}$$

$$\cos \theta = \frac{x_1x_2 + y_1y_2}{\sqrt{x_1^2 + y_1^2}\sqrt{x_2^2 + y_2^2}}$$

This gives us a very simple formula which we can use to measure how nearly the two lines coincide (or fail to coincide).

If there are three observations (x_1,y_1,z_1) and (x_2,y_2,z_2) for each of the two phenomena, we might extend what we have done by plotting the data as two points in three-dimensional space, as in Figure 8.3. The formula used above for the cosine of θ may now be used in its three-dimensional form; so the correlation coefficient r now is

obtained as follows:

$$r = \frac{x_1x_2 + y_1y_2 + z_1z_2}{\sqrt{x_1^2 + y_1^2 + z_1^2}\,\sqrt{x_2^2 + y_2^2 + z_2^2}}$$

As an illustration, consider the two sets of numbers {1,3,5} and {2,6,10}. The second set contains numbers which are just twice the numbers in the first set; so we would expect these sets to be highly

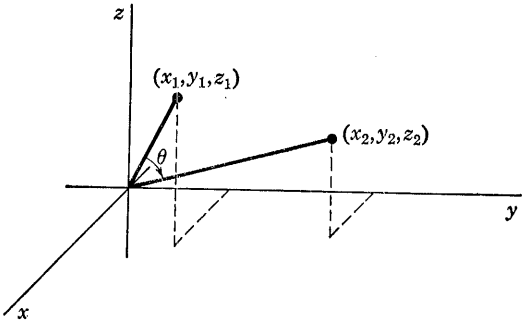


Figure 8.3

(in fact, *perfectly*) correlated. Since the cosine of an angle varies only from -1 to $+1$, the largest $|r|$ can become is 1, and this represents an angle whose cosine is 1 (or -1), i.e., $\theta = 0°$ (or $\theta = 180°$). Our data determine the two points (1,3,5) and (2,6,10), which actually lie on the same line, as shown in Figure 8.4. Therefore the

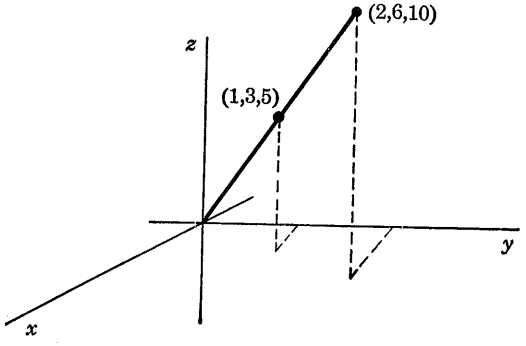


Figure 8.4

angle θ between the lines from the origin to each of the two points is 0°. Assuming that we do not know this, however, we use the formula above for r and we obtain

$$r = \frac{1 \cdot 2 + 3 \cdot 6 + 5 \cdot 10}{\sqrt{1^2 + 3^2 + 5^2}\,\sqrt{2^2 + 6^2 + 10^2}}$$

$$r = \frac{70}{\sqrt{35}\,\sqrt{140}} = \frac{70}{\sqrt{4900}} = 1$$

If we now choose another point on the same line, but on the opposite side of the origin, such as $(-3, -9, -15)$, we would expect to find $r = -1$, since $\theta = 180°$ and $\cos 180° = -1$.

$$r = \frac{1(-3) + 3(-9) + 5(-15)}{\sqrt{1^2 + 3^2 + 5^2}\,\sqrt{(-3)^2 + (-9)^2 + (-15)^2}}$$

$$r = \frac{-105}{\sqrt{35}\,\sqrt{9(35)}} = \frac{-105}{3(35)} = -1$$

Two sets of data with $r = -1$ are said to be *perfectly negatively correlated.* It is possible for r to take on any value between -1 and $+1$, and the closer $|r|$ is to 1, the better the correlation between the data.[1]

As another example, suppose we use the points (1,3,5) and (2,6,8) to represent three observations of two phenomena. These numbers are almost, but not quite, the same as those illustrated in Figure 8.4. Since we have moved only one point slightly, we should expect to find that the two phenomena are still highly correlated, but not perfectly correlated. In other words, we would expect the correlation coefficient to be near $+1$, but not equal to $+1$. We find that

$$r = \frac{1 \cdot 2 + 3 \cdot 6 + 5 \cdot 8}{\sqrt{1^2 + 3^2 + 5^2}\,\sqrt{2^2 + 6^2 + 8^2}}$$

$$= \frac{60}{\sqrt{35}\,\sqrt{104}}$$

$$= \frac{60}{\sqrt{3640}}$$

$$= .9945$$

[1] In statistics, r^2 (sometimes called the *coefficient of determination*) is the fraction of the variance in Y which is caused by the variance in X. When $r = .5$, so that $r^2 = .25$, we may say that 25 per cent of the variance in Y is due to the variance in X.

This is, in fact, the cosine of 6°; so the lines joining the origin to the points (1,3,5) and (2,6,8), respectively, form an angle of 6° with each other.

If we now consider the general case of many observations of two phenomena, we may write the natural generalization of the formulas we have already seen for r. Let us suppose that there are N observations $X_1, X_2, \ldots, X_N$ for phenomenon X and N observations $Y_1, Y_2, \ldots, Y_N$ for phenomenon Y. Then we have for r:

$$r = \frac{X_1Y_1 + X_2Y_2 + \cdots + X_NY_N}{\sqrt{X_1^2 + X_2^2 + \cdots + X_N^2}\,\sqrt{Y_1^2 + Y_2^2 + \cdots + Y_N^2}}$$

(We might talk about N-dimensional space, just as we used two-dimensional and three-dimensional space to plot our points above, and even call r the cosine of some angle in this N-dimensional space. We could even go on to talk about *infinite-dimensional* spaces.)

One thing we would like to be able to verify, however, is that r still has the property it inherited from the cosine, e.g., $|r| \leq 1$. There is a very famous inequality, called Schwarz's inequality, which states

$$|X_1Y_1 + \cdots + X_NY_N| \leq \sqrt{X_1^2 + \cdots + X_N^2}\,\sqrt{Y_1^2 + \cdots + Y_N^2}$$

for all numbers $X_1, \ldots, X_N, Y_1, \ldots, Y_N$. If we assume that not all the X observations are zero and that not all the Y observations are zero (which was implicitly assumed as soon as we wrote the formula for r), we may divide both sides of Schwarz's inequality by the product of the two square roots on the right. This yields the inequality $|r| \leq 1$.

Now let us move on to the flow diagram and program for computing r. Since this program would probably be used by other, more complicated programs, let us write it in the form of an external function. Afterward we shall consider the form of a program which would call on this external function. There are several ways to organize the computation for r. Two different flow diagrams are exhibited in Figures 8.5*a* and 8.5*b*. (We use capital letters here in anticipation of the symbols used later in writing the program.)

The variables SUM, XSUM, and YSUM are used to accumulate the sum of the products X_IY_I, the sum of the squares of the X's, and

the sum of the squares of the Y's, respectively. The method of accumulating sums employed here is very commonly used on computers. One starts with zero, and then each term of the sum is added in turn, until finally the entire sum has been computed.

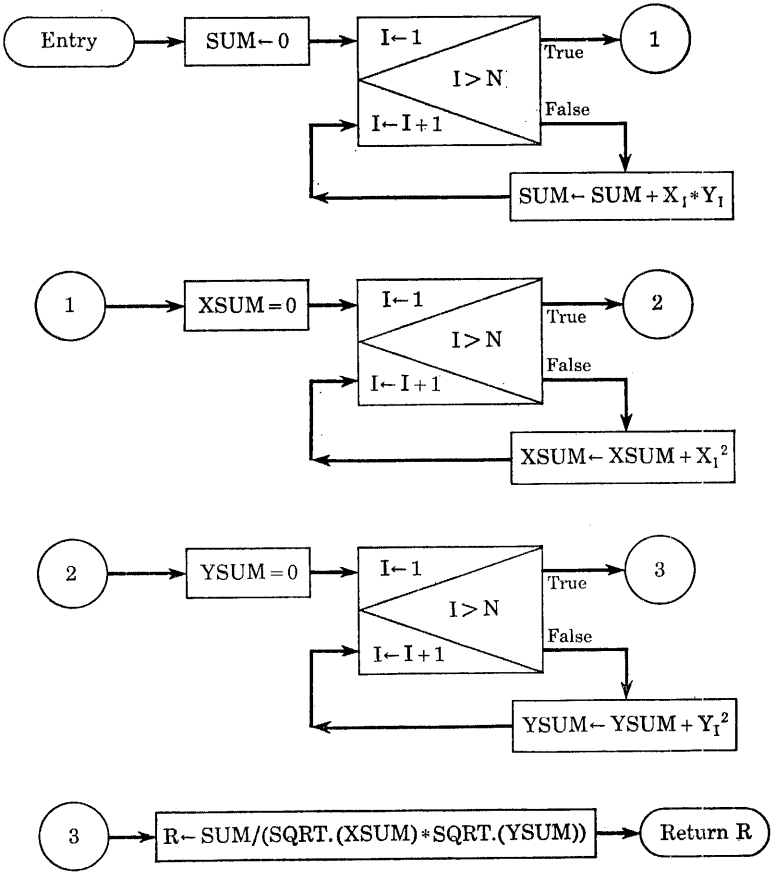


Figure 8.5*a*

These two diagrams illustrate that there may be several computational algorithms, all giving the correct answer, corresponding to one mathematical algorithm. These computational methods will usually differ in the amount of computation required, hence in the time needed to carry out the computation. Sometimes, because of roundoff error, the answers may actually be slightly different, and

an entire branch of mathematics, called *numerical analysis*, is devoted to the study of efficient methods for computing solutions to various problems, with particular emphasis on the determination and control of roundoff and other kinds of errors which occur in numerical work. In this example the value computed for R in Figure 8.5*b* will probably be closer to the theoretically correct answer because the error incurred in computing the square root (and there is always some error, if only because we cannot carry an infinite number of digits inside the computer) will be incurred only once instead of twice, as in Figure 8.5*a*. The time needed to compute R (in the

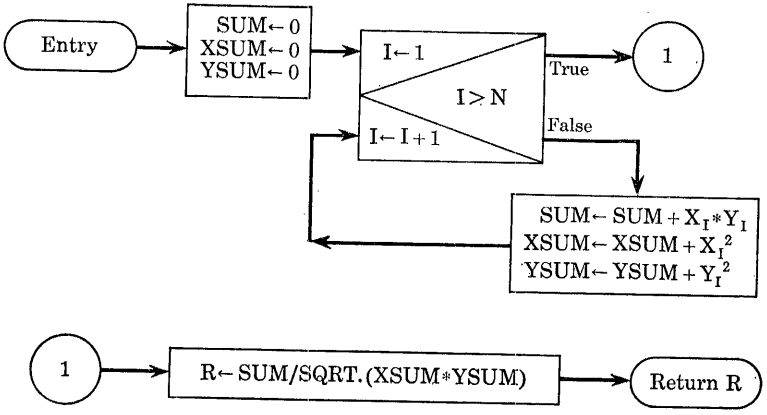


Figure 8.5*b*

box involving the square root) will also be less in Figure 8.5*b*, since the square-root computation, a time-consuming process, is done only once. Even more important, the so-called *red-tape* time involved in incrementing and testing the loop variable three times in Figure 8.5*a* is drastically reduced in Figure 8.5*b*. From several points of view, then, the second diagram provides a much better algorithm. (Comparisons are not always this one-sided, however, and the decision as to which algorithm to use is seldom this easy.) We can see the difference between the two algorithms again in the corresponding programs, which are shown in Figures 8.6*a* and 8.6*b*, respectively. Note that X and Y have been made arguments to the function, and we therefore do not include DIMENSION statements for them here. Since the effect of such statements is to set aside storage for the vec-

tor so declared, we leave this to the *calling program* which will specify which particular vectors are to be used. Storage space will have been allocated in the calling program, and none need be allocated for these arguments in the external-function definition program.

```
        EXTERNAL FUNCTION (N,X,Y)
        ENTRY TO CORR.
        SUM = 0
        THROUGH LOOP1, FOR I = 1, 1, I .G. N
LOOP1   SUM = SUM + X(I) * Y(I)
        XSUM = 0
        THROUGH LOOP2, FOR I = 1, 1, I .G. N
LOOP2   XSUM = XSUM + X(I) * X(I)
        YSUM = 0
        THROUGH LOOP3, FOR I = 1, 1, I .G. N
LOOP3   YSUM = YSUM + Y(I) * Y(I)
        FUNCTION RETURN SUM/(SQRT.(XSUM) * SQRT.(YSUM))
        INTEGER I, N
        END OF FUNCTION
```

Figure 8.6*a*

```
        EXTERNAL FUNCTION (N,X,Y)
        ENTRY TO CORR.
        SUM = 0
        XSUM = 0
        YSUM = 0
        THROUGH LOOP, FOR I = 1, 1, I .G. N
        SUM = SUM + X(I) * Y(I)
        XSUM = XSUM + X(I) * X(I)
LOOP    YSUM = YSUM + Y(I) * Y(I)
        FUNCTION RETURN SUM/SQRT.(XSUM * YSUM)
        INTEGER I, N
        END OF FUNCTION
```

Figure 8.6*b*

8.2 INPUT-OUTPUT STATEMENTS

In all the programs we have considered so far, we have never raised the question as to how the numbers or letters with which we are computing are brought into the computer. It is clear that we shall need some kind of statement which will bring in (or *read*) some appro-

priate amount of *data*. Equally important, we shall need a statement that will cause the computer to produce *results* of the computation in some form which will be intelligible to the user.

For input, we shall use the statement

READ DATA

which shall indicate that several numbers shall be brought into the computer. We shall label each number with the name of the variable for which it is a value, and we shall continue to bring in numbers until we encounter a termination mark, such as an *, after one of the numbers. When this character is brought into the computer, no more numbers are brought in until a new call for data is executed. For example, suppose that a program begins with the statement

START READ DATA

and ends by executing the statement

TRANSFER TO START

Each time the program transfers to START, it will execute the READ DATA statement and read into the computer a sequence of numbers, until an * is encountered after one of the data values. Then the remaining numbers, if any, will wait until the next execution of a READ DATA statement.

It often happens that a program will repeatedly transfer back to its beginning statements, read a new set of data, and do the same computation on these new data. In such programs the second set of data is usually exactly the same as the first set of data except for a few key numbers. If the program is written so that the numbers which do not vary from one set of data to the next are not changed by the computation, then the second (and succeeding) sets of data may assume that these values are already set properly. Only the new values need be read in as the second set of data. For example, in the social security problem which we considered in Chap. 4, the first set of data would specify the values of WAGES and SALARY for the first man, and THRESH and RATE as well (see Fig. 4.2). The second set of data would need to provide new values only for WAGES and SALARY. The values of THRESH and RATE

which were previously read in would still be available for the second set of data.

To keep the rules for preparing the data as simple as possible, let us use the actual name of the variable to indicate where the value should go when it is read, and let us write constants just as we do in the statements of the program. A typical sequence of data values (terminated by an *) might then be

$$A = 1.2, C(3) = 4, D = .5 *$$

Any arithmetic that needs to be done on these values can be written into the program; so there is no need to allow *expressions* here. We shall expect a *constant* on the right of each equal sign. For the same reason, we shall expect only *constant* subscripts to be used on the left side of the equal sign. If we wish to bring in several values for entries in a vector, such as $Q(6) = 1.2$, $Q(7) = 0$, $Q(8) = .34$, and $Q(9) = -1.03$, we may take advantage of their being stored consecutively and write $Q(6) = 1.2, 0, .34, -1.03$.

Our treatment of output will be very similar, except that here we must provide a list of values to be produced as output. Since most computers print results on some form of printer or typewriter, let us write

PRINT RESULTS A, B + C, 3.14, F.(X,Y)

as a typical statement listing four expressions whose values are to be printed. We shall assume some arbitrarily fixed printing format for the results, such as six columns of numbers, and we shall stipulate that every execution of a PRINT RESULTS statement causes printing on a new line.

Two small extensions of the rules for forming expressions (see Section 2.4) will be useful here. It is sometimes convenient to print a comment (such as NO SOLUTION) as part of the output. The rules given earlier for forming alphabetic constants allow only six characters, including spaces, between dollar signs (which act like quotation marks). We shall allow any length alphabetic constant in PRINT RESULTS statements, so that we can write comments. Thus, one might write

PRINT RESULTS $NO SOLUTION, INPUT WAS A= $,
A, $, B = $, B, $, C = $, C

to obtain the printed output

NO SOLUTION, INPUT WAS A = 5.6, B = 1.2, C = 0.51

The other change which we shall want to make in the rules for forming expressions concerns the printing of values of a vector. If we need to print the values of Q(1), . . . , Q(17), it would be very convenient if we could write

PRINT RESULTS Q(1), . . . , Q(17)

We shall call this the *block* notation, since it describes a block of storage, and agree that this block notation may be used, but only on PRINT RESULTS statements.

As an illustration of the use of input and output statements, let us write a small calling program (Figure 8.7) which makes use of the correlation coefficient external function CORR. of Figure 8.6*b*:

```
START  READ DATA
       PRINT RESULTS CORR.(N,X,Y)
       TRANSFER TO START
       DIMENSION X(300), Y(300)
       INTEGER N
       END OF PROGRAM
```

Figure 8.7

Here we arbitrarily set an upper limit on N of 300 when we used this figure in dimensioning X and Y. Note that the program as written continually returns to START to read more data each time it finishes processing a set of data. The usual procedure is that the computation stops when there are no more data waiting to be called in. Then one may vary the number of sets of data at will. We shall see additional examples of the use of input and output statements later.

PROBLEMS

1. Insert appropriate input (i.e., READ DATA) and output (i.e., PRINT RESULTS) statements in some of the programs which were developed in previous chapters. In particular, for those indicated below, the following

values should be considered results:

Figure	*Results*
3.2	Q(50), Q(25), Q(10), Q(5), Q(1)
4.2	WAGES, SALARY, TAX
5.4	CODE(1), . . . , CODE(N)

2. It is usually considered a good idea to print the values of the data immediately after bringing them into the computer by means of the READ DATA statement. Why is this a good idea? Modify the three programs produced for Problem 1 to include this initial printing.

3. One of the most important steps in creating a good computer program is the check that the program actually works. For an ordinary program, i.e., not an external function, this amounts to testing the program on several well-chosen sets of data. For this purpose the data should be chosen to force the program through as many different paths as possible. For example, the social security program should be tested with data which require that the tax be zero, with data that require that only part of this week's earnings be taxed, and with data that require that all this week's earnings be taxed. Of special interest are the *boundaries*, for example, the data which make

WAGES + SALARY

exactly equal to THRESH. The check data should test whether these boundary situations are treated correctly. Choose appropriate check data for the following programs:

a. Figure 3.2 (the change problem)
b. Figure 4.2 (the social security problem)
c. Figure 5.4 (the encoding-decoding problem)

4. To check a program which is an external function (see Problem 3), one must provide an appropriate calling program, as well as data. Construct a test calling program and check data (if needed) for each of the following external functions:

a. Figure 6.12 (RAND.)
b. Figure 7.1*b* (SORT.)
c. Figure 7.3*b* (SEARCH.)

CHAPTER NINE

A PROGRAM TO PRODUCE PROGRAMS[1]

9.1 STATEMENT OF THE PROBLEM

ONE OF THE MOST interesting problems one can bring to the computer is the writing of its own programs. This is not so difficult as it may sound. We have already seen in Chapter 5 that by means of our language we have been able to translate sequences of characters into other sequences of characters. Writing a program is also the generation of sequences of characters (to make up statements), except that we need a rule (or algorithm) to determine which sequences to generate. Using an analogy, suppose we were in the tour department of a large automobile club, with the task of providing each driver with a set of marked maps (i.e., a program) showing his particular route in great detail. We cannot prepare any set of maps until we receive a specific request from someone indicating his origin, his destination, and whatever special conditions he wishes to impose, such as historical sites, and so on. We can prepare a set of standard procedures for preparing marked tour maps, however, so that as soon as a request is received, a map is generated. The standard

[1] The material in this chapter is drawn largely from an unpublished paper, Generation of Computer Programs by a Computer, by Robert M. Graham and Bernard A. Galler.

procedure is an *algorithm*. The result of applying it (i.e., the map) is again an algorithm—at a different level—telling the tourist how to proceed along his route.

In this chapter we shall consider a collection of problems, and for each problem there will be a corresponding program for its solution. We shall be interested in an algorithm for producing these programs. (The programs which will result in this way will therefore correspond to the marked tour maps.) After discovering this algorithm for producing programs, however, we shall go one step further. We shall write a program for this algorithm. This will be a program whose input is a problem description and whose output is a *program* to solve that problem.

The problems we shall be dealing with will be simple networks of switches, which we shall describe in more detail below. More specifically, the problem will be to determine whether or not a particular setting of the switches in a network will allow *flow* through the network. Such problems arise with networks which occur in electric circuits, railroad switchyards, irrigation canals, and so on. A typical network might look like Figure 9.1. Here *a*, *b*, *c*, and *d* are switches which are shown in *open* position, but which may be *closed*, also.

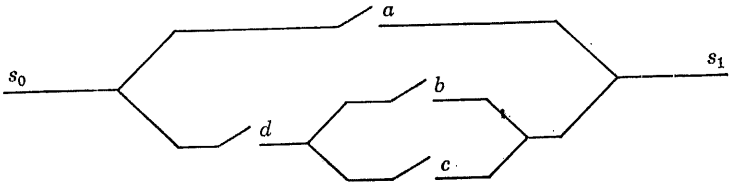


Figure 9.1

We shall say that there can be flow through a closed switch, but there cannot be flow through an open switch. It is clear that in Figure 9.1 there can be flow from s_0 to s_1 if and only if *a* is closed, or *d* is closed and either *b* or *c* (or both) is closed. We shall study such networks and the conditions under which there can be flow for each network. Although more sophisticated methods are available for dealing with such networks, we shall aim toward a simple, easily understood method.

How complicated will we allow the networks to become? Let us describe the collection M of networks which we wish to consider in the same way that we defined arithmetic expressions in Chapter 2.

1. We shall say that switches $s_1, s_2, \ldots, s_k$ are *connected in series* if the output from s_i is the input to s_{i+1} for $i = 1, 2, \ldots, k-1$. Switches connected in series may be pictured as in Figure 9.2. Any collection of switches connected in series will be a network in M.

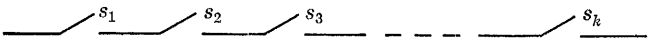


Figure 9.2

2. If G_1 and G_2 are networks, we shall say that they are *connected in parallel* if they have the same source of input and feed the same output channel. This can be pictured as in Figure 9.3*a*. We shall also say that G_1 and G_2 are *connected in series* if the output of one network is the input to the other. This can be pictured as in Figure 9.3*b*. Any network formed by connecting, in series or parallel, networks already in M will also be in M.

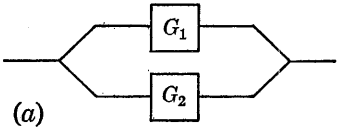


Figure 9.3*a*

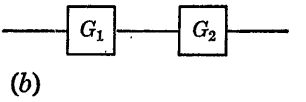


Figure 9.3*b*

3. The only networks in M are those arising in (1) and (2).

This defines the class M of networks that we shall be considering here. The reader should now verify that the network in Figure 9.1 is actually in M. For that network we earlier stated the conditions under which one could have flow: if *a* is closed, or *d* is closed and either *b* or *c* (or both) is closed. Let us write "A" for the statement "*a* is closed," "B" for the statement "*b* is closed," and so on, so that a value *true* for A implies that the switch *a* is closed, and thus there could be flow. Similarly, *false* corresponds to an open switch and, therefore, to no flow. Then, using the computer language which we have been developing, we may write the flow conditions for Fig-

ure 9.1 as the expression

A .OR. (D .AND. (B .OR. C))

Now, as soon as we know a particular setting of the switches, such as *a* open, *b* closed, *c* closed, and *d* open, we can determine whether or not there can be flow in the network.

9.2 BOOLEAN VARIABLES

In Chapter 2, we defined Boolean expressions in terms of *basic Boolean expressions.* A Boolean expression was obtained by combining basic Boolean expressions in various ways by means of the connectives .AND. and .OR. and the use of parentheses. The smallest unit that could occur on either side of a connective was the basic Boolean expression, which always consisted of a relation, such as .LE. (less than or equal) or .NE. (not equal), and an arithmetic expression on either side of that relation. Now we see that in this network problem it is very convenient for us to deal with single variables, such as A, B, C, and D, which represent entire statements and, therefore, have truth values of their own. It seemed quite natural to write, as we did above,

A .OR. (D .AND. (B .OR. C))

where we thought of A, B, C, and D as being either *true* or *false.* These variables are neither integer nor noninteger; their values are logical (or Boolean) values, and they are therefore called *Boolean variables.* Let us extend our previous definition of basic Boolean expression to include Boolean variables. Then the definition of the general Boolean expression need not be changed at all. The two definitions would now appear as follows:

A *basic Boolean expression* is either a Boolean variable or one of the relations $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ preceded and followed by any two arithmetic expressions.

1. Basic Boolean expressions and the Boolean constants *true* and *false* are Boolean expressions.

2. If $\mathcal{P}$ and $\mathcal{R}$ are already known to be Boolean expressions, then so are $(\mathcal{P})$, $\mathcal{P}$ .AND. $\mathcal{R}$, and $\mathcal{P}$ .OR. $\mathcal{R}$.

3. The only Boolean expressions are those which are generated by (1) and (2).

Now that we have introduced the Boolean variable, we face the problem of recognizing one when we meet it in a program. Just as we found that we had to declare integer variables (and later, statement-label variables), we shall now need to have a way to declare variables to be Boolean. Since we already have the statements

```
INTEGER N, J, GCD.
STATEMENT LABEL START, FINISH, GO
```

we shall introduce the new statement that is analogous to these:

```
BOOLEAN A, B, C, D, N1, N2
```

Now let us look at that last example more closely. The switch settings *a* open, *b* closed, *c* closed, *d* open produce the values A *false*, B *true*, C *true*, and D *false*. The computation which finds the truth value of the flow-condition expression given above should be clear if we go from left to right in Table 9.1. Since the final value is *false*,

Table 9.1

A	B	C	D	B .OR. C	D .AND. (B .OR. C)	A .OR. (D .AND. (B .OR. C))
False	True	True	False	True	False	False

we would decide that there can be no flow in that network for those switch settings. The complete truth table for the expression we have been using as an example appears in Table 9.2. The arrow indicates the row we have just computed as an example.

It is possible in theory to solve every problem of this kind by constructing the truth table, thus giving the behavior of the network for all possible switch combinations. Since there are 2^n rows in a truth table for n switches, however, it becomes impractical to generate the entire table for larger networks, but any particular combination of

switch settings can be used to evaluate the logical expression, as we did above. What we need, then, is a way to obtain easily the logical expression from a description of the network. Unfortunately, before we can discuss an algorithm for generating the logical expression, we must find a way to describe a network in a simple, but unambiguous way.

Table 9.2

A	B	C	D	A .OR. (D .AND. (B .OR. C))
True	True	True	True	True
True	True	True	False	True
True	True	False	True	True
True	True	False	False	True
True	False	True	True	True
True	False	True	False	True
True	False	False	True	True
True	False	False	False	True
False	True	True	True	True
False	True	True	False	False ←
False	True	False	True	True
False	True	False	False	False
False	False	True	True	True
False	False	True	False	False
False	False	False	True	False
False	False	False	False	False

9.3 NETWORK DESCRIPTIONS

It is easy to see that along with the switches, the structure of a network is given by the nodes, i.e., the points at which more than one line enters or leaves. In order to describe the way the switches in the network are connected, we may list with each switch the two nodes on either side of it. Figure 9.4 shows the network of Figure 9.1 with the nodes labeled n_1, n_2, n_3 and n_4. Each switch now has one node

immediately to its left and one node immediately to its right. We shall refer to these as the L node and the R node associated with that switch. If there are two switches connected in series (which does not happen to occur in this example), we will find it convenient to introduce a fictitious node between them. This will ensure that each switch will have its own L node and R node.

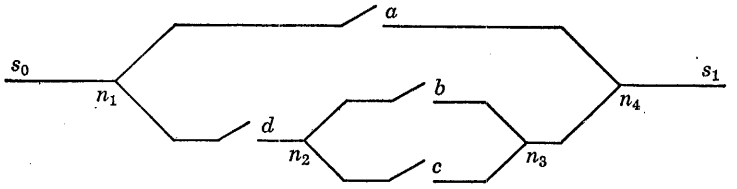


Figure 9.4

It is also possible for two nodes to have no switch between them, for example, nodes n_3 and n_4 in Figure 9.4. We could eliminate n_3 entirely and let n_4 be the R node for both *b* and *c*. But we would then have a branch point with no label. On the other hand, we could handle this situation by inserting a fictitious switch *z* between n_3 and n_4 and stipulate that this switch is always closed. In other words, the statement Z, which means "*z* is closed," is always *true*. If we do this, each branch point is a node, but we have increased the number of switches. Since none of these reasons seems to be compelling enough to force us to choose one method over the other, let us develop an algorithm which will allow both.

Just as we associated the name A with the switch *a* by letting A be the statement "*a* is closed," let us associate the name N*i* with the node n_i by letting N*i* be the statement, "There is a path through which there could be flow through n_i." Thus, from Figure 9.4, we see that N1 is *true*, and N2 is *true* if N1 is *true* and D is *true* (i.e., *d* is closed), and so on. In fact, the following statements describe the network in Figure 9.4 quite well:

N1 = *true*
N2 = N1 .AND. D
N3 = N2 .AND. (B .OR. C)
N4 = (N1 .AND. A) .OR. N3

The value (*true* or *false*) of N_4 is the answer to the problem as to whether or not there could be flow through the network. For the case we considered above, where A = *false*, B = *true*, C = *true*, and D = *false*, we would have, from the above equations,

N1 = *true*
N2 = *false*
N3 = *false*
N4 = *false*

so there would be no flow in the network.

It is interesting to observe that since N1 is always *true*, the expression N1 .AND. D is always *true* if D is *true* and always *false* if D is *false*. Thus, N1 .AND. D behaves exactly the same as D itself, as far as its truth value is concerned. We could, if we wished, write N2 = D in the above set of equations. Similarly, N1 .AND. A may be replaced by A alone, and we could write the whole set of equations as follows:

N1 = *true*
N2 = D
N3 = N2 .AND. (B .OR. C)
N4 = A .OR. N3

If we now substitute D for N2 in the N3 equation, and substitute the resulting expression for N3 into the last equation, we obtain

N4 = A .OR. (D .AND. (B .OR. C))

and we see that the right side is the expression with which we originally started (see Table 9.2).

Table 9.3

L node	Switch	R node
n_1	*a*	n_4
n_1	*d*	n_2
n_2	*b*	n_3
n_2	*c*	n_3
n_3	*z*	n_4

Returning now to the need for describing a network by means of its switches and nodes, we see that a complete description can be obtained by listing (in any order) all the switches and, with each switch, its L node and R node. Thus the network used as an example (Figure 9.4) would be described as in Table 9.3.

Now that we have a way to describe a network, we are in a position to discuss the algorithm which starts with the network description and generates the equations which we saw above. These equations were

N1 = *true*
N2 = N1 .AND. D
N3 = N2 .AND. (B .OR. C)
N4 = (N1 .AND. A) .OR. N3

We shall later find it useful to write the third equation in a different way. Table 9.4 shows that the expression

(N2 .AND. B) .OR. (N2 .AND. C)

has the same truth values as the expression

N2 .AND. (B .OR. C)

for all combinations of values of N2, B, and C, and we may therefore replace one expression by the other in such equations as we have.

Table 9.4

N2	B	C	(N2 .AND. B) .OR. (N2 .AND. C)	N2 .AND. (B .OR. C)
True	True	True	True	True
True	True	False	True	True
True	False	True	True	True
True	False	False	False	False
False	True	True	False	False
False	True	False	False	False
False	False	True	False	False
False	False	False	False	False

The equivalence of these two expressions with respect to their truth tables is an example of the *distributive law* as applied to Boolean connectives. The analogue in ordinary algebra is the law that allows substitution of the expression

$$x \cdot y + x \cdot z$$

for the expression

$$x \cdot (y + z)$$

It is interesting to observe that there is also in Boolean algebra a second distributive law (i.e., with .AND. and .OR. interchanged)

$$\text{P .OR. (Q .AND. R)} = \text{(P .OR. Q) .AND. (P .OR. R)}$$

which is valid for all values of P, Q, and R. The corresponding second law in ordinary algebra does not hold for all values of x, y, and z; i.e., it is *not* true that for all values of x, y, and z,

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

9.4 THE NETWORK ALGORITHM

If we now examine the properties of the equations

N1 = *true*
N2 = N1 .AND. D
N3 = (N2 .AND. B) .OR. (N2 .AND. C)
N4 = (N1 .AND. A) .OR. N3

we see that the first equation will always have the form N1 = *true*, if n_1 is the node closest to the source of the flow. We also observe that each equation involves on the *right side* only nodes which have already been represented in a previous equation on the *left side*, i.e., nodes for which we have already computed a value. Finally, each equation consists of one or more .AND. terms joined by the connective .OR., and each .AND. term represents a path in the network diagram which comes from the left into the node represented by the variable on the left side of the equation. (The form of the third equation indicates that it represents *two* paths coming into n_3 from the left.)

These properties will serve as a guide in our efforts to construct the equations from a description of the network.

One slight complication which we must keep in mind is that there are no rules as to how to number the nodes in the network; so we cannot assume any sequential ordering of the nodes. Moreover, we shall have to accept any listing of the switches and the L nodes and R nodes as a description of the network, regardless of the order in which they are listed. We could set down detailed rules for the person who uses these algorithms as to how nodes are to be numbered and how the switches are to be ordered in the network description. It is very desirable, however, to place as few restrictions as possible on the user. Not only do restrictions increase the chance of error (because of violations of these restrictions, if nothing else), but they make the use of an algorithm much less attractive. One stipulation we will make, nevertheless, is that the left-most node shall always be labeled n_1.

In what follows we shall have occasion to refer repeatedly to those nodes which have been represented on the left sides of previously generated equations. We shall call such nodes *computed nodes*. We have already remarked that each equation (after the first) in the set which we wish to construct has a node represented on the left side which is not yet a computed node and all nodes on the right side of the equation are computed nodes. Let us call the node represented on the left side of the equation the *target node*. Then all the nodes involved on the right side *immediately precede* the target node on various paths coming into the target node. As an example of this, we see in Figure 9.4 that n_1 and n_3 immediately precede n_4 on the two paths coming into n_4. Similarly, n_2 immediately precedes n_3 on both the paths coming into n_3. Thus, if we find all occurrences of the target node as an R node in the network description, the L nodes that occur on the same lines are the nodes which immediately precede the target node along the paths into the target node.

If n_k is the target node, and if n_k occurs on a line in the network description such as

$$n_j \qquad s \qquad n_k$$

then there can be flow through n_k if there can be flow through n_j and if the switch s is closed. Thus, we would expect to find, as part of

the right side of the equation for Nk, the expression

Nj .AND. S

This would be joined by the connective .OR. to all the other .AND. expressions arising in the same way. Unfortunately, if even one of the L nodes which become involved on the right side of the equation is not a computed node, the target node cannot be computed by this equation. We can construct an equation only for a target node which is not yet a computed node, but which has the property that all the nodes involved on the right side of the equation are computed nodes. In other words, whenever the target node occurs as an R node in the network description, the corresponding L node must be a computed node. We may now state the complete algorithm for constructing the appropriate set of equations:

1. Generate the equation

N1 = *true*

and label the node n_1 "computed."

2. In the network description, select as the target node the first node from the top in the R-node column, say n_k, which is not yet computed, but which has the property that for every occurrence of n_k as an R node, the corresponding L node is a computed node. (In Table 9.3, we would choose n_2, since only n_1 is computed so far.) If no target node can be found, i.e., if all nodes are computed, the process is completed.

3. For each line of the network description containing the chosen target node n_k as an R node, such as

n_j s n_k

form the expression Nj .AND. S.

Generate the equation having Nk on the left side and having on the right side all such .AND. expressions, joined by the connective .OR. . (From Table 9.3 we would generate the equation

N2 = N1 .AND. D

since there is only one occurrence of n_2 as an R node.) Label n_k "computed."

4. Repeat steps (2) and (3) until all nodes are "computed."

Applying this algorithm directly to the network which is deseribed in Table 9.3, we obtain

N1 = *true*
N2 = N1 .AND. D
N3 = (N2 .AND. B) .OR. (N2 .AND. C)
N4 = (N1 .AND. A) .OR. (N3 .AND. Z)

As we saw earlier, since Z is always *true*, we may replace

N3 .AND. Z

by N3, since this .AND. expression is *true* when N3 is *true* and *false* when N3 is *false*. If we do this, we see that we have our original equations.

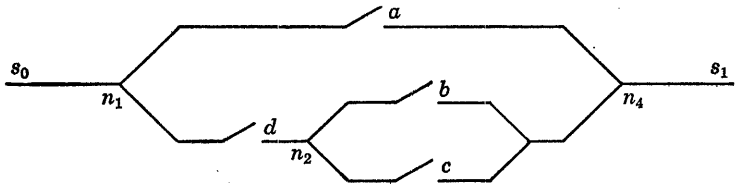


Figure 9.5

If we had chosen to eliminate the fictitious switch z from the network description by not labeling the node n_3 at all, we might have labeled the network as in Figure 9.5. The network would then be described as in Table 9.5.

Table 9.5

L node	Switch	R node
n_1	a	n_4
n_1	d	n_2
n_2	b	n_4
n_2	c	n_4

If we apply to this network the algorithm stated above for generating the equations, we obtain the following set of equations:

N1 = *true*
N2 = N1 .AND. D
N4 = (N1 .AND. A) .OR. (N2 .AND. B) .OR. (N2 .AND. C)

Again noting that N1 .AND. D may be replaced by D, we have N2 = D, and therefore

N4 = (N1 .AND. A) .OR. (D .AND. B) .OR. (D .AND. C)

and this in turn simplifies to

N4 = A .OR. (D .AND. (B .OR. C))

which agrees with the original equation.

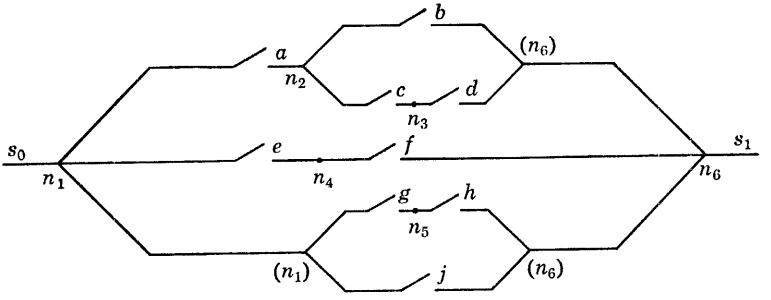


Figure 9.6

As a last example, we shall consider a more complicated network as shown in Figure 9.6. (Nodes labeled in parentheses have been eliminated in favor of the node carrying the same label without parentheses.) This network would have the description given in Table 9.6. Applying the algorithm to this network, we obtain

N1 = *true*
N2 = N1 .AND. A
N3 = N2 .AND. C
N4 = N1 .AND. E
N5 = N1 .AND. G
N6 = (N2 .AND. B) .OR. (N3 .AND. D) .OR. (N4 .AND. F)
.OR. (N5 .AND. H) .OR. (N1 .AND. J)

We may observe, as we did earlier, that by substituting from the first five equations into the last, and simplifying, we may obtain a single equation:

```
N6 = (A .AND. (B .OR. (C .AND. D))) .OR. (E .AND. F)
                                    .OR. (G .AND. H) .OR. J
```

This is not really of any concern to us, however, since the set of equations above furnishes a very feasible procedure for determining

Table 9.6

L node	Switch	R node
n_1	a	n_2
n_2	b	n_6
n_2	c	n_3
n_3	d	n_6
n_1	e	n_4
n_4	f	n_6
n_1	g	n_5
n_5	h	n_6
n_1	j	n_6

whether there can be flow or not. In fact, the only statements that are missing from the complete program to compute the possibility of flow in the network are (see Figure 9.7)

```
READ DATA
NORMAL MODE IS BOOLEAN
```

at the beginning and

```
PRINT RESULTS N6
END OF PROGRAM
```

at the end. The input data will consist of a complete set of switch settings for all the nodes in the network, and the result will be *true* or *false*, depending on the possibility of flow in the network with the given switch positions.

To summarize what has been accomplished so far, then, we may start with a network such as the one in Figure 9.6, described as in Table 9.6. Associated with this network is a *program*, shown in Figure 9.7. When this program goes to the computer, with it will go some data. For this program the data will consist of values (either *true* or *false*) for the Boolean variables A, B, C, D, E, F, G, H,

```
    READ DATA
    NORMAL MODE IS BOOLEAN
    N1 = true
    N2 = N1 .AND. A
    N3 = N2 .AND. C
    N4 = N1 .AND. E
    N5 = N1 .AND. G
    N6 = (N2 .AND. B) .OR. (N3 .AND. D) .OR. (N4 .AND. F)
1         .OR. (N5 .AND. H) .OR. (N1 .AND. J)
    PRINT RESULTS N6
    END OF PROGRAM
```

Figure 9.7

and J. Once a set of these values is read into the computer (because of the execution of the READ DATA statement), the statements evaluating N1, . . . , N6 will be executed, and finally the value of N6 will be printed out as the answer.

9.5 THE ALGORITHM FOR GENERATING PROGRAMS

The problem we have set for ourselves is to write a program whose output is the program in Figure 9.7. This program that we need should first generate as its output the READ DATA and NORMAL MODE statements, then follow the algorithm described above to generate the equations, and conclude by printing out the last two statements in Figure 9.7.

Let m be the number of rows in the network description. We shall store the *subscripts* of the L nodes as the values of a vector L. In other words, L(4) will be the subscript of the L node in the fourth row of the network description. In Table 9.6, L(4) = 3. Similarly,

we shall store as the values of a vector R the subscripts of the R nodes. Thus, R(2) will be the subscript of the R node in the second row of the network description. In Table 9.6, R(2) = 6. We shall also let S(1), . . . , S(m) be the (alphabetic) names of the Boolean variables associated with the switches. We then have for Table 9.6, m = 9, S(1) = A, S(2) = B, . . . , S(8) = H, and S(9) = J. The values of m, L(1), . . . , L(m), S(1), . . . , S(m), and R(1), . . . , R(m) will be read in as data by our equation-generating program.

We shall need an additional vector, which we shall call P. The vector P will be used to record which nodes are *computed nodes*. We shall set P(1), . . . , P(m) to zero at the beginning of the program, and whenever a node n_i becomes a computed node, we shall set P(i) = 1, so that it will always be possible to find out very quickly whether or not any particular node has been computed. The necessary equations will have been generated when the values in the P vector corresponding to nodes in the network description all have the value 1.

Figures 9.8 and 9.9 exhibit the flow diagram for the equation-generating program, while Figure 9.10 shows the program itself. Before we begin the more detailed discussion of these figures, however, there is a small point that should be considered.

We have been writing *true* and *false* for the values of Boolean variables, and although this is certainly all right for the present context, there is no provision in our computer language for small italicized letters. When we actually need to include these *Boolean constants* in statements in the language (such as in the statement N1 = *true*), we need a representation which is consistent with the kinds of characters which are available in our alphabet. Logicians often use "T" and "F" for *true* and *false*, respectively, but these letters would look too much like variable names. Sometimes logicians refer to these constants as 1 and 0, also, but these could be confused with the ordinary integers 1 and 0. Let us compromise on the following notation. For *true*, we shall write "1B", and for *false* we shall write "0B". Since a variable name can never start with a digit, these two Boolean constants cannot be confused with anything else. The statement

N1 = *true*

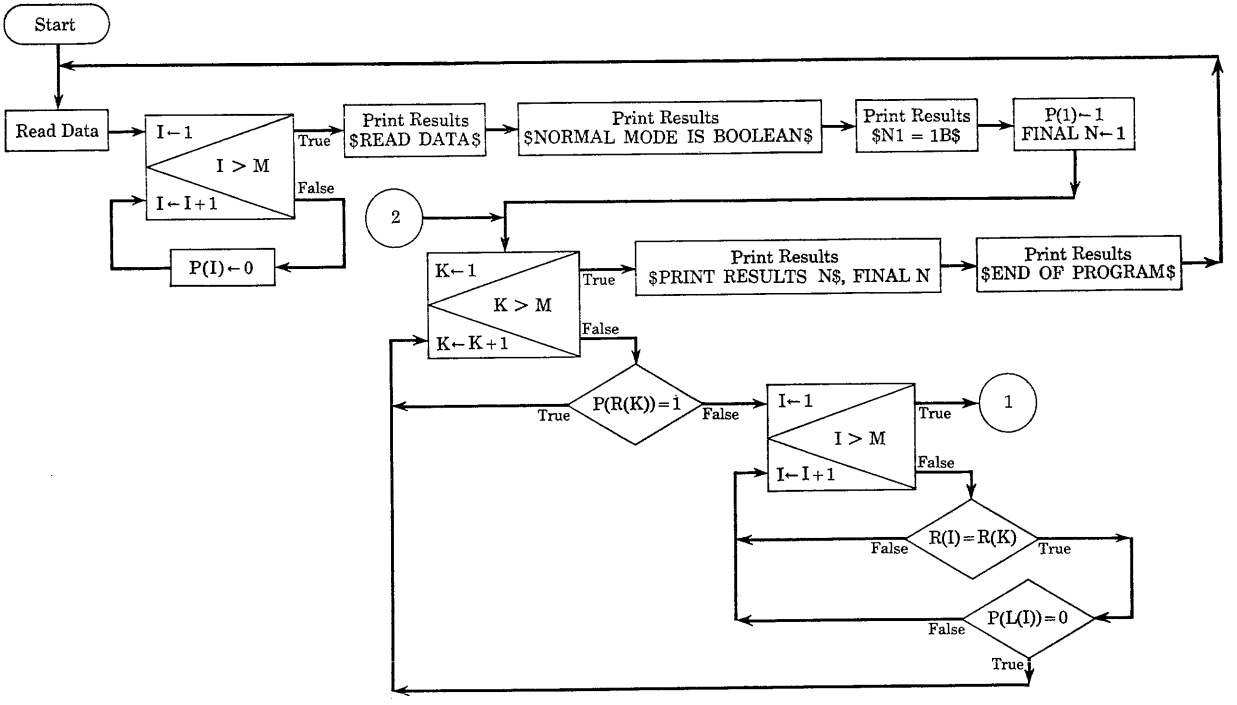

Start
Read Data
I←1
I > M
I←I+1
True
False
Print Results
$READ DATA$
Print Results
$NORMAL MODE IS BOOLEAN$
Print Results
$N1 = 1B$
P(1)←1
FINAL N←1
P(I)←0
2
K←1
K > M
K←K+1
True
False
Print Results
$PRINT RESULTS N$, FINAL N
Print Results
$END OF PROGRAM$
P(R(K))=1
True
False
I←1
I > M
I←I+1
True
False
1
R(I)=R(K)
False
True
P(L(I))=0
False
True

Figure 9.8

now becomes

$$N1 = 1B$$

This change has been incorporated into Figures 9.8 and 9.10.

If we look at Figure 9.8, we see that we begin by reading in the data, which consist of the values of m, L(1), . . . , L(m), S(1), . . . , S(m), and R(1), . . . , R(m). There follows a small loop which zeros the elements of the vector P. Then we generate the initial statements of the program which we are producing as the output of this program. Note that the alphabetic information between dollar signs constitutes the output. To this point we have produced the statements

```
READ DATA
NORMAL MODE IS BOOLEAN
N1 = 1B
```

The next box contains the substitution P(1) ← 1 to record the fact that n_1 is now a computed node and also records as the value of the variable FINAL N the subscript of the last node to be computed. This is necessary since the final PRINT RESULTS statement in the program we are producing must name the last node which was computed. (See Figure 9.7, where n_6 was the last node to be computed.)

We now enter the loop which recognizes the next node to be computed, i.e., the target node. Here we encounter for the first time a subscripted variable of the form P(R(K)). This means: "Take the integer which is the value of R(K) and use it as a subscript to find a value in the P vector." Let us see what meaning this has in our situation. The variable K will designate which row of the network description we are considering. Since R(K) is the subscript of the R node which occurs in the Kth row, P(R(K)) is the number in the P vector corresponding to that R node. Looking at Table 9.6, for example, if K = 2, then R(K) = R(2) = 6 (the subscript of n_6) and P(R(K)) = P(6). The Boolean expression in Figure 9.8

$$P(R(K)) = 1$$

implies that the R node in the Kth row is a computed node, and this

expression will be *true* if that node has been computed and *false* otherwise. Using the same example, if n_6 is a computed node, then P(6) will have the value 1; otherwise P(6) will have the value 0.

If we find that all R nodes have been computed, the equations will all have been generated, and we can finish by generating the

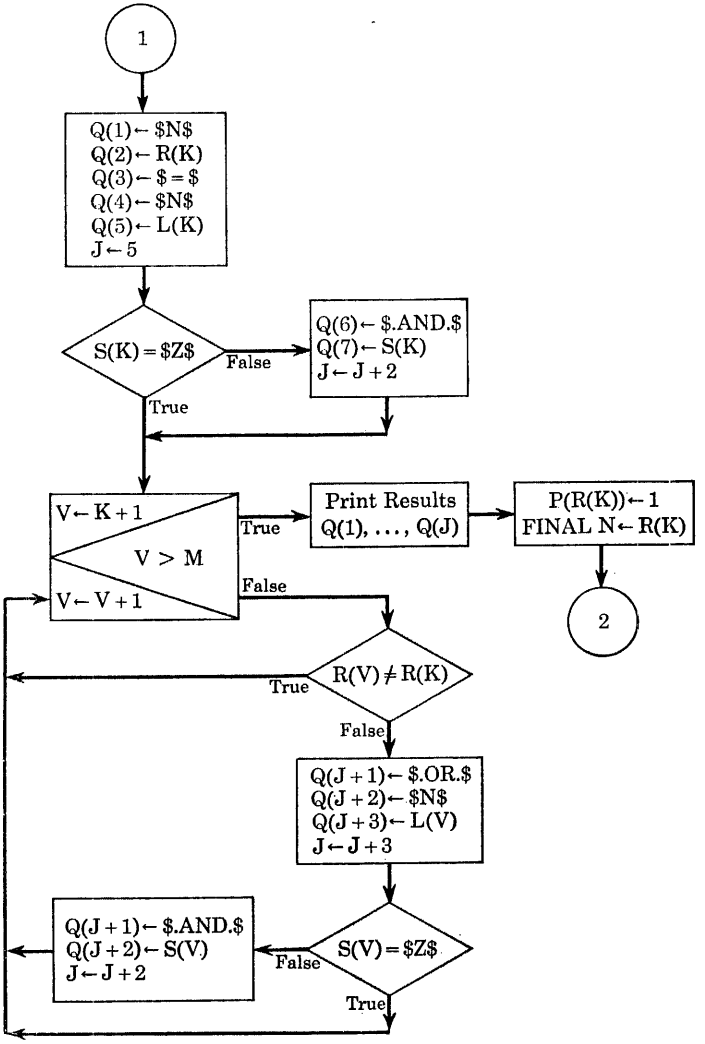


Figure 9.9

statements

PRINT RESULTS N6
END OF PROGRAM

if FINAL N = 6, for example. Assuming, however, that at least one node, say R(K), is not yet computed, we must now find all occurrences of this node as an R node and check to see whether the corresponding L nodes have been computed. If these L nodes have all been computed, then we are able to generate the equation which computes the value of the Boolean variable corresponding to this R node. If any one of the L nodes that is associated with this R node is not yet computed, we cannot generate the equation. If this happens, we must move to the next R node that is not yet computed, and so on. The lower half of Figure 9.8, then, consists of the K loop, which moves from one R node to another. If P(R(K)) is not 1 [so that the node with subscript R(K) is not yet computed], we move to the I loop, which looks at each row to see (1) if the R node is the R node we are considering, and (2) if it is, is the L node computed or not [i.e., is P(L(I)) = 0]? As soon as a row is found in which P(L(I)) = 0, so that the L node is not yet computed, we cannot construct the equation at all for R(K), and we leave the I loop altogether and return to increase K by 1 to search for another uncomputed node. If all the associated L nodes are computed, we exit in the normal way to the entry marked 1 in Figure 9.9 to generate the equation. At this time, we have a value for K which indicates the row whose R node is about to be computed. Everything that happens in Figure 9.9 is related to this R node and, therefore, to this value of K. After we generate the equation for this R node, we shall return to Figure 9.8 at entry 2. As an example, let us refer again to the network description given in Table 9.6 and assume that we have already produced the equations for N1, N2, . . . , N5 by going to Figure 9.9 for each one. Now we would discover that for K = 2, P(R(K)) = P(6) = 0, since n_6 is not yet a computed node. Since all the other nodes are already computed, we soon enter Figure 9.9 with K = 2 and R(K) = 6 and expect to produce at this time the equation

N6 = N2 .AND. B .OR. N3 .AND. D .OR. N4 .AND. F
.OR. N5 .AND. H .OR. N1 .AND. J

Note that parentheses are not necessary here, because of our convention that .AND. has a higher precedence ranking than .OR. .

We shall build up the symbols that make up the equation in a vector called Q and print out the equation only when it is complete, since we wish the whole equation to appear on the same line of the printed output. If we printed out each symbol or number when we determined that it was to be included in the equation, we would find them printed on different lines, since each PRINT RESULTS statement prints on a new line. We shall therefore collect the various symbols and numbers that are to occur in the equation and print them all out together. If we need the symbol "N" (e.g., as the first character in the equation), we must therefore put it into the Q vector. This can be accomplished by the substitution

$$Q(1) \leftarrow \$N\$$$

Similarly, when we will need an equal sign at a later point, we shall write

$$Q(3) \leftarrow \$=\$$$

A vector in which information is collected so as to be treated all together, for output or for some other purpose, is often called a *buffer*. Later we will determine that the entire equation has been constructed in the buffer, and we will be ready to print it out. The most efficient way to do this is to print only that part of the Q vector which this particular equation uses. In this way short equations will require the printing of only a few values of the Q vector. In order to know how far an equation extends into the Q vector, let us remember as the value of a variable J the last subscript used in the Q vector. Thus, when we have stored symbols or numbers in Q(1), . . . , Q(17), J will have the value 17. Moreover, whenever we add three more entries into the equation, we shall increase J by 3, and so on. At the end of the construction of the equation, we may simply write

PRINT RESULTS Q(1), . . . , Q(J)

to print it out.

Table 9.7 shows the contents of the Q vector and the value of J

Table 9.7

Q/1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	J
N	6	=	N	2																							5
N	6	=	N	2	.AND.	B																					7
N	6	=	N	2	.AND.	B	.OR.	N	3																		10
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D																12
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D	.OR.	N	4													15
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D	.OR.	N	4	.AND.	F											17
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D	.OR.	N	4	.AND.	F	.OR.	N	5								20
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D	.OR.	N	4	.AND.	F	.OR.	N	5	.AND.	H						22
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D	.OR.	N	4	.AND.	F	.OR.	N	5	.AND.	H	.OR.	N	1			25
N	6	=	N	2	.AND.	B	.OR.	N	3	.AND.	D	.OR.	N	4	.AND.	F	.OR.	N	5	.AND.	H	.OR.	N	1	.AND.	J	27

each time new entries are made during the part of the algorithm shown in Figure 9.9. The example used in this table is the equation for n_6 shown above.

We begin to construct the equation by putting the character "N" into Q(1), then the value of R(K) goes into Q(2), the equal sign goes to Q(3), another "N" goes to Q(4), and then Q(5) receives the number of the L node L(K). At this point J is set to 5, since Q(5) is the last number to have been placed in the Q vector. Whenever the switch in the Kth row is the fictitious switch z, we do not generate the ".AND. Z" part of the right side of the equation. This corresponds to the simplification made above, where we replaced

N3 .AND. Z

by the simpler expression

N3

The decision about z is made in the diamond-shaped box containing the expression

S(K) = Z

The lower half of Figure 9.9 is a large loop which generates the rest of the equation, if there is any. (The variable which controls the iteration is V here. It could be any variable that does not already have a value that is needed, such as K or J. A variable which controls only the loop and has no meaning outside it, such as V here, is often called a *dummy variable*.) In this loop we examine each row below the Kth row for additional occurrences of the node whose number is R(K). (In our example, we are looking for additional occurrences of n_6.) We check R(V) for each value of V to see if R(V) $\neq$ R(K). If this is *true*, we are not interested in R(V). If it is *false*, and we have found another occurrence, we add a few more entries to the Q vector, representing more symbols or numbers which are to appear in the equation, and increase J accordingly. (Again we watch out for the switch z. Table 9.6 does not illustrate this point, but Table 9.3, which contains z in its last row, would make use of this part of the algorithm when generating the equation for n_4.) After all the occurrences of R(K) have been handled in this way, and after the equation has been generated by the PRINT RESULTS state-

ment with the arguments Q(1), . . . , Q(J), we record the new status of the R node as *computed* by the substitution P(R(K)) ← 1. We also record as the new value of FINAL N the value of K, as indicated earlier, and return to Figure 9.8 to look for the next as-yet-uncomputed R node. Thus, as we indicated earlier, each time we enter Figure 9.9, we generate one equation and return to Figure 9.8.

When all nodes have been computed, we generate

PRINT RESULTS N

followed by the value of FINAL N, which adds to the statement the subscript of the last node to be computed. The END OF PROGRAM statement is then generated, and we have as our output a program such as the one in Figure 9.7.

Figure 9.10 exhibits the program which corresponds very closely to Figures 9.8 and 9.9. Only one point needs additional comment. The two separate questions that are asked within the I loop in Figure 9.8 have been consolidated into the one Boolean expression which appears in the statement labeled C3.

```
      R THIS SECTION CORRESPONDS TO FIGURE 9.8.
        NORMAL MODE IS INTEGER
START   READ DATA
        THROUGH C1, FOR I = 1, 1, I .G. M
C1      P(I) = 0
        PRINT RESULTS $READ DATA$
        PRINT RESULTS $NORMAL MODE IS BOOLEAN$
        PRINT RESULTS $N1 = 1B$
        P(1) = 1
        FINAL N = 1
C2      THROUGH C4, FOR K = 1, 1, K .G. M
        WHENEVER P(R(K)) = 1, TRANSFER TO C4
        THROUGH C3, FOR I = 1, 1, I .G. M
C3      WHENEVER R(I) .E. R(K) .AND. P(L(I)) .E. 0, TRANSFER TO
     1    C4
        TRANSFER TO C5
C4      CONTINUE
        PRINT RESULTS $PRINT RESULTS N$, FINAL N
        PRINT RESULTS $END OF PROGRAM$
        TRANSFER TO START
```

Figure 9.10

```
      R THE NEXT SECTION GENERATES THE EQUATIONS
      R AND CORRESPONDS TO FIGURE 9.9.
C5        Q(1) = $N$
          Q(2) = R(K)
          Q(3) = $=$
          Q(4) = $N$
          Q(5) = L(K)
          J = 5
          WHENEVER S(K) .NE. $Z$
              Q(6) = $ .AND. $
              Q(7) = S(K)
              J = J + 2
          END OF CONDITIONAL
          THROUGH C6, FOR V = K + 1, 1, V .G. M
          WHENEVER R(V) .NE. R(K), TRANSFER TO C6
          Q(J + 1) = $ .OR. $
          Q(J + 2) = $N$
          Q(J + 3) = L(V)
          J = J + 3
          WHENEVER S(V) .NE. $Z$
              Q(J + 1) = $ .AND. $
              Q(J + 2) = S(V)
              J = J + 2
          END OF CONDITIONAL
C6        CONTINUE
          PRINT RESULTS Q(1), . . . , Q(J)
          P(R(K)) = 1
          FINAL N = R(K)
          TRANSFER TO C2
          DIMENSION L(100), R(100), S(100), P(100), Q(500)
          END OF PROGRAM
```

Figure 9.10 (*Continued*)

PROBLEMS

1a. Carry out in detail the algorithm of Figures 9.8 and 9.9 and produce a program for the network whose description is given in Table 9.8 below.

b. Construct the network from the description in Table 9.8.

2. In Figures 9.8 and 9.9 an algorithm is presented which generates statements. In particular, the lower half of Figure 9.8 finds all occurrences as an R node of a given node. This is done in order to apply a test to the

corresponding L nodes. In the lower half of Figure 9.9, we again search all R nodes, looking for all occurrences of the same R node as before. This time we wish to generate the equation. It is rather wasteful to search all the R nodes a second time, however, and if some record were maintained of the occurrences of the R node during the first search (i.e., in Figure 9.8), the second search could be entirely eliminated. Modify the flow diagrams and the program so that each row (except row K) which contains the current R node as its R node is remembered in some new auxiliary vector. This vector can then be consulted in Figure 9.9 when one needs these rows again. (Do not erase this list for each new R node. Simply place new entries on top of old ones and keep a count of the length of the current list, just as we did in the Q vector.)

Table 9.8

L node	Switch	R node
n_1	a	n_2
n_1	b	n_2
n_2	c	n_4
n_2	d	n_4
n_1	e	n_3
n_3	f	n_4

3. Another way to handle the question raised in the preceding problem is to *chain* together the rows in which we are interested. For example, suppose rows 5, 7, 17, and 18 all had the target R node as their R nodes. Let us create a new vector, called C, and start the search in Figure 9.8. When the first of these four rows with the target R node is encountered (i.e., row 5), we set C(5) = 0 and C(0) = 5. When we come to the next such row (i.e., row 7), we set C(7) = 5 and C(0) = 7. At the seventeenth row, we set C(17) = 7 and C(0) = 17. Finally, we set C(18) = 17 and C(0) = 18. Note that whenever we encounter some row in this process, say row k, we set C(k) = C(0) and then C(0) = k. Thus, C(0) always remembers the last such row encountered, the C entry corresponding to that row remembers the one before that, and so on, until one of them contains zero [C(5) = 0 in this example]. The final status of the C vector for this example would be: C(0) = 18, C(5) = 0, C(7) = 5, C(17) = 7, and C(18) = 17. In Figure 9.9 one would simply start with C(0) and *unchain* by using row 18, then finding from C(18) that row 17 is next, and so on, each time watching for a C entry containing zero [i.e., C(5)] as the signal

to stop. Modify Figures 9.8, 9.9, and 9.10 to use this chaining procedure. This method would be especially good compared with the original algorithm given in Figures 9.8 and 9.9 if there are many nodes, but few occurrences of each one. Why?

4. Let us suppose that someone had been used to writing his simple conditional statements in a somewhat different form before he came across this book. In his method one would evaluate an arithmetic expression and would transfer to one of three specified statements, depending on whether the value of the expression was negative, zero, or positive. Thus, he might have written

```
IF (A − B + C * D) NEG, ZERO, POS
```

where the expression to be tested is A − B + C * D and NEG, ZERO, and POS are statement labels to which transfers are to be made. The word IF indicates the type of statement, just as we have used WHENEVER. *Using our language*, he can achieve the same effect now by writing either

```
WHENEVER A − B + C * D .L. 0, TRANSFER TO NEG
WHENEVER A − B + C * D .E. 0, TRANSFER TO ZERO
TRANSFER TO POS
```

or

```
WHENEVER A − B + C * D .L. 0
    TRANSFER TO NEG
OR WHENEVER A − B + C * D .E. 0
    TRANSFER TO ZERO
OTHERWISE
    TRANSFER TO POS
END OF CONDITIONAL
```

or some variation of these. Discover an algorithm which translates his form to ours. Be sure to allow extra parentheses in the arithmetic expression. [*Hint:* Since every left parenthesis must be matched by a right parenthesis, the end of the expression occurs when a right parenthesis appears which matches the *first* left parenthesis. You can keep a count (starting at zero) of left parentheses, raising it each time one is encountered and lowering it each time a right parenthesis appears. What must the count be when you reach the right end of the expression?] Construct a flow diagram and a program for your algorithm. Assume that there is a vector C(0), . . . , C(100) into which the IF statement will go when the READ DATA statement is executed. Assume, also, that the data are such that we will

have C(0) = I, C(1) = F, C(2) = $($, and so on. In other words, we may analyze the IF statement which is to be translated into our language by looking at one character at a time. Furthermore, assume that with the IF statement, as part of the data, is an integer N which indicates the total number of characters (counting parentheses, commas, etc.) in the IF statement. The IF statement may be expected to be free of errors and to contain no blanks.

CHAPTER TEN

SIMULTANEOUS LINEAR EQUATIONS

10.1 THE GEOMETRIC INTERPRETATION

SOLVING SETS of two, three, or four simultaneous linear equations has been a standard part of high school algebra courses for generations. Although such sets of equations may arise in a great many physical situations, a commonly used example was the mixture problem: What quantities of two solutions, one 95 per cent pure and the other 15 per cent pure, must be mixed to obtain 10 gallons of a 45 per cent pure solution? Now one hears reports of problems arising in the design of nuclear reactors which give rise to 25,000 equations involving 25,000 variables. These equations usually have some special properties, such as having no more than six or eight variables in each equation, and algorithms are developed to take advantage of these properties. We shall be concerned with the general problem, and we shall assume no special properties of the equations.

Problems involving simultaneous equations can be interpreted geometrically. The linear equation involving two variables x and y can be represented by a graph which turns out to be a straight line. In fact, this accounts for the name "linear" which is applied to an equation in which each variable appears with no higher exponent

than 1 and in which each term involves at most one variable. When we plot the graph of a linear equation, we are really plotting the set of points which satisfy the equation. If there are two variables, we plot the graph using two coordinate axes, and we obtain a *line*. If there are three variables in the equation, such as

$$3x + 4y + 5z = 60$$

we plot the graph using three coordinate axes, and we obtain a *plane*, part of which is shown in Figure 10.1. When there are more than three variables, we cannot plot the graph, but we continue to use terms which suggest the geometric interpretation. Thus, a linear

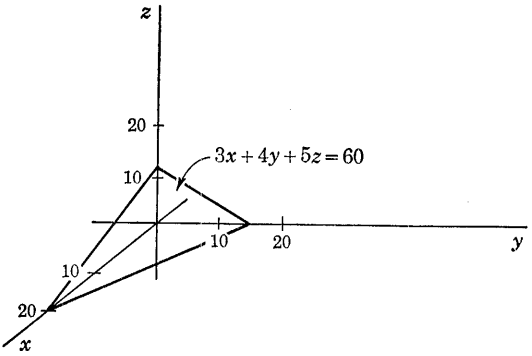


Figure 10.1

equation in more than three variables is said to be the equation of a *hyperplane*, with the prefix *hyper* used to indicate that there are too many variables involved to allow one to graph it as a plane.

When we have a set of linear equations to solve simultaneously, the geometric interpretation suggests that we are searching for a point (or collection of points) satisfying each of the equations, and thus lying on each of the lines or planes *at the same time*. If we have two equations in three variables, such as

$$\begin{aligned} x + y + z &= 1 \\ x + 4y - z &= 2 \end{aligned}$$

we are asking for the set of points [i.e., triples of numbers (x,y,z)] satisfying the two equations simultaneously. Since two planes intersect in a line (unless they are parallel), there are an infinite number of solutions, e.g., all the points on that line. Figure 10.2 illustrates the intersection of these two planes (with the line of intersection shown as a dashed line). If a third equation were added to the set, it would represent another plane. The set of points common to all

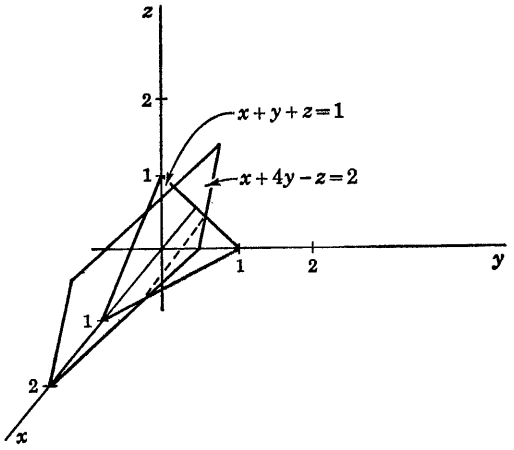


Figure 10.2

three planes would be the set of points lying both on the dashed line in Figure 10.2 and on the new plane. We would therefore be looking for the intersection of a line and a plane. This is generally a single point [i.e., a single triple of numbers (x,y,z)], which we then refer to as *the solution* of the set of three equations.

It could happen that the first two planes were parallel to each other, however, or if they did intersect in a line, that this line either lay entirely in the third plane or was parallel to it. In either of these unusual cases, we would not have a single point as the solution. We would either have no solution (if two of the planes were parallel, such as planes b and c in Figure 10.3), or we would have an infinite number of solutions (if the third plane contained the line of intersection of the first two planes, as in Figure 10.4). In the rest of this

chapter, we shall be developing an algorithm for the solution of simultaneous linear equations. We shall have to provide for some

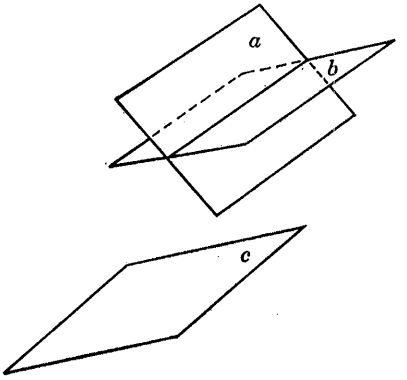


Figure 10.3

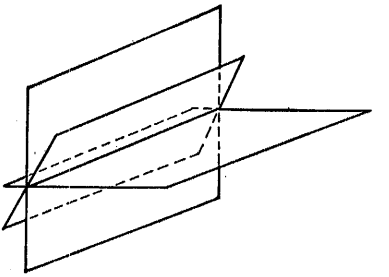


Figure 10.4

part of the algorithm to recognize these cases, usually referred to as *degenerate* cases.

10.2 ALGORITHMS FOR SIMULTANEOUS LINEAR EQUATIONS

In elementary algebra courses we usually learn several algorithms which are quite suitable for sets of two or three equations. One may use graphical techniques actually to plot the lines or planes involved, or one may use algebraic techniques. One common algebraic method is sometimes called "substitution," in which one solves for a variable, say z, in one equation and substitutes the resulting expression into each of the remaining equations, thereby eliminating z and at the same time reducing the number of equations by one. If there are two or more equations remaining, the process is repeated with some other variable. For example, given the set

$$\begin{aligned} x + y + z &= 1 \\ x + 4y - z &= 2 \\ 4x - y + 2z &= 5 \end{aligned}$$

we might solve for z in the first equation,

$$z = 1 - x - y$$

and substitute this into each of the other equations, obtaining

$$\begin{aligned} 2x + 5y &= 3 \\ 2x - 3y &= 3 \end{aligned}$$

Now, solving for x in the first equation,

$$x = -\tfrac{5}{2}y + \tfrac{3}{2}$$

we substitute into the second equation, obtaining

$$-8y = 0$$

so the value of y in the solution is 0. Since $x = -\frac{5}{2}y + \frac{3}{2}$, we have $x = \frac{3}{2}$, and since $z = 1 - x - y$, it follows that $z = -\frac{1}{2}$. Thus, the solution point is $(\frac{3}{2},0,-\frac{1}{2})$. (This process of finding the other solution values once one of them is known is called the *back solution.*) We shall refer to this example several times as we develop other methods.

Another method, sometimes called "addition and subtraction," consists in adding (or subtracting) multiples of certain rows to (or from) other rows so as to eliminate one or more variables. Using the same set of equations as above, this method would proceed as follows:

Given the equations

$$\begin{aligned} x + y + z &= 1 \\ x + 4y - z &= 2 \\ 4x - y + 2z &= 5 \end{aligned}$$

we subtract the first equation from the second, and we also subtract four times the first equation from the third equation. This leaves the following set of equations:

$$\begin{aligned} x + y + z &= 1 \\ 3y - 2z &= 1 \\ -5y - 2z &= 1 \end{aligned}$$

Subtracting the second equation from the third now leaves

$$\begin{aligned} x + \ y + \ z &= 1 \\ 3y - 2z &= 1 \\ -8y \qquad &= 0 \end{aligned}$$

From the third equation we now have $y = 0$. From the second equation of the last set, we then have $z = -\frac{1}{2}$, and from the first equation $x = \frac{3}{2}$.

There is one assumption which has been made in this method and which should be made explicit. We have several times modified an equation in a set of simultaneous linear equations by adding to it or subtracting from it a multiple of another equation in the set. The assumption is that *the set of solution values does not change.* More explicitly, suppose now that all equations are written with all non-zero terms on the left side. Then, if we are dealing with three equations as an example, we may write our set of equations in the following way:

$$\begin{aligned} P(x,y,z) &= 0 \\ Q(x,y,z) &= 0 \\ R(x,y,z) &= 0 \end{aligned}$$

or, using P, Q, and R as abbreviations,

$$\begin{aligned} P &= 0 \\ Q &= 0 \\ R &= 0 \end{aligned}$$

The assumption is that the set of equations

$$\begin{aligned} P &= 0 \\ Q &= 0 \\ R - kP &= 0 \end{aligned}$$

has the same set of solutions as the original set of equations, for every constant k.

Since this assumption is basic to everything that follows, let us see how it may be justified. What we must argue is that every solution of the original set of equations is also a solution of the modified

set of equations, and every solution of the modified set is a solution of the original set. Suppose, then, that (x_0,y_0,z_0) is any solution of the original set of equations, i.e.,

$$\begin{aligned} P(x_0,y_0,z_0) &= 0 \\ Q(x_0,y_0,z_0) &= 0 \\ R(x_0,y_0,z_0) &= 0 \end{aligned}$$

Since $R(x_0,y_0,z_0) - kP(x_0,y_0,z_0) = 0 - k \cdot 0 = 0$, we see that (x_0,y_0,z_0) is also a solution of the modified set of equations. On the other hand, if (x_1,y_1,z_1) is any solution of the modified set of equations, then

$$\begin{aligned} P(x_1,y_1,z_1) &= 0 \\ Q(x_1,y_1,z_1) &= 0 \\ R(x_1,y_1,z_1) - kP(x_1,y_1,z_1) &= 0 \end{aligned}$$

It is clear, then, that

$$R(x_1,y_1,z_1) = R(x_1,y_1,z_1) - kP(x_1,y_1,z_1) = 0$$

so the original set of equations is satisfied by (x_1,y_1,z_1) as well. We see therefore that this operation on sets of simultaneous equations does not alter the set of solution values.

10.3 THE JORDAN ALGORITHM

We are now faced with the problem of selecting a good algorithm for machine computation. Of the many available, we shall develop in some detail the Jordan method, which is based on the addition and subtraction method discussed above. If we examine the illustration used earlier with the addition and subtraction method, we see that an attempt was made to pick out coefficients that were *convenient.* Thus, although x was eliminated first (from the second and third equations), we next eliminated z. The coefficients of z were equal, thus making it easy to eliminate z by subtraction. Such scanning of coefficients to find convenient ones is simple for our eyes, but quite difficult to organize into a computer algorithm, especially since we would have to describe operationally what "con-

venient coefficients" were. Actually, all this scanning for convenient coefficients is motivated by our desire to reduce the effort involved in hand computation. Since computers do arithmetic on large numbers or numbers with fractional parts just as easily as on small integers, a great deal of this motivation now disappears. We shall find it much more worthwhile to obtain a simple algorithm which requires as little searching for special numbers as possible than to look for a more complicated algorithm which attempts to find "convenient coefficients."

In order to free ourselves from notation which suggests only two or three variables, let us restate the problem, using a more general notation. Starting with a set of n equations in n variables $x_1, \ldots,$ x_n, we shall write the equations as follows:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= a_{1,n+1} \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= a_{2,n+1} \\
\cdots\cdots\cdots\cdots\cdots&\cdots\cdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= a_{n,n+1}
\end{aligned}$$

(Note that the comma used as part of the subscript on the right side of the equations is there only to help keep the complicated subscript easy to read. It is not needed with simpler subscripts, and when we have a particular value for n, say $n = 3$, we shall write $a_{n,n+1}$ as a_{34}, without the comma.) In this notation, when we write a_{23}, we mean the coefficient of x_3 in the second equation. Thus, in the example used above

$$\begin{aligned}
x + y + z &= 1 \\
x + 4y - z &= 2 \\
4x - y + 2z &= 5
\end{aligned}$$

where $n = 3$, we would have $a_{11} = a_{12} = a_{13} = a_{14} = a_{21} = 1$, $a_{22} = 4$, $a_{23} = -1$, $a_{24} = 2$, and so on. The array of numbers

$$\begin{bmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & a_{1,n+1} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & a_{2,n+1} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & a_{n,n+1}
\end{bmatrix}$$

is usually called the *matrix of coefficients* of the set of simultaneous linear

equations. The matrix of coefficients of the three equations used as an illustration above is

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & -1 & 2 \\ 4 & -1 & 2 & 5 \end{bmatrix}$$

The reason we are interested in this matrix of coefficients is that almost all the important information needed to solve the equations is contained in this array. The only information not contained here is the set of labels x, y, and z to be attached to the first, second, and third columns, respectively. (We shall also have to remember that there really is an equal sign in front of the right-most column.) Because so much of the structure of the problem is contained in the matrix of coefficients, our algorithm will operate entirely on this matrix. The names of the variables may be attached to the solution values at the very end.

We note that certain operations on the array (which we shall call *elementary operations*) may be performed without changing the set of solutions to the original equations. For example, we may multiply (or divide) each number in a row of the array by any constant (except that we cannot divide by zero). We may do this because each row represents an equation, and performing this operation on one of a set of simultaneous equations does not change the set of solution values. Similarly, we may interchange any two rows without changing the solution. We may even interchange any two columns except the right-most one, *provided we also interchange the names of the variables attached to these columns.* Finally, one of the most important operations which we shall need, and which does not change the solution, is adding (or subtracting) a constant multiple of one row to (or from) another row. This is the operation on the array which corresponds to the basic operation of the addition and subtraction method discussed in Section 10.2.

Moving now to the actual algorithm, we might ask first: "Given the elementary operations which may be performed on the array without changing the solution values, how shall we best perform these operations so as to be able to discover the solution? Each elementary operation transforms the array into a new array. Is there some *best*

array toward which we should aim?" Suppose we actually had the solution values

$$\begin{aligned} x_1 &= b_1 \\ x_2 &= b_2 \\ &\cdots \\ x_n &= b_n \end{aligned}$$

Since these equations can be written as

$$\begin{aligned} 1 \cdot x_1 + 0 \cdot x_2 + \cdots + 0 \cdot x_n &= b_1 \\ 0 \cdot x_1 + 1 \cdot x_2 + \cdots + 0 \cdot x_n &= b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ 0 \cdot x_1 + 0 \cdot x_2 + \cdots + 1 \cdot x_n &= b_n \end{aligned}$$

we have as the matrix of coefficients of this set of equations

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & b_1 \\ 0 & 1 & \cdots & 0 & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & b_n \end{bmatrix}$$

If we could transform the original matrix of coefficients into this final form, we could read the solution directly from the matrix. (If we had for some reason interchanged some columns, we would have to do some matching of names with solution values, but we shall see later that this is a very straightforward process.) Note that the ones in the final array are the values of the array entries $a_{11}, a_{22}, a_{33}, \ldots, a_{nn}$. This set of entries is called the *main diagonal*. We shall set as our objective, then, the transformation of the original array into this final form, which has ones on the main diagonal, the solution values in the right-most column, and zeros everywhere else. If we can do this, no back solution will be necessary, since each solution value may be read directly from the array.

Let us choose as our starting point the upper left corner, i.e., the entry a_{11}. If we divide the entire first row by a_{11}, we will already have the desired 1. (We shall be applying the elementary operations freely now, without always calling attention to their use.) We shall have to avoid division by zero here, but for the time being, let us ignore such complications. The next step is to obtain zeros in the

rest of the first column and/or the rest of the first row (except possibly the right-most entry). Since the operation available to us for modifying entries in the array deals with entire rows, it would not be very easy to work within the first row. We may, however, use it to modify other rows, just as we did in Section 10.2 when we used the first equation to eliminate x from the second and third equations. Eliminating x there corresponds exactly to obtaining zero entries in all but one position in the first column of the matrix of coefficients. Just as we subtracted four times the first equation from the third equation, we now subtract a_{21} times the first row from the second, a_{31} times the first row from the third row, and so on. (In that example, $a_{21} = 1$ and $a_{31} = 4$.) Each time we do this, all the entries in the rows being modified will probably change, but this should not concern us, since we obtain the desired zeros in the first column. If we were to apply the process as we have so far described it to the array of the example, we obtain the array

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 3 & -2 & 1 \\ 0 & -5 & -2 & 1 \end{bmatrix}$$

In this method we have so far used one special entry (a_{11}) in one special row (the first row). We shall now want to do similar operations with other rows and other special entries in those rows. The row being used to modify other rows will be referred to as the *operating row*, and the special entry in that row, which will always turn out to be the entry on the main diagonal, will be called the *pivot entry*.

We are now ready to move to the second column. The first thought might be to use the first row as the operating row again in order to obtain the zeros one needs in the second column. Unfortunately, any attempt to use the first row to modify others now will probably introduce nonzero entries into the first column again. If we use any other row, however, the zero we now have created in the first column of that row will not modify any of the other numbers in the first column. The simplest rule is to move to the second row and divide this row by the (current) value of a_{22} to obtain a 1 as the value of the pivot entry. (The values of the second row entries in

the illustration now become

$$0 \quad 1 \quad -\tfrac{2}{3} \quad \tfrac{1}{3}$$

since the pivot entry a_{22} had the value 3.) Then we may use the second row as the operating row and a_{22} as the pivot entry to modify every other row (including the first row) so as to obtain zeros everywhere in the second column except at the pivot entry. Again referring to the illustration, we see that 1 times the second row is subtracted from the first row, and -5 times the second row is subtracted from the third row. The illustration matrix now becomes

$$\begin{bmatrix} 1 & 0 & \tfrac{5}{3} & \tfrac{2}{3} \\ 0 & 1 & -\tfrac{2}{3} & \tfrac{1}{3} \\ 0 & 0 & -16\tfrac{2}{3} & \tfrac{8}{3} \end{bmatrix}$$

If we apply the same procedure to the third row of this illustration matrix, we shall divide the third row by $-16\frac{2}{3}$, obtaining

$$\begin{bmatrix} 1 & 0 & \tfrac{5}{3} & \tfrac{2}{3} \\ 0 & 1 & -\tfrac{2}{3} & \tfrac{1}{3} \\ 0 & 0 & 1 & -\tfrac{1}{2} \end{bmatrix}$$

We then subtract $\frac{5}{3}$ times the third row from the first row and $-\frac{2}{3}$ times the third row from the second row. We then obtain the following array

$$\begin{bmatrix} 1 & 0 & 0 & \tfrac{3}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\tfrac{1}{2} \end{bmatrix}$$

from which we may read the solution $x = \frac{3}{2}$, $y = 0$, $z = -\frac{1}{2}$, by inserting the equal sign and attaching the appropriate names to the columns.

Still ignoring the question of division by zero, let us describe the procedure we have developed in terms of the names of the array entries. For example, dividing an entire row by a number requires a small loop. The variable of the iteration (j) will be used as the second subscript of the array entry, since this subscript designates the

column in which the entry occurs. To divide the first row by a_{11}, for instance, we might write the flow diagram as in Figure 10.5.

There is a rather subtle error in Figure 10.5, however. To understand this error, one should actually carry out the computation for a typical first row of an array. This means that a_{11} should have a value different from 1, since $a_{11} = 1$ is not the typical case. The trouble with Figure 10.5 is that for $j = 1$, a_{1j} is really a_{11}. The very first division the first time around the loop will compute a new value (e.g., 1) for a_{11}. In every iteration of this loop after the first, we shall use the current value of a_{11} in the division, i.e., $a_{11} = 1$. Thus, the effect of the diagram in Figure 10.5 is to set a_{11} equal to 1 and

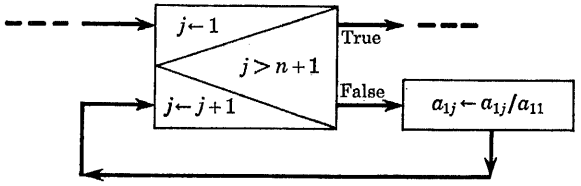


Figure 10.5

leave every other number unchanged. As an illustration of this behavior, let us examine in detail what would happen to a typical first row of some array, such as

$$10 \qquad 3 \qquad -2 \qquad 10$$

if we used the method of Figure 10.5. For $j = 1$, we compute

$$a_{11} = a_{11}/a_{11}$$

or

$$a_{11} = 1$$

The value of a_{11} is now 1, and this will be the value used in the computation that follows. For $j = 2$, we have

$$a_{12} = a_{12}/a_{11}$$

or

$$a_{12} = \tfrac{3}{1}$$

so that

$$a_{12} = 3$$

which defeats our purpose in dividing by the pivot entry.

There are two ways out of this dilemma. One way would be to save the initial value of a_{11} somewhere else and divide every number in the row by it. This method is shown in Figure 10.6. Another method, slightly shorter in execution time, simply starts dividing the

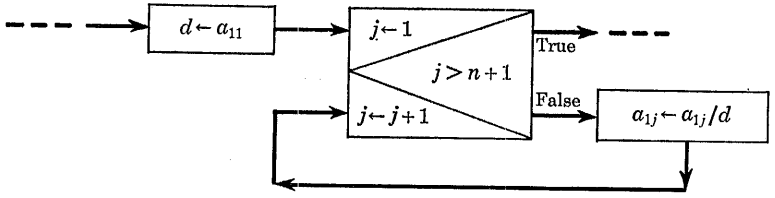


Figure 10.6

numbers at the right end of the row and moves toward the left. In this way, the division which destroys a_{11} is now performed during the *last* iteration, and we are not in the position of using the new value (e.g., 1) for any further divisions. This method, which we shall use, is shown in Figure 10.7.

There is another point to be made about this particular loop, also. Dividing the operating-row entries by the pivot entry is an operation which we shall perform each time we move to another column. Remember that in each case we shall move to a new operating row

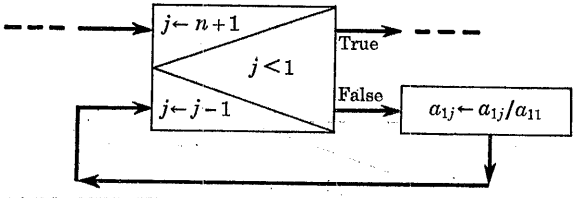


Figure 10.7

with its own pivot entry on the main diagonal. It is the value of this pivot entry which is to be used as the divisor when going through the loop in order to obtain a 1 in the pivot position. We will, moreover, have zeros to the left of the pivot entry, and there is no use dividing these zeros by the pivot entry. The division loop might just as well start from the right end and move to the left as in Figure 10.7, but stop after dividing the pivot entry by itself. If we are working on

the third row, for example, we shall let j start with the value $n + 1$ and end with the value 3, so that the first value divided by the pivot entry a_{33} will be $a_{3,n+1}$ and the last will be a_{33}. In general, if we are working on the kth row, we shall let j decrease from $n + 1$ to k. This is shown in Figure 10.8.

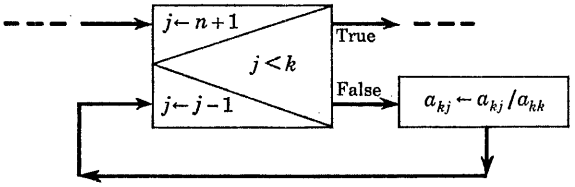


Figure 10.8

The other important part of this simultaneous-linear-equations algorithm is the modification of a row by subtracting from it a multiple of the operating row. If, for example, the operating row is the second row, and we are going to modify the entries in the third row, then each of the entries in the third row will have something subtracted from it. We have already seen that what we subtract from the third row is a_{32} times the second row (so that the new value of a_{32} will be zero). Therefore, each entry in the third row will have a_{32} times the corresponding entry in the second row subtracted from it. Again, we write the diagram in Figure 10.9 for this loop, and

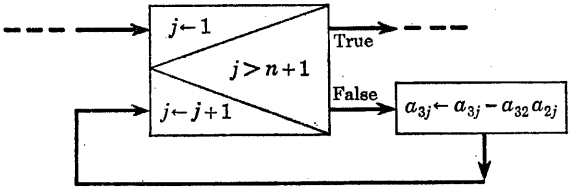


Figure 10.9

we discover an error in it. Just as we destroyed the divisor in Figure 10.5 during the first time around the loop, we now destroy the value of a_{32}, except that here a_{32} is set to zero. In the example we have

been considering, we had the following array at one stage:

$$\begin{bmatrix} 1 & 0 & \frac{5}{3} & \frac{2}{3} \\ 0 & 1 & -\frac{2}{3} & \frac{1}{3} \\ 0 & -5 & -2 & 1 \end{bmatrix}$$

The next step would be to use the second row to modify the third row to obtain a zero as the value of a_{32}. Using the algorithm of Figure 10.9, and starting with $j = 1$, we have $a_{31} = a_{31} - a_{32} \cdot a_{21} = 0 - (-5)(0) = 0$, and then $a_{32} = a_{32} - a_{32} \cdot a_{22} = (-5) - (-5)(1) = 0$. Now a_{32} has the value zero, and this is the value that will be used in the next computation:

$$a_{33} = a_{33} - a_{32} \cdot a_{23} = (-2) - (0)(-\tfrac{2}{3}) = -2$$

which is incorrect. Again, to eliminate this difficulty either we may save the value of a_{32} or we may start from the right end of the row

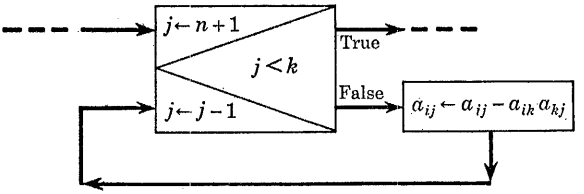


Figure 10.10

(i.e., with $j = n + 1$) and modify a_{32} last. We shall choose the second alternative again. We may ask now if some simplification such as stopping with the column containing the pivot entry will work here as it did earlier. It will in fact work here, also, since the entries in the operating row to the left of the pivot entry (if any) are all zero, and we are therefore not subtracting anything from the entries in these columns in the row being modified. If we let the kth row be the operating row and the ith row be the row being modified, we are now led to the diagram in Figure 10.10. Notice that in Figure 10.10 the number a_{ik} is that entry in the ith row (the row being modified) which is in the same column as the pivot entry a_{kk}. Figure 10.10 should be compared with Figure 10.9, in which $i = 3$ and $k = 2$.

We shall continue to refer to the operating row as the kth row. The column being cleared to zeros (except in the pivot position) will automatically be the kth column. The row being modified will continue to be called the ith row, and the variable that indicates the column subscript (as in Figures 10.8 and 10.10) will be j. The general algorithm (still ignoring division by zero) is shown in Figure 10.11. We begin by bringing in the data for the problem, e.g., the number of equations n and the matrix of coefficients a_{11}, a_{12}, . . . , $a_{1,n+1}$, a_{21}, a_{22}, . . . , $a_{2,n+1}$, a_{31}, . . . , $a_{n,n+1}$. Now we begin the overall loop on the variable k, which is going to indicate at the same time (1) the operating row (the kth row), (2) the pivot entry a_{kk}, and (3) the column being cleared to zeros (except for the pivot entry). After the iteration in which k has the value n, the matrix will have been transformed into the final form mentioned above, in which there are 1s on the main diagonal, the solution values in the right-most column, and zeros everywhere else. Since this algorithm does not involve interchanging any rows or columns, we may simply print out the right-most column exactly in order, and these numbers will be the solution values. As indicated in Figure 10.11, these array entries are $a_{1,n+1}$, $a_{2,n+1}$, . . . , $a_{n,n+1}$. In the example used above as an illustration, with $n = 3$, these would be the numbers $a_{14} = \frac{3}{2}$, $a_{24} = 0$, and $a_{34} = -\frac{1}{2}$.

Once k has received a value and we are within the scope of the overall iteration based on k, we have selected an operating row, a pivot entry, and a column to be cleared to zeros. The first task now is to divide the entries in the operating row by a_{kk} to obtain a 1 in the pivot position. We have already discussed the algorithm needed to do this, and the next part of Figure 10.11 is just the loop shown in Figure 10.8. Once this loop is completed, we come to that part of the algorithm which uses the operating row to modify each of the other rows. (We should note that it does not modify itself.) We now need a loop which will move us from one row to the next, allowing us each time to modify the current row. If we call the row to be modified the ith row, this means that we need a loop with iteration variable i. Whenever i receives a value and we move into the scope of this iteration, we immediately ask whether $i = k$. If so, we go back and ask for the next value of i, since the kth row (i.e., the operating row) is the one row we do not wish to modify. If $i \neq k$,

we have the modification loop discussed above, and the part of the diagram that follows is the same loop that appeared in Figure 10.10. After i has run through its range of values, we have cleared one entire column, and it is time to move to the next operating row.

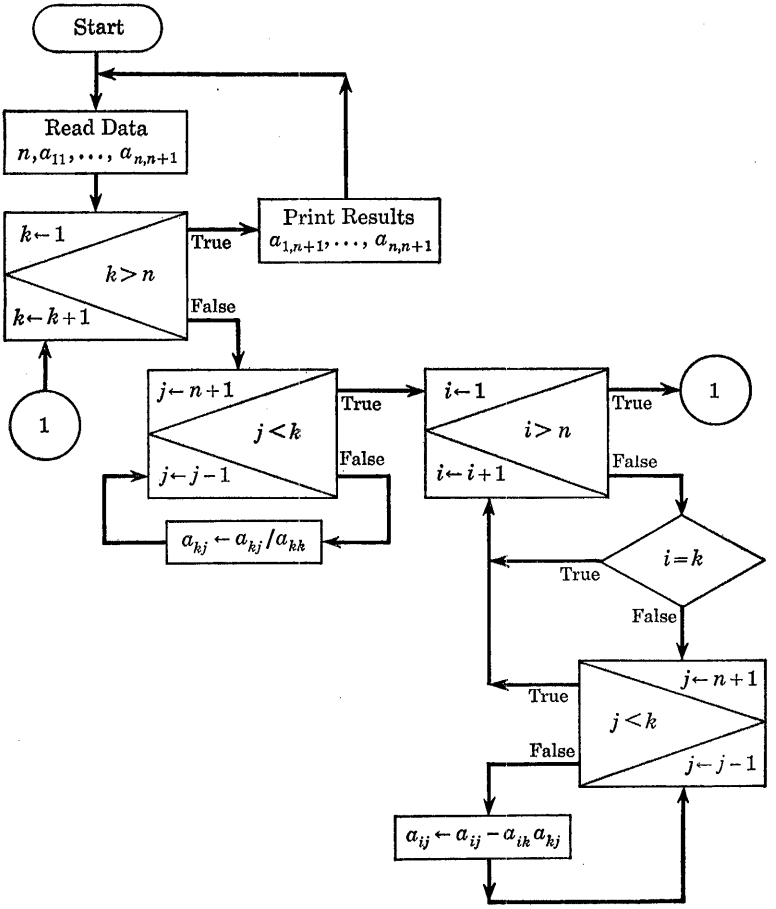


Figure 10.11

We therefore return to the overall loop and increase k so that the process can begin again.

It would be a good idea to do a careful computation on some set of simultaneous linear equations using this algorithm before going on.

Set aside some fictitious computer storage locations called i, j, and k, label the matrix of coefficients a_{11}, a_{12}, . . . , $a_{n,n+1}$, and follow the algorithm in every detail.

10.4 THE DIMENSION STATEMENT FOR ARRAYS

We should now be able to write the program for the Jordan algorithm, except that there is no provision in our language for writing more than one subscript on an element of a vector. When we introduced vectors in the discussion of the change problem, we always used one subscript [for example, Q(25)], and the dimension declaration indicated the highest subscript that might occur in the computation, thereby determining the amount of storage to be set aside for the vector. Thus, we wrote

DIMENSION Q(50)

which indicated that 51 storage locations should be set aside for Q(0), Q(1), . . . , Q(50). Now we see that sometimes it is necessary to describe the entries in this block of storage as a rectangular array with two subscripts on each entry. In some problems it is necessary to put three or more subscripts on each entry, thus subdividing the block of storage even further.

One example of the use of several subscripts is the problem of allocating shipping facilities, in which x_{ipwcm} might represent the amount of item i to be shipped from plant p via warehouse w to customer c in month m. In this shipping problem it is actually very convenient to be able to change the number of subscripts the entries will have each time the program is run on the computer. For example, if an analysis is to be done for only one item during one month, there is no need to carry the first and last subscripts along. In this case, one would want to use only three subscripts. Then the next set of data might need subscripts representing the item and the month, but it might not have any provision for warehouses, and the problem would involve only the four subscripts i, p, c, and m. In order to write a program which could accept as part of the data the number of subscripts to be used, we must have a very flexible dimensioning procedure.

Another bit of flexibility which is useful is the ability to change the number of rows (or columns) at any time. For example, in the shipping problem it might develop during the computation that 12 months is too long a period to consider. The number of months should be reduced, perhaps to 6. This means that the highest subscript to be used is now 5 or 6, depending on the method used for labeling months. We shall see below what effect this has on the storage allocation, but the important point here is that one might need the ability to change the dimension information *while executing the program.* If the shipping-problem algorithm is sufficiently complete, it might even determine after computing for some time that warehouses are not a good idea at all, and the number of subscripts should be reduced to four *during the execution of the program.* We shall not use much of this flexibility in the simultaneous-linear-equations problem, but we should provide for it when designing a language in which to express algorithms for a great variety of problems.

There is still the important question of how we should interpret more than one subscript when we are actually using the storage of a computer. Earlier we described the storage area as if all the locations were arranged in a single line, and we used a single subscript as an indicator to tell how far along this line we were. Even though we shall use the same storage area of the computer, we shall now be using more than one subscript. In other words, whether we consider a sequence of numbers stored in the machine to be one long sequence (i.e., a vector) or whether we consider the sequence to consist of blocks of numbers (i.e., the rows of an array) placed contiguously in storage depends only on how we choose to label the individual entries. If we label an entry with a single numerical label indicating how far down the sequence it is from some initial point, then it is a *vector.* If we use a pair of numbers to label an entry (one number to designate the row, the other to designate a position in the row), we call it an *array.*

We shall thus find it necessary to know the relationship between the single subscript we would use for an element if we consider the sequence to be a vector and the multiple subscripts used for that same computer location if we think of the sequence as an array. In most computers, each storage location must be referred to by means of a single number (called its *address*), and we may regard each block

of locations in the storage area as a vector. Multiple subscripts must then be converted to a single subscript to indicate the position in that vector. To be more specific, suppose A is a vector with 200 locations set aside for it (allowing subscripts 0 to 199). Suppose, also, that we store an array of three rows and four columns inside this vector in the order A_{11}, A_{12}, A_{13}, A_{14}, A_{21}, A_{22}, . . . , A_{34}. This order implies that we store the first row, then the second row, etc. We shall say that this array is *stored by rows*, and we shall agree always to store arrays in this way. (It does not make much difference whether we agree to store by rows or by columns, as long as we are consistent. We have chosen to store arrays by rows.) Arrays with more than one subscript will always be stored with the right-most subscript varying first, then the next right-most, etc. If there are two subscripts, this amounts to storing the array by rows. If there are three subscripts for an array B, with highest subscripts p, q, and r, respectively, we would have B_{111}, B_{112}, B_{113}, . . . , B_{11r}, B_{121}, B_{122}, . . . , B_{1qr}, B_{211}, . . . , B_{pqr}.

If the array A_{11}, . . . , A_{34} is stored so that A_{11} coincides with A(0), then the subscripts line up as shown in Table 10.1.

Table 10.1

Single subscript.....	0	1	2	3	4	5	6	7	8	9	10	11
Double subscripts....	11	12	13	14	21	22	23	24	31	32	33	34

What is the rule which will determine the single subscript, given the double subscripts? If r is the single subscript and ij the double subscript, then the rule is given by the formula

$$r = 4(i - 1) + (j - 1)$$

It is easy to test this rule on Table 10.1. For example, if $i = 3$, $j = 2$, we have $r = 4(3 - 1) + (2 - 1) = 9$, so that A_{32} coincides with A(9). The derivation of the formula is quite straightforward. The *base entry* A_{11} corresponds to A(0). Given any entry A_{ij}, we must determine how far out into the vector A it lies, i.e., how many elements of the vector precede it. Since we are storing arrays by rows it will be necessary to count the number of complete rows that precede the entry A_{ij} and multiply this count by the number of

entries in each row. (There are $i - 1$ *complete* rows, since A_{ij} is in the ith row.) This will yield the number of elements of the vector contained in all the complete rows before the ith row. If we add to this the number of entries which precede A_{ij} in the ith row itself (i.e., $j - 1$), we would have the single subscript r we are after. We see now that in the formula given above for r, the term $4(i - 1)$ represents the number of entries in complete rows, with four entries per row, and $(j - 1)$ represents the number of entries in that part of the ith row which precedes A_{ij}. It is interesting to observe that the formula used here for relating single subscripts to double subscripts does not involve the total number of rows in the array at all. This is because we have stored the array by rows. If we had stored it by columns, we would have needed the total number of rows, but we would not have needed the total number of columns. We shall see a similar situation later in the handling of more than two subscripts.

In the example above of a 3×4 array A (i.e., three rows and four columns), we assumed that the base entry A_{11} was made to coincide with A(0). What would happen if someone wished to use zero subscripts, such as A_{10}, or A_{01}, or even A_{00}? If we apply to A_{00} the formula which we discussed above for finding the single subscript r, and if we assume that there are still four columns, we obtain

$$r = 4(0 - 1) + (0 - 1) = -5$$

so that A_{00} corresponds to $A(-5)$, which does not exist. The number -5 implies, however, that A_{00} should be five storage locations before A(0). If we would move the whole array over, so that the base entry A_{11} coincides with A(5), then A_{00} would coincide with A(0), and we could allow zero subscripts. We could even allow *negative* subscripts, if the base entry A_{11} were moved far enough along the vector. (Negative subscripts are very useful when they correspond to physical quantities, such as temperatures, debts, certain examination scores, etc.) We shall have to modify the formula to take into account this shifting of the base entry. If b is the single subscript of the base entry A_{11}, the new formula will be

$$r = n(i - 1) + (j - 1) + b$$

where n is the number of columns. The only effect of the shift of

the base entry is to add the amount b in each single-subscript computation. Using the example above, but putting A_{11} at A(5), we would have Table 10.2.

Table 10.2

Single subscript.....	5	6	7	8	9	10	11	12	13	14	15	16
Double subscript....	11	12	13	14	21	22	23	24	31	32	33	34

To this array the following formula would apply:

$$r = 4(i - 1) + (j - 1) + 5$$

Thus, for A_{32}, we would have

$$r = 4(3 - 1) + (2 - 1) + 5 = 14$$

showing that A_{32} now coincides with A(14).

Is there a similar formula for arrays with three subscripts? Using the same method of counting the number of entries in the array that precede the entry A_{ijk}, we may deduce the following formula for an array with m rows, n columns, and p layers (the categories for the third subscript are sometimes called "layers"):

$$r = np(i - 1) + p(j - 1) + (k - 1) + b$$

where b is again the shift amount for the base entry A_{111}. It is an interesting exercise to write down the derivation of this formula in detail. Note that the total number of rows m was again not needed in this formula. The formula for four subscripts for an $m \times n \times p \times q$ array is now easily derived to be (for A_{ijkh}):

$$r = npq(i - 1) + pq(j - 1) + q(k - 1) + (h - 1) + b$$

The method used to create such formulas is to subtract one from each of the subscripts and then multiply each one by the highest values of each of the subscripts which follow (if any), adding b as usual at the end. (If we were going to compute this value of r, a more efficient way to write the formula would be

$$r = q[p\{n(i - 1) + (j - 1)\} + (k - 1)] + (h - 1) + b$$

since there are fewer operations to be performed.)

Let us return now to the question of building all this flexibility into the dimensioning of arrays in our language. We saw earlier that the greatest flexibility we could allow would be to let the program modify the dimension information *during execution.* If this is to be possible, however, this information must be stored in the computer in such a way that one can write a statement in our language which can refer to it. We are thus led to the very powerful procedure of storing this dimension information for an array A (such as the number of subscripts, the value of b, etc.) as ordinary integer values of some other vector, say B, which we shall call the *dimension vector for* A. We shall indicate the association between the array A and its dimension vector B by means of the dimension statement as follows:

DIMENSION A(500,B(0))

which will mean that the highest *single subscript* to be expected for A will not exceed 500 (just as for ordinary vectors) and the dimension information for A will be found starting in B(0), i.e., in B(0), B(1), etc. The 500 is needed here, as before, to determine the total amount of storage to be allocated to A. The dimension information stored in the vector B as ordinary integer values will indicate how the block of storage for A will be interpreted when subscripts are used. Let us specify the form in which this information is to be stored. The numbers to be stored here are (1) the number of subscripts, (2) the value of b, and (3) the total number of columns, layers, etc., but not the total number of rows. The total number of columns will coincide with the highest subscript for columns, if that subscript starts with 1. If it ranges from -10 to $+20$, however, then the total number of columns is 31. In general, if a subscript varies from a to b, then the number we need for the dimension information is $b - a + 1$. Since most arrays are numbered starting with 1, however, this number will usually coincide with the highest subscript. Let the dimension information be stored, then, as follows:

B(0) = number of subscripts
B(1) = b
B(2) = total number of columns
B(3) = total number of layers
etc.

The number of entries here will depend on the number of subscripts. For example, an array with three rows and four columns might have the following dimension information:

$$B(0) = 2$$
$$B(1) = 1$$
$$B(2) = 4$$

A vector is really a special case of what we are calling an array, in that (1) we have just one subscript, (2) the base entry A_1 is always at A(1), and (3) there is just one column per row. Since this information is always fixed for a vector, we do not need to store separate dimension information, and this is indicated in the dimension statement by not putting the name of the dimension vector in the parentheses along with the highest subscript. Thus, we wrote

```
DIMENSION Q(50)
```

in the change problem. Should we wish to vary the base-entry position so as to use negative subscripts on vector elements, we may use a dimension vector and indicate that there is only one subscript. The value of b need not be fixed at 1 if we do this, and by making b large enough we would be able to use negative subscripts. Thus, a vector for which negative subscripts will be used might have the following dimension statement and dimension vector:

```
DIMENSION VEC(100,V(0))
V(0) = 1
V(1) = 50
```

This would allow the use of negative subscripts down to -50.

It is important to remember that the dimension vector is itself a vector since we use subscripts for it, and its name must therefore appear in a dimension statement also, showing the highest subscript to be used for it. For the case of the vector VEC just above, we would probably write

```
DIMENSION VEC(100,V(0))
DIMENSION V(1)
```

since the highest subscript expected on V is 1. To make such statements more convenient, let us allow the combination of any number

of dimension declarations (in any order) into a single statement. Thus, we may write

```
DIMENSION V(1), VEC(100,V(0))
```

for this case and

```
DIMENSION A(500,B(0)), B(2)
```

for the example mentioned earlier of an array A with three rows and four columns. There is no need to declare dimension vectors to be of integer mode, since the kind of values they will have (i.e., integers) is clear as soon as they appear as dimension vectors in the dimension statements of other arrays. We may make one other convention to simplify some of our writing. Since the name of a vector or array has at present no meaning if it is written *without* a subscript, let us give it the meaning of having an implied subscript zero after it. Thus, the dimension statement

```
DIMENSION V(1), VEC(100,V(0))
```

could be written

```
DIMENSION V(1), VEC(100,V)
```

The question of how to write an array name with more than one subscript raises another point that should be settled. Since we already have a notation for vectors [i.e., VEC(1), VEC(2), etc.], let us extend this to include arrays by writing the subscripts between parentheses, separated by commas. Thus, A_{34} would be written A(3,4). The number of subscripts should agree with the number indicated in the dimension vector, although we could always write the appropriate single subscript (as computed by our formula for r) if we found it convenient. The subscripts themselves may be any integer expressions, as usual.

10.5 THE VECTOR-VALUES STATEMENT

We have now specified the form of the more general dimension statement and the form of the information which is to go into the dimension vector, i.e., the number of subscripts, the base-entry shift amount

b, and the total number of columns, layers, etc. We have also described the method of writing multiple subscripts when referring to one of these array elements, such as A(3,4). Looking ahead to writing a program which uses these new additions to the language, however, there is still one question to be answered: How does the dimension information get into the dimension vector? One way would be to write a sequence of arithmetic substitution statements, such as

```
B(0) = 2
B(1) = 1
B(2) = 4
```

but it is inconvenient to write this much for each array. Even more important, these instructions take up storage space and take time to execute. Although this method would work, it is not very desirable. Another method would be to read in as data all the values needed for the dimension vector. Thus, early in the execution of each program would be a statement

```
READ DATA
```

and along with all the other data for the problem, all the dimension information could be brought in. Although this method allows each set of data to bring in its own special dimension information, it can be a nuisance to have to prepare as data any information which will not change at all. For example, in the simultaneous-equations problem the total number of columns may change with each set of data, but not the number of subscripts or the base-entry subscript *b*.

What we need is a way to say, "Here are some values which should be set in advance for certain vectors. No time should be spent computing these values; they should simply be entered into the computer at the time the program itself is brought in." The writer of the program should be able to write his statements as if the information is already there. We must provide a new *declarative* statement which has the job of describing some *preset* vector values. A convenient form for this statement is

```
VECTOR VALUES B = 2, 1, 4
```

The interpretation of this statement is that B [which is really B(0), according to the convention established in Section 10.4] will be preset to the integer value 2 (meaning two subscripts), B(1) will be preset to the value 1 (the value of *b*), and B(2) will be preset to 4 (the number of columns).

Such a statement will be useful for presetting vectors for many other purposes, also. This is a convenient way to build into the program certain constants which may be needed. We might have chosen to use this type of statement to preset the values of the coins in the change problem, for example. In the encoding-decoding problem of Chapter 5, the standard alphabet could be preset in this way by writing

```
VECTOR VALUES STAND = $A$, $B$, $C$
```

and so on. At the time we discussed this alphabet, we had no way to bring these characters into the computer at all.

Since all entries of any vector are of the same mode, i.e., integer, noninteger, Boolean, etc., we shall have to stipulate that all values to be preset into a given vector have the same mode. Moreover, for some problems it might be useful to preset values in positions of a vector which are not at the beginning, such as R(6), R(7), etc. The name of the vector should be allowed to have a constant single subscript, then, so that values may be preset into any set of contiguous locations within the vector. We might write

```
VECTOR VALUES R(6) = 1.2, .3, -4.1
```

which would preset R(6), R(7), and R(8) to the noninteger values 1.2, .3, and −4.1, respectively.

Certain additional information is available in the vector-values statement. For example, the mode of the values of the vector is established by the values to which it is being preset. Let us agree, then, that no mode declaration is necessary for any vector which appears in a vector-values statement. A program should not be considered incorrect, however, if it contains more than one (implicit or explicit) mode declaration for a given vector, provided they all agree as to which mode it is. Some indication is also available as

to the total amount of storage needed for a vector which appears in a vector-values statement. It needs at least enough storage to accept the values being declared in that statement. Thus, the statement just above which presets R(6), R(7), and R(8) implies that R is of noninteger mode and needs to accommodate a subscript at least as high as 8. We should accept this as *implicitly* declaring the highest subscript of R to be 8, i.e., as implying the statement

```
DIMENSION R(8)
```

Later, however, we may find the following statement in the program (these statements, being declarative, may occur anywhere in the program):

```
VECTOR VALUES R(17) = 2.1
```

which implies the effect of the statement

```
DIMENSION R(17)
```

Later in the program, we may actually find an *explicit* declaration

```
DIMENSION R(50)
```

Rather than consider these to be conflicting declarations, which would not be the case, we shall simply use the largest value we find for that vector, whether implicitly or explicitly declared.

Of the methods described above for putting the dimension information into the dimension vector, the most commonly used are (1) reading it in as data by means of the READ DATA statement and (2) presetting it via the vector-values statement. As we indicated above in the shipping problem, however, it is very useful in some programs to be able to compute new values for some of the information. In writing the program for the simultaneous linear equations we shall deliberately use a combination of all three methods to illustrate their use. The number of rows (n) of the matrix of coefficients will be read in as data, the number of columns ($n + 1$) will be computed from this input value n and stored in the dimension vector, and the number of subscripts and the value of b will be set by a vector-values statement.

10.6 THE PROGRAM FOR THE JORDAN ALGORITHM

We are now in a position to write the program corresponding to the flow diagram of Figure 10.11. The program appears as Figure 10.12. For easier reference, the flow diagram of Figure 10.11 is reproduced here.

Note that the box in Figure 10.11 which calls for printing out the

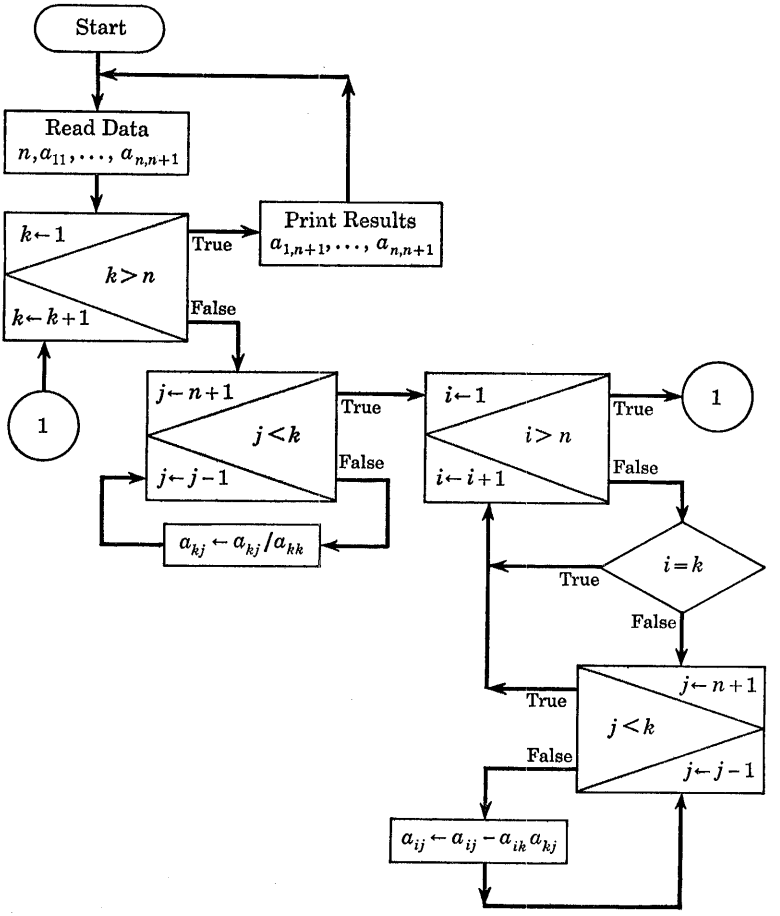


Figure 10.11

```
        R THE READ DATA STATEMENT WHICH FOLLOWS
        R BRINGS IN THE NUMBER OF EQUATIONS N
        R AND THE MATRIX OF COEFFICIENTS A(1,1), . . . ,
        R A(N,N + 1)
INPUT     READ DATA
        R THE NEXT THREE STATEMENTS SET UP THE
        R DIMENSION INFORMATION FOR THE MATRIX
        R OF COEFFICIENTS
          DIMENSION A(400,B), B(2)
          VECTOR VALUES B = 2, 1
          B(2) = N + 1
        R THE NEXT STATEMENT BEGINS THE JORDAN ALGORITHM
          THROUGH LOOP1, FOR K = 1, 1, K .G. N
          THROUGH LOOP2, FOR J = N + 1, -1, J .L. K
LOOP2     A(K,J) = A(K,J)/A(K,K)
          THROUGH LOOP1, FOR I = 1, 1, I .G. N
          WHENEVER I .NE. K
              THROUGH LOOP3, FOR J = N + 1, -1, J .L. K
LOOP3         A(I,J) = A(I,J) - A(I,K) * A(K,J)
LOOP1     END OF CONDITIONAL
          THROUGH LOOP4, FOR I = 1, 1, I .G. N
LOOP4     PRINT RESULTS A(I,N + 1)
          TRANSFER TO INPUT
          INTEGER N, K, I, J
          END OF PROGRAM
```

Figure 10.12

results has been expanded into an iteration statement

```
THROUGH LOOP4, FOR I = 1, 1, I .G. N
```

and a standard output statement

```
LOOP4     PRINT RESULTS A(I,N + 1)
```

One would normally expect to use the notation A(1,N + 1), . . . , A(N,N + 1), but this notation always implies that *all* storage locations between the first and last locations as indicated be printed out. The entries A(1,N + 1), A(2,N + 1), . . . , A(N,N + 1) are not consecutively stored, since each occurs in a different row.

The question may arise as to why the number of columns $(n + 1)$

was computed by the statement

$$\text{B(2)} = \text{N} + 1$$

instead of being included in the vector-values statement, perhaps in the following way:

$$\text{VECTOR VALUES B} = 2,\ 1,\ \text{N} + 1$$

Unfortunately, all numbers which appear in vector-values statements must be *constants*, since they must be preset. Since $n + 1$ depends on the value of n which comes in as data, it cannot be preset, and therefore it must be computed after n comes into the computer along with the other data.

10.7 THE DIVISION BY ZERO PROBLEM

We have now developed the Jordan algorithm through the flow diagram and the program. There has been one important omission, however, and that is the case in which the pivot entry on the main diagonal is zero, causing a division by zero during the loop which was shown in Figure 10.8. We can always put in a conditional statement to test whether or not the pivot entry is zero, but we must decide what to do if it does turn out to be zero. (If it is not zero, we would go ahead as in the present program.)

An example of a set of simultaneous linear equations in which this occurs is the following:

$$\begin{aligned} x + y + z &= 3 \\ x + y + 2z &= 4 \\ 2x - y - z &= 0 \end{aligned}$$

for which there is a solution, *viz.*, $x = y = z = 1$. If we apply the Jordan algorithm which we have developed to the matrix of coefficients

$$\begin{bmatrix} 1 & 1 & 1 & 3 \\ 1 & 1 & 2 & 4 \\ 2 & -1 & -1 & 0 \end{bmatrix}$$

we obtain the following array after the first row has been used as the operating row to modify the other two rows:

$$\begin{bmatrix} 1 & 1 & 1 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & -3 & -3 & -6 \end{bmatrix}$$

Now we see that the entry a_{22}, which would have been the next pivot entry, is zero, and the algorithm we have developed would be in difficulty. One of the points we made earlier, however, is that interchanging rows (which amounts to interchanging equations) does not alter the set of solution values. Using this, we may interchange the second and third rows to obtain the array

$$\begin{bmatrix} 1 & 1 & 1 & 3 \\ 0 & -3 & -3 & -6 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

which no longer has the zero-divisor trouble. Using the new second row as the operating row leaves the array

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

and using the third row as the operating row now produces the array

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

from which the solution $x = y = z = 1$ is easily obtained.

One way out of the zero-divisor dilemma, then, is to find some other row with a nonzero entry in the column containing the zero pivot entry and interchange the two rows. The operating row now does not have a zero on the main diagonal, and the problem is avoided. It does not do any good to find a nonzero entry in a row that has already been an operating row (such as the first row in the example above), because it will have a 1 as its former pivot entry on the main diagonal, and that 1 would destroy zeros in other rows if that row were to be used as an operating row again. Thus, we need

to look for nonzero entries *below* the row in which the zero pivot entry occurs.

What will happen if all the rows below the one with the zero pivot entry also have zeros in that column? This is exactly the case we discussed in Section 10.1 in connection with Figures 10.3 and 10.4, e.g., the situation in which there was no solution (because the lines or planes did not meet at all) or the case in which there were too many solutions (because the planes intersected in an entire line). As examples of these possibilities we shall consider the two sets of equations

$$\begin{aligned} x + y + z &= 3 \\ x + y + z &= 4 \\ 2x - y - z &= 0 \end{aligned}$$

and

$$\begin{aligned} x + y + z &= 3 \\ 3x + 3y + 3z &= 9 \\ 2x - y - z &= 0 \end{aligned}$$

The first set will have no solutions, since the first two equations can never both be true at the same time. (The corresponding planes are parallel.) The second set has too many solutions, since the first two equations represent the same plane, and the first plane therefore contains the whole line of intersection of the second and third planes. Note that if we divide the second equation of the second set by 3, these two sets will differ only on the right side of the equations, and the trouble we will encounter will be discovered while computing with the coefficients on the left side only. It follows that whenever we encounter this zero-divisor difficulty and cannot find any nonzero pivot entry at all, we shall conclude that there is no unique solution, but we cannot say whether it is because there are no solutions at all or because there are too many.[1] We shall have to modify the pro-

[1] For those who know about determinants, this inability to find a nonzero pivot entry means that the determinant of the square matrix obtained by using just the left side of the equations is zero. This can be shown to imply that there is no unique solution to the equations. We could resolve the question as to which is the correct reason, since there is no solution at all if the right side of the equations contains any nonzero values at all and there are many solutions if the right side contains only zeros. We shall be content with a comment as to what has happened, however.

gram exhibited in Figure 10.12 to comment "NO UNIQUE SOLUTION" in case this situation arises.

Now let us see how to interchange two rows. Interchanging rows requires saving one row in temporary storage while moving the other row. The row that was saved must then be moved to its new position. This is very similar to the interchange of two numbers in the sorting algorithm which we discussed in an earlier chapter. The statements in the sorting program which accomplished the interchange of A(I) and A(J) were

```
X = A(I)
A(I) = A(J)
A(J) = X
```

Since such moves of entire rows of numbers can be time-consuming, is there a way to accomplish the same thing without actually moving so many numbers? The general problem of moving too many things around in a computer must be considered in other situations as well, for example, if one is sorting blocks of information according to some *key*, or *label*, attached to each block. We saw earlier that many interchanges may be necessary in a sorting computation, and if each interchange moves many numbers around, the computation time becomes prohibitive. A commonly used device is to leave the blocks where they are and store with the key the location of the block. Then only the keys are sorted, each one taking with it now not the block, but its location. When the sorting is done, the blocks have effectively been sorted, since their keys are in the correct order, and each one leads immediately to its corresponding block via the block location. In the process only pairs of numbers were moved, rather than whole blocks.

A similar device will be useful here, where we wish to avoid moving entire rows. (The row here corresponds to the block of information referred to in the preceding paragraph.) Let us attach a *label* to each row, e.g., its row number as a row of the original matrix of coefficients. Then, in handling the problem of division by zero, we shall interchange the labels (i.e., the row numbers) and leave the rows themselves where they are. The effect that this will have on the algorithm is that whenever an entry was identified as being in

a row, say the third row, it will now be identified as being in the row whose *number* is currently the third number. As an example, let us use the set of equations with the solution $x = y = z = 1$ which we considered above:

$$\begin{aligned} x + y + z &= 3 \\ x + y + 2z &= 4 \\ 2x - y - z &= 0 \end{aligned}$$

We now must maintain a list of the row labels, with the normal ordering as the initial setting:

Position	Label
1	1
2	2
3	3

We saw that the matrix of coefficients is transformed into the array

$$\begin{bmatrix} 1 & 1 & 1 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & -3 & -3 & -6 \end{bmatrix}$$

after the first row is used as the operating row. Now we interchange, not the second and third rows, but the second and third labels, giving the following list of row labels:

Position	Label
1	1
2	3
3	2

Since we will now refer to the pivot entry as *the entry in the row whose number is second* (rather than the entry in the second row), we shall in effect be referring to the entry in the third row. Since we are not interchanging columns here at all, column references will remain unchanged.

We now need a simple way to refer to the row whose number is in the third position of a table. This is very similar to the situation

in Chapter 9, where we needed to refer to a node whose number was to be found as the R node in another table, e.g., the network description. Let us suppose that the row labels are stored as the values of a vector called T (for *table*). We have at the beginning of the computation, then, T(1) = 1, T(2) = 2, and so on. [It is more convenient to avoid using T(0) for anything at all this time.] We might expect to preset these values by means of the statement

```
VECTOR VALUES T(1) = 1, 2, 3
```

except that our algorithm is supposed to work for n equations, and we would not know how many values to preset. We shall therefore use a loop, determined by n:

```
      THROUGH LOOP, FOR I = 1, 1, I .G. N
LOOP  T(I) = I
```

Throughout the program there will be references to entries in the array, say A(I,J). Now, however, we will not want to refer to an entry in the Ith row. If I is 3, we will need the third number in the vector T; i.e., we need the T(3)th row. More generally, instead of I, we need T(I) as the first subscript. Instead of writing A(I,J), therefore, we shall write A(T(I),J). At the beginning, T(3) = 3, so the effect will be the same, but once an interchange has been made, the effect of going through the table will be noticed.

In Figure 10.13 we have the revised program. The following changes have been incorporated: (1) All row references are now made via the vector T, as described above. (2) Before dividing by the pivot entry, a test is made to see if it is zero. If it is, a search is made of the rest of that column to see if a nonzero entry can be found. If not, the comment "NO UNIQUE SOLUTION" is printed. If it can be found, the appropriate row labels (i.e., entries in the vector T) are interchanged. (3) The vector T is initialized by a small loop, as indicated earlier. Note that T is described as having highest subscript 19. This follows from the allocation of 400 locations to the $n \times n + 1$ array A. If $n(n + 1) = 400$, then $n \leq 19$. This program should be compared with that shown in Figure 10.12.

```
        R THE READ DATA STATEMENT WHICH FOLLOWS
        R BRINGS IN THE NUMBER OF EQUATIONS N
        R AND THE MATRIX OF COEFFICIENTS A(1,1), . . . ,
        R A(N,N + 1)
INPUT     READ DATA
        R THE NEXT THREE STATEMENTS SET UP THE
        R DIMENSION INFORMATION FOR THE MATRIX
        R OF COEFFICIENTS AND ROW NUMBER TABLE
          DIMENSION A(400,B), B(2), T(19)
          VECTOR VALUES B = 2, 1
          B(2) = N + 1
        R THE NEXT STATEMENT BEGINS THE JORDAN ALGORITHM
          THROUGH Z, FOR I = 1, 1, I .G. N
Z         T(I) = I
          THROUGH LOOP1, FOR K = 1, 1, K .G. N
          WHENEVER A(T(K),K) .NE. 0
DIVIDE        THROUGH LOOP2, FOR J = N + 1, -1, J .L. K
LOOP2         A(T(K),J) = A(T(K),J)/A(T(K),K)
          OTHERWISE
        R FIND NONZERO PIVOT ENTRY, IF ANY
              THROUGH Z1, FOR I = K + 1, 1, I .G. N
Z1            WHENEVER A(T(I),K) .NE. 0, TRANSFER TO FOUND
              PRINT RESULTS $NO UNIQUE SOLUTION$
              TRANSFER TO INPUT
        R TRANSFER TO FOUND INDICATES THAT ROW NUMBER
        R INTERCHANGE CAN BE MADE, FOLLOWED BY
        R NORMAL ROW DIVISION
FOUND         X = T(I)
              T(I) = T(K)
              T(K) = X
              TRANSFER TO DIVIDE
          END OF CONDITIONAL
          THROUGH LOOP1, FOR I = 1, 1, I .G. N
          WHENEVER I .NE. K
              THROUGH LOOP3, FOR J = N + 1, -1, J .L. K
LOOP3         A(T(I),J) = A(T(I),J) - A(T(I),K) * A(T(K),J)
LOOP1     END OF CONDITIONAL
          THROUGH LOOP4, FOR I = 1, 1, I .G. N
LOOP4     PRINT RESULTS A(T(I),N + 1)
          TRANSFER TO INPUT
          INTEGER N, K, I, J, T, X
          END OF PROGRAM
```

Figure 10.13

PROBLEMS

1. We saw in Section 10.4 that we could find the single (vector) subscript r for an array entry if we are given the double subscripts i and j, the single subscript b for the base entry A_{11}, and the number of columns n, according to the rule

$$r = n(i - 1) + (j - 1) + b$$

What is the inverse rule, i.e., the rule for finding i and j when r is given, assuming that n and b are known? Can you derive a more general inverse rule to handle more than two subscripts? (*Hint:* Use the MODULO. function defined in Figure 6.12.)

2. Derive the formula given in Section 10.4 for the single subscript r in terms of i, j, and k:

$$r = np(i - 1) + p(j - 1) + (k - 1) + b$$

where the array has m rows, n columns, and p layers.

3. The following two statements occur in a program:

```
DIMENSION A(150,B)
VECTOR VALUES B = 2, 10, 15
```

a. How many subscripts can be used with A?

b. How many columns does A have?

c. What is the maximum possible value of the row subscript of A in the program?

d. What value must I have if A(I) and A(3,4) are in the same location?

e. How many locations are reserved for A?

f. What is the mode of the vector B? Why?

4. There is another algorithm for solving sets of simultaneous linear equations called the Gauss algorithm. This is similar to the Jordan algorithm, except that instead of clearing to zeros every entry in a column except the pivot entry, we clear to zero *only those entries which lie below the main diagonal.* For the example used earlier

$$\begin{aligned} x + y + z &= 1 \\ x + 4y - z &= 2 \\ 4x - y + 2z &= 5 \end{aligned}$$

with the solution $x = \frac{3}{2}$, $y = 0$, and $z = -\frac{1}{2}$, the matrix of coefficients

is transformed into the following array in this case:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & -\frac{2}{3} & \frac{1}{3} \\ 0 & 0 & 1 & -\frac{1}{2} \end{bmatrix}$$

From the last equation we obtain $z = -\frac{1}{2}$, and then a back solution is needed, just as in the method of addition and subtraction discussed in Section 10.2. Modify the flow diagram (Figure 10.11) and program (Figure 10.12) for the Jordan algorithm so that they will represent the Gauss algorithm instead. In this problem you should ignore the possibility of division by zero.

5. Numerical analysts have shown that the roundoff error is reduced considerably if the largest entry in the operating row (in absolute value) is used as the pivot entry. This is due to the fact that all numbers in the operating row are then less than or equal to 1 (in absolute value) after the division by the pivot entry. (Dividing a number by a larger number always produces a number less than 1.) Many programs for the Jordan algorithm use this as a guide and search out the largest value. They then interchange columns (or column numbers, as we did row numbers) to make this largest value the pivot entry. If all entries in a row are zero, there is no unique solution, since any triple of numbers (x_0,y_0,z_0) is a solution to the equation

$$0 \cdot x + 0 \cdot y + 0 \cdot z = 0$$

If this method is used, there is no need for row interchanges at all, since the zero divisor difficulty is handled by column interchanges. Modify Figures 10.11 and 10.12 to use this method.

CHAPTER ELEVEN

THE MAD LANGUAGE

IN THE PRECEDING chapters we have studied a variety of problems, and in the process we have described a language which has proved quite useful for communicating algorithms to computers. One may very well ask, however, how closely this language corresponds to the languages actually being used on computers. The purpose of this chapter and the next is to show how our language is related to some of the languages in actual use and how these are, in turn, related to the computer itself. The discussion of these topics must necessarily be condensed, and we cannot, for example, completely describe the other languages. We shall try, however, to show how our language is related to each of them.

It should be quite apparent that many of the decisions we made in developing this language could have been made differently. For example, the word WHENEVER which is used in the conditional statements might have been chosen to be IF instead. Why did we choose the longer word? One reason is that IF might have been confused with the name of a variable, whereas WHENEVER could never be the name of a variable (since it contains more than six characters). For example, given the conditional statement

```
WHENEVER A(I + 1,J + K) .L. 3
```

let us rewrite it using IF:

$$\text{IF } A(I + 1,J + K) .L. 3$$

Remembering that we have decided to ignore blanks completely, it takes a rather involved analysis to decide that the three letters IFA at the beginning do not form the name of a variable but are in fact two separate objects, a word IF indicating the type of sentence and a variable name A. This difficulty in interpretation is avoided by choosing the words which indicate the type of sentence in such a way that they cannot be mistaken for names of variables. We chose such words as THROUGH, WHENEVER, TRANSFER TO, DIMENSION, BOOLEAN, EXECUTE, INTERNAL FUNCTION, READ DATA, and so on, because they all contain more than six letters, and they cannot be confused with anything else.

Such decisions have led us to our particular language. Except for some minor differences, which we shall point out below, this language is entirely contained in a language called MAD, developed at the Computing Center of the University of Michigan, and in use since February, 1960. Almost all the important features of the MAD language have been described here and included in our language. We shall briefly describe some of the additional features which we have not yet seen. Before doing that, however, let us point out those features of our language which do not exist in MAD.

1. The integer division in our language agrees in every respect with the greatest-integer function. In other words, $7/3 = 2$ and $-7/3 = -3$. In MAD, the rule for integer division is that any fractional part that the quotient might have is simply dropped. For positive numbers the two definitions agree, but for negative numbers with fractional parts, they do not agree. Thus, for programs written in the MAD language, $7/3 = 2$, but $-7/3 = -2$ instead of -3. This difference would not have had any effect on the change problem in Chapter 1, since we were dealing there with positive numbers, but the program for decoding messages in Chapter 5 would have to be modified slightly if written in MAD.

2. One cannot use a dimension vector for a *vector* in MAD. In other words, a true vector using only one subscript cannot have its base entry shifted to allow negative subscripts. We saw an example

of this in our language in Section 10.5 where a vector VEC was given a dimension vector V by means of the statement

DIMENSION V(1), VEC(100,V)

This is a relatively minor point, and it does not affect any of the problems we have considered here.

3. In the PRINT RESULTS statement in MAD, one cannot include comments such as $NO UNIQUE SOLUTION$. Another statement, PRINT COMMENT, is available for this in MAD.

All the other differences between our language and MAD can be described as additional features available in MAD. Some, but not all, of these additional features are:

1. Nonintegers (i.e., floating-point numbers) may include an extra integer which indicates the power of 10 to be used as a scale factor. This integer is appended with the letter E. Thus, 3.2E−1 = 3.2×10^{-1} = .32, .012E3 = 12., 1.2E−10 = .00000000012, and 4E5 = 400000. The presence of the E indicates that the number is noninteger, even if the period is omitted.

2. Along with .OR. and .AND., several other Boolean connectives are available. Three connectives .THEN., .EQV., and .EXOR. (representing implication, logical equivalence, and the exclusive *or*, respectively) are defined by the truth table below (Table 11.1). Remember that we are now using 1B and 0B for *true* and *false*, respectively.

Table 11.1

$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$ .THEN. $\mathcal{R}$	$\mathcal{P}$ .EQV. $\mathcal{R}$	$\mathcal{P}$ .EXOR. $\mathcal{R}$
0B	0B	1B	1B	0B
0B	1B	1B	0B	1B
1B	0B	0B	0B	1B
1B	1B	1B	1B	0B

The Boolean unary operation .NOT. is also available in MAD. It is defined by Table 11.2.

3. There are a few special statements in MAD to enable one to

write internal or external functions which call on themselves. In other words, it may happen occasionally that within a definition program for a function, there is a call for that function itself. This is not allowed in our language.

Table 11.2

℘	.NOT. ℘
0B	1B
1B	0B

4. MAD has very general input and output statements, as well as the READ DATA and PRINT RESULTS statements, which allow the writer of the program great flexibility in describing the format of the input on the punched card or other input medium and the format of the line to be printed or punched as output. For the purpose of communicating algorithms to the computer, this flexibility is of minor importance, as we have seen.

5. MAD has facilities for allowing the writer of the program to organize his variable storage to some extent. Thus, to conserve storage, he may declare two arrays A and B to be "equivalent" by means of the statement

EQUIVALENCE (A,B)

which means that they will share the same storage. The writer may also declare certain variables, vectors, or arrays to be "erasable," meaning that they are used only as very temporary storage, such as X is used above in the interchange of two variables. The storage space allocated to them may be "erased" in the sense that it may be shared by external functions that this program may call upon. When these external functions use that storage space, they will erase any information previously stored there.

These additional facilities in MAD do not add a great deal to the language. The language we have discussed in the first 10 chapters is essentially the same as MAD. From now on we shall refer only to MAD, and we shall not need to distinguish between the real MAD language and our language.

CHAPTER TWELVE

OTHER COMPUTER LANGUAGES

12.1 THE LANGUAGE AND THE COMPUTER

IN THIS CHAPTER we shall be concerned with some of the other languages which are available for computers. Before we can appreciate even the existence of several languages, however, we should consider the relationship between a language, such as MAD, and the computer.

Throughout this entire study of computer problems and algorithms, very little mention has been made as to how the statements of the language would be rendered intelligible to the computer. This question was avoided because we were interested in a language which would be good for expressing algorithms, and we were only indirectly concerned with the details of actually executing the program on a machine. We did hint in Chapter 11 at one of the devices used to make our statements more intelligible when we pointed out why some words were deliberately chosen to contain more than six letters. It is very reasonable to ask, however, how a computer does make sense out of programs such as we have been writing, since computers normally do not accept as their instructions such statements as

THROUGH A, FOR I = 1, 1, I .G. N

The physical computer is designed to modify the contents of certain registers (or storage locations) when its control unit is given an instruction which causes it to do so. The machine instruction is usually a number, in which certain digits are interpreted as the operation to be performed, other digits are interpreted as the address in storage of the operand (or operands), and other digits might indicate where the next instruction is to be found. Thus, in the IBM 650 computer, for example, a typical instruction might be

1512150123

in which the first two digits (15) indicate that the contents of the location specified by the next four digits (1215) are to be added to the lower half of the accumulator register. The next four digits (0123) give the address of the location in storage in which the next instruction to be executed is stored. Most of these hardware instructions actually do very little computation. One instruction may cause a number to be subtracted from another. Another instruction may cause a number to be moved from one location to another location in storage. Still another may instruct the control unit to examine the contents of some storage location. If the number there is zero, the next instruction should be skipped; otherwise it should be executed. The point is that it normally takes long sequences of such relatively small instructions to do the computation described in such statements as

```
V = SQRT.(-2. * ELOG.(.5 * (1. - .ABS. (1. - 2. * R))))
```

which we saw in Figure 6.12. As an example, on the IBM 7090 computer this statement might be represented by the hardware instructions shown in Figure 12.1. [The language used is FAP (FORTRAN Assembly Program). It is essentially the same as the hardware language.] On other computers, the sequence might be much longer.

Where do these sequences of hardware instructions come from when the computer is presented with one of our MAD programs? The computer cannot actually execute anything but hardware instructions; so somehow each MAD program must be *translated* into

```
        LDQ     TWO
        FMP     R
        FAD     ONE
        SSM
        FAD     ONE
        XCA
        FMP     HALF
        TSX     ELOG, 4
        XCA
        FMP     TWO
        TSX     SQRT, 4
        STO     V
ONE     DEC     1.
TWO     DEC     -2.
HALF    DEC     .5
```

Figure 12.1

the hardware language.[1] This translation process does not solve the problem for which the program was written. It takes one representation of the algorithm (i.e., a MAD program) and transforms it into another representation of the same algorithm (i.e., a sequence of hardware instructions). Each of the two representations serves a definite purpose. The MAD representation is relatively easy for a person to write and understand at a later time. It is also quite easy to find errors, make corrections, and make changes in the algorithm itself. The hardware representation, however, has the one great advantage that it is directly intelligible to the computer.

We need a translator, then, which can look at a MAD program and generate the corresponding hardware instructions. In the early days of computers, before such languages as MAD were available, computer users wrote the hardware instructions directly, and they were (perhaps unconsciously) doing the translation themselves. They had an idea of the form of the algorithm, and they often had a flow diagram, but they translated the algorithm into hardware instructions manually. As these computer users became more sophisticated in their use of the machine and more aware of its capabilities, lan-

[1] Some machines are now beginning to appear which can do a limited amount of computation directly from an arithmetic expression. Statements which control the order in which other statements are executed, such as iteration statements, must still be translated into the hardware language of each machine.

guages such as MAD and the others to be discussed below were developed, and with each one a *translator program* was provided. The translator program is a set of instructions, usually written in the hardware language (or a language very close to the hardware language). This program does the translation *on the computer* from the MAD language to the hardware language. The translation is treated as just another problem to be solved on the computer; the data consist of our MAD statements, and the results of the computation are hardware instructions. The translator itself is usually called a *compiler* (since it compiles sets of hardware instructions), and the language which it translates (in our case, MAD) is often called an *algebraic language.*

The algorithm for carrying out the translation was developed in much the same way in which we worked out the algorithms for other problems in this book, except that the translation problem is much more complicated than the problems we have discussed. The translation algorithm was then (manually) translated into the hardware language of the computer. When the errors had been discovered and removed, the designers of the language had a translator with which they could translate programs written in the MAD language into the hardware language of their computer.

In the sections which follow, we shall discuss FORTRAN and ALGOL, two other algebraic languages. We shall not attempt to describe these languages in any detail at all. We are interested only in making some comparisons between these languages and MAD.

12.2 THE FORTRAN LANGUAGE

One of the first algebraic languages to be developed was FORTRAN (Formula Translator), produced by the IBM Corporation. The design of the language and the writing of the translator for the IBM 704 computer were major steps forward in the use of computers at the time (1956), since almost everyone was using the hardware language of the computers that existed then. Actually, most people were using languages which needed to be translated to the hardware language, but which were not very far from the hardware language, anyway. These languages, usually called *assembly languages* (because the translator assembled the hardware instructions for them), did

little beyond allowing one to write

```
ADD
```

instead of the numerical code. (The example of a sequence of hardware instructions for the IBM 7090 computer in Section 12.1 was written in FAP, an assembly language.) On the other hand, FORTRAN introduced the use of arithmetic expressions such as we have used in MAD, and the translator for the FORTRAN language was required to produce many hardware instructions for each statement of the program. FORTRAN is now one of the most widely used languages in the computer field. Translators have been written for the FORTRAN language for several computers, and it is safe to say that FORTRAN will be in use for many years.

Some of the statements of the FORTRAN language are quite similar to those of the MAD language, but many of the features which we have found very useful here are not available in FORTRAN. For example, the language does not include alphabetic constants, Boolean expressions, compound-conditional statements, or the vector-values statement. Corresponding to the very general conditional statements in MAD, which allow one to construct complicated Boolean expressions and even combine several alternatives by means of the compound-conditional statement, FORTRAN provides only the IF statement, which has the form

```
IF (I - 3) 5, 6, 7
```

This is interpreted as follows: If the expression in parentheses (i.e., I − 3) is less than zero, transfer to the statement labeled 5; if the expression is equal to zero, transfer to the statement labeled 6; and if the expression is greater than zero, transfer to statement 7. (In FORTRAN all statement labels are integers.)

```
WHENEVER I .G. J .AND. G .E. I + 2
    A(I) = B(J) + 3.
OR WHENEVER G .E. 1 .OR. I .GE. J - 2 .AND. G .E. 0
    A(I) = B(J)
OTHERWISE
    A(I) = B(J) - 3.
END OF CONDITIONAL
C = A(I) * A(I)
INTEGER I, J, G
```

Figure 12.2

Figure 12.3 shows the FORTRAN program which is needed to make the decision shown in MAD in Figure 12.2. We have replaced the variable name G by the name LG, since in FORTRAN one cannot declare the mode of a variable. Any variable name whose first letter is I, J, K, L, M, or N is automatically of integer mode, and all other names are automatically noninteger. The GO TO statement corresponds to the TRANSFER TO statement in MAD.

```
      IF (I - J) 3, 3, 1
1     IF (LG - I - 2) 3, 2, 3
2     A(I) = B(J) + 3.
      GO TO 8
3     IF (I - J + 2) 5, 4, 4
4     IF (LG) 5, 6, 5
5     IF (LG - 1) 7, 6, 7
6     A(I) = B(J)
      GO TO 8
7     A(I) = B(J) - 3.
8     C = A(I) * A(I)
```

Figure 12.3

The iteration statement in FORTRAN (usually called the DO statement) has a very restricted form:

$$\text{DO } n \text{ K} = M_1, M_2, M_3$$

where n is a statement label, K is the iteration variable (integer mode only), and M_1, M_2, M_3 may be either unsigned integer constants or nonsubscripted integer variables. The interpretation is that the scope consists of the statements which follow, up to and including the statement labeled n. The scope is to be executed for $\text{K} = M_1$, then $\text{K} = M_1 + M_3$, and so on, incrementing each time by M_3, until $\text{K} > M_2$. (If M_3 is omitted, it is assumed to be 1.) Note that this does not allow one to decrease the iteration variable; it may only increase. This statement should be contrasted with the two iteration statements in MAD:

```
THROUGH C, FOR X = X, -X/2, ARG .E. A(Y) .AND. Y .LE. N
```

which appeared in the program for SEARCH. in Chapter 7 (Figure 7.3*b*), and

```
THROUGH LOOP1, FOR VALUES OF D = 50, 25, 10, 5, 1
```

which was used in Figure 3.2, the program for the change problem. In order to achieve the action of the first of these MAD iteration statements, one would have to write in FORTRAN:

```
1  IF (ARG - A(LY)) 3, 2, 3
2  IF (LY - N) 4, 4, 3
3  [Some block]
   [of program]
   LX=LX/2
   GO TO 1
4  [The next statement]
```

For the second MAD iteration statement, one would probably write in FORTRAN:

```
      DIMENSION N(5)
      N(1) = 50
      N(2) = 25
      N(3) = 10
      N(4) = 5
      N(5) = 1
      DO 1 I = 1,5
      D = N(I)
      [Some block]
      [of program]
1     CONTINUE
```

Although FORTRAN allows one to write function-definition programs, these may be written only as *external* functions or as *one-line* internal functions, such as the one-line definition of MODULO. in Figure 6.12:

```
INTERNAL FUNCTION MODULO.(Y,Z) = Y - (Y/Z) * Z
```

Moreover, the one-line definition, if used, *must* be at the beginning of the program. The flexible dimension information which we provided in MAD does not exist in FORTRAN, either. All subscripts of vectors and arrays must have positive integer values starting with 1, and once an array is declared to be two-dimensional, it must have exactly two subscripts each time its name is used. Subscripts must

have the form

$$c_1 * v + c_2 \qquad \text{or} \qquad c_1 * v - c_2$$

where c_1 and c_2 are integer constants and v is a nonsubscripted variable. (Either c_1 or c_2 may be omitted, or c_2 may appear alone.) Examples of the use of such subscripts are B(3 * I + 1), A(4), and C(I). Unfortunately, this form must be adhered to so strictly that it is incorrect to write B(1 + 3 * I). With this restricted form of subscript, the MAD statement

```
A(T(I),J) = A(T(I),J) - A(T(I),K) * A(T(K),J)
```

which appeared in the program for the Jordan method (Figure 10.13) would have to be written as follows:

```
M1 = LT(I)
M2 = LT(K)
A(M1,J) = A(M1,J) - A(M1,K) * A(M2,J)
```

Finally, FORTRAN does not allow expressions to contain both integer and noninteger terms. Thus, if A is a noninteger variable, then

```
A + 2
```

would be considered an error in FORTRAN, while MAD would convert the 2 to the noninteger form 2.0 automatically.

These examples will serve to illustrate the differences between the MAD and FORTRAN languages. There are differences between the translators for these languages, also. On those computers for which there exist both MAD and FORTRAN translators, the MAD translator is 4 to 20 times as fast as the FORTRAN translator, thus saving a great deal of computer time during translation. On the other hand, the FORTRAN translators generally produce a better hardware version of the program than MAD, in that the FORTRAN version usually has up to 20 per cent fewer instructions, and it may execute (i.e., actually solve the problem) in half the time. Of course, the time needed to run any program is the sum of the translation time and the execution time.

Here we see the reason for the existence of several algebraic languages. Each language (together with its translators) must be judged

according to several criteria, and no one of them as yet excels in all areas. These criteria include (1) completeness of the language, in terms of the ease with which algorithms may be described, (2) efficiency of the translator in doing its translation to the hardware language, and (3) length and time of execution of the hardware program produced. In addition to these criteria, one must consider the needs of the person who is using the language. A program which is expected to undergo constant revisions, such as a program to evaluate various management-decision policies in a large corporation, must have a translator which does its job very rapidly (since each revision necessitates another translation). The hardware version of this program need not execute as rapidly, however, since each revision will probably not be run on the computer very many times. This is also true for programs written for a great many scientific computations.

On the other hand, programs for payroll computation, inventory control, and other fairly stable applications may be run for many hours without changing anything except the data. Here it is important to obtain an efficient hardware representation of the program, so that the computer will be used as economically as possible. We can tolerate a slower, less efficient translator for such programs, if it is expected to produce good, fast hardware representations.

If we measure MAD and FORTRAN by the three criteria mentioned above, we see that MAD is a very fine language for expressing algorithms, and the translator for MAD is extremely fast. On the other hand, FORTRAN produces a hardware version of the program which runs faster on the computer and takes up less of the storage. Some users will decide that it is to their advantage to use MAD. Others will decide to use FORTRAN.

It should be noted here that a program called MADTRAN has been written entirely in the MAD language to translate statements of the FORTRAN language into corresponding statements of the MAD language.[1] Programs originally written in FORTRAN may now be translated into MAD by the computer, using the MADTRAN program. For example, the statement

```
DO 17 I = 4, 16, 2
```

[1] The program was written by Robert F. Rosin of the University of Michigan.

translates into

```
THROUGH S17, FOR I = 4, 2, I .G. 16
```

and so on. The most difficult part of this particular translation is the recognition of the fact that DO17I is not the name of a variable.

As a final example of the similarities and differences between the MAD and FORTRAN languages, Figure 12.5 shows a FORTRAN version of the program given in Chapter 7 for the binary search algorithm called SEARCH. . To make comparison easier, the original MAD program is reproduced here as Figure 12.4.

```
      EXTERNAL FUNCTION (N,A,ARG,Y,NOT IN)
      INTEGER N, Y, X
      ENTRY TO SEARCH.
      WHENEVER ARG .L. A(1)
          Y = 1
          TRANSFER TO NOT IN
      OR WHENEVER ARG .G. A(N)
          Y = N + 1
          TRANSFER TO NOT IN
      END OF CONDITIONAL
B     THROUGH B, FOR X = 1, X, X .GE. N
      Y = X
      THROUGH C, FOR X = X, -X/2, ARG .E. A(Y) .AND. Y .LE. N
      WHENEVER ARG .L. A(Y) .OR. Y .G. N
          Y = Y - X/2
      OTHERWISE
          Y = Y + X/2
      END OF CONDITIONAL
C     WHENEVER X .E. 1, TRANSFER TO D
      FUNCTION RETURN Y
D     WHENEVER ARG .G. A(Y), Y = Y + 1
      TRANSFER TO NOT IN
      STATEMENT LABEL NOT IN
      END OF FUNCTION
```

Figure 12.4

Since FORTRAN does not allow statement labels or variables of statement label mode as arguments to functions (i.e., external functions), we shall use another variable M, which we shall set to 0 or 1, depending on whether ARG is or is not in the table, respectively. The calling program will have to test the value of M on the return

to see whether the argument was found in the table or not. Moreover, FORTRAN requires arguments to functions which are arrays to be dimensioned in the function exactly as they are in the calling program. Since the calling program for a function intended for general use is usually unknown, this has hampered somewhat the writing of general functions in FORTRAN. The name LSRCH is again due to the mode convention in FORTRAN. Since the value of Y is returned as the value of the search if ARG is in the table, the name of the function must be of integer mode.

```
      FUNCTION LSRCH(N,A,ARG,LY,M)
      DIMENSION A(100)
      IF (ARG - A(1)) 1, 19, 3
 1        LY = 1
 2        M = 1
          GO TO 16
 3    IF (ARG - A(N)) 5, 20, 4
 4        LY = N + 1
          GO TO 2
 5    LX = 1
 6    IF (LX - N) 7, 8, 8
 7        LX = 2 * LX
          GO TO 6
 8    LY = LX
 9    IF (LY - N) 10, 10, 11
10    IF (ARG - A(LY)) 11, 15, 12
11    LY = LY - LX/2
      GO TO 13
12    LY = LY + LX/2
13    IF (LX - 1) 17, 17, 14
14    LX = LX/2
      GO TO 9
15    M = 0
16    LSRCH = LY
      RETURN
17    IF (ARG - A(LY)) 2, 2, 18
18    LY = LY + 1
      GO TO 2
19    LY = 1
      GO TO 15
20    LY = N
      GO TO 15
      END
```

Figure 12.5

Additional information about FORTRAN is available in IBM reference manuals C28-6000-1 and C28-6003-1.

12.3 THE ALGOL LANGUAGE

The other important language we shall discuss is ALGOL (Algorithm Oriented Language). This language was developed by representatives of computer organizations and mathematicians from Europe and the United States in an attempt to standardize as much as possible the way in which algorithms are communicated from one person to another. As a result of several meetings in 1958 and 1960, a description of the reference language was published.[1] This reference language is the standard with which individual interpretations of the statements of the language may be compared. Such interpretations may actually depart from this reference language to suit publication practices, such as writing a_6 instead of $a[6]$. Other departures may have to be made because of the hardware of a computer. For example, if brackets are not included among the characters acceptable to a particular computer, the writers of the translator may require in the language that $a[6]$ be written as $a(6)$. We shall consider here only the reference language.

ALGOL is a relatively recent development as an algebraic language. It is a very complete and general language, allowing one to write extremely complicated programs, in some ways going beyond the features that are built into MAD. This generality has made it difficult to create translators for the ALGOL language which do the translation rapidly enough. Although some success appears imminent in this area, ALGOL has been more successful as a publication language. Several journals have adopted it as their official language for publishing algorithms, and as more people become accustomed to reading it, its use should become quite widespread.

Another area in which ALGOL has had a great influence is in the design of other algebraic languages, such as MAD. Many of the features in MAD were patterned after the corresponding features in ALGOL, in an effort to make translation from ALGOL to MAD and from MAD to ALGOL as direct as possible. Thus, while there

[1] *Comm. Assoc. for Computing Machinery*, vol. 3, no. 5, May, 1960,

are differences between the two languages, they are far outweighed by the similarities. For example, in MAD there are two forms of the iteration statement. One form uses a list of values of the iteration variable, as in the statement

```
        THROUGH LOOP, FOR VALUES OF D = 50, 25, 10, 5, 1
LOOP    A(D) = B(D)
```

The other form uses a termination condition, as in the sequence

```
        THROUGH LOOP1, FOR K = 1, 2, K .G. N
LOOP1   A(K) = B(K)
```

In the ALGOL language there is one very general iteration statement which includes both of the MAD forms as special cases. This ALGOL statement has the form

$$\textbf{for } x := \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3 \textbf{ do } \mathcal{S}$$

where $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$ represent the various sets of values over which x ranges and $\mathcal{S}$ is the computation in the scope of the iteration. (The scope $\mathcal{S}$ may be a single statement or a sequence of statements, as we shall explain shortly.) Now, $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$ may be any expressions, so that x takes on one value after another as in the first MAD form. An example of this is the ALGOL statement

$$\textbf{for D} := 50, 25, 10, 5, 1 \textbf{ do A[D]} := \text{B[D]};$$

Another form which $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$ may have is

$$\mathcal{E} \textbf{ while } \mathcal{B}$$

where $\mathcal{E}$ is an expression to be substituted for x each time through the iteration and $\mathcal{B}$ is a Boolean expression which serves as a *continuation* condition. This means that the scope is repeatedly executed *as long as* $\mathcal{B}$ is *true*. This form is similar to the second MAD form shown above, except that MAD uses a termination condition, and the scope is repeatedly executed *until* $\mathcal{B}$ is *true*. An example of this form is the ALGOL sequence

$$x := 1; \textbf{ for } x := x + 1 \textbf{ while } x < n + r \textbf{ do } y[x] := x - r;$$

A third form in which $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$ may occur is the following:

$$\mathcal{F}_1 \textbf{ step } \mathcal{F}_2 \textbf{ until } \mathcal{F}_3$$

where $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ are expressions representing the initial value of the iteration variable x, the increment to be added to x after each execution of the scope, and the final value for x, respectively. This is illustrated by rewriting the preceding example as follows:

$$\textbf{for } x := 1 \textbf{ step } 1 \textbf{ until } n + r \textbf{ do } y[x] := x - r;$$

In any iteration statement there may be several such descriptions of ranges of values for the iteration variable, and they may be in any or all of the forms we have seen. Thus, one might very well have

$$\textbf{for } x := 1, 3, 5, x + 2 \textbf{ while } x < 20, x + 3 \textbf{ step } 3 \textbf{ until } 35, 39, 40 \textbf{ do } y[x] := 0;$$

One can accomplish the same thing in MAD by breaking this statement into several of the MAD iteration statements, but it is possible to write it in one statement in ALGOL.

In ALGOL the concept of *scope* of a statement, such as an iteration statement or conditional statement, is indicated by using a kind of "statement parentheses," e.g., the words **begin** and **end.** Thus, the following compound conditional in MAD

```
WHENEVER I .G. J
    B(I) = C(I) - 3
    X = 0
    Y = 1
OR WHENEVER I .E. J
    B(I) = C(I)
    X = 1
    Y = 0
OTHERWISE
    B(I) = C(I) + 3
    X = 0
    Y = 0
END OF CONDITIONAL
```

would appear in ALGOL as

```
if i > j then begin B[i] := C[i] − 3;  x := 0;  y := 1
end else if i = j then begin B[i] := C[i];  x := 1;
y := 0 end else begin B[i] := C[i] + 3;
x := y := 0 end;
```

This example shows some of the other differences between ALGOL and MAD, such as the use of the term **if** instead of WHENEVER, **else if** instead of OR WHENEVER, and **else** instead of OTHERWISE.

There are other differences, of course, but most of them cannot be described unless we include a complete description of the ALGOL language. The more basic differences mentioned earlier between MAD and FORTRAN do not exist here. To illustrate this point, the program for SEARCH., which appeared in Section 12.2 in both MAD and FORTRAN, is given in Figure 12.6 in the ALGOL language. Note that the term **procedure** is used to represent an internal function. ALGOL does not have external functions. If we want to use the same variable name in a procedure and in the program which calls on the procedure, but still have no connection between the two occurrences of the name, we would have to declare the procedure to be a separate *block* of the program. MAD achieves

```
integer procedure  Search (n, A, arg, y, not in);
value n, arg; integer n, y; array A;
label not in;
begin integer x; if arg < A(1) then
        begin y := 1;  go to not in end
    else if arg > A(n) then
        begin y := n + 1;  go to not in end;
    for x := 1, 2 × x while x < n do;
    y := x
    for x := x, x ÷ 2 while arg ≠ A(y) ∨ y > n do
        begin if arg < A(y) ∨  y > n then
            y := y − x ÷ 2
        else y := y + x ÷ 2;
        if x = 1 then go to mark end;
    go to out;
mark: if arg > A(y) then y := y + 1;
    go to not in;
out: Search := y end
```

Figure 12.6

this effect by translating external functions separately from the main program. A variable name used in an external function in MAD has no relationship to the same name used in the calling program.

The symbol $\vee$ which appears in the ALGOL program in Figure 12.6 is the Boolean connective *or*, represented as .OR. in MAD. In the first **for** statement of the program, there is no statement to be executed after the word **do** (i.e., before the semicolon). This corresponds to the empty scope which we obtained in the MAD program by labeling the iteration statement with its own scope indicator B.

12.4 CONCLUSIONS

We have seen that there are other languages beside our MAD language, and we have made some comparisons with two of them. Each has its strong points and its weak points, and eventually all of them will be replaced by better languages. The term *better* will imply different criteria to different people, and we may expect that the languages yet to be developed will again have their advantages and disadvantages, which may be quite different from those we have discussed. The general form of these languages will probably change slowly, however. We may find statements being executed concurrently, for example, once the hardware needed to do this is more available. But the lessons learned here about the description of algorithms, the design of a language, and the various techniques, such as sorting and searching, should be useful for some time, even if the languages (and machines) do change.

APPENDIX A

SUMMARY OF THE RULES OF THE LANGUAGE

FOR EASIER REFERENCE, the rules of the language and some of the definitions are summarized here. They are listed according to the order in which they appear in the book. If there is any question as to the interpretation of any of the rules or definitions given here, the discussion in the appropriate chapter should be consulted.

Chapter 1

DEFINITION: The greatest integer function, denoted $[x]$, assigns to x the largest integer y such that $y \leq x$.

SUBSTITUTION STATEMENT: A statement of the form $\mathcal{V} = \mathcal{E}$, where $\mathcal{V}$ is the name of a variable (possibly subscripted) and $\mathcal{E}$ is any arithmetic expression. (This is extended in Chapter 9 to include Boolean variables on the left and Boolean expressions on the right.)

Chapter 2

DEFINITION: The *name of a variable* contains one to six capital letters or digits, the first of which is a letter.

DEFINITION: A *numeric constant* contains one to eleven digits with or without a decimal point, and with or without a sign. The constant is an *integer* if it does not contain a decimal point. An *alphabetic*

constant contains up to six characters (any character except the dollar sign) preceded and followed by dollar signs.

DEFINITION: An *arithmetic expression* is defined as follows:

1. Numeric and alphabetic constants and variable names are arithmetic expressions.
2. If $\mathfrak{A}$ and $\mathfrak{B}$ are already known to be arithmetic expressions, then so are the combinations $\mathfrak{A} + \mathfrak{B}$, $\mathfrak{A} - \mathfrak{B}$, $+\mathfrak{A}$, $-\mathfrak{A}$, $\mathfrak{A} * \mathfrak{B}$, $\mathfrak{A}/\mathfrak{B}$, $\mathfrak{A}$ .P. $\mathfrak{B}$, .ABS. $\mathfrak{A}$, and $(\mathfrak{A})$.
3. The only arithmetic expressions are those which are generated by (1) and (2).

DECLARATIONS:

INTEGER A,B

This indicates that A and B are variables whose values are integers.

NORMAL MODE IS INTEGER

This indicates that all variables (unless declared otherwise) are of integer mode.

DEFINITION: A *basic Boolean expression* consists of one of the relations $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ preceded and followed by any arithmetic expressions. (This is extended in Chapter 9 to include Boolean variables.)

DEFINITION: A *Boolean expression* is defined as follows:

1. Basic Boolean expressions are Boolean expressions.
2. If $\mathfrak{P}$ and $\mathfrak{R}$ are already known to be Boolean expressions, then so are $(\mathfrak{P})$, $\mathfrak{P}$ .AND. $\mathfrak{R}$, and $\mathfrak{P}$ .OR. $\mathfrak{R}$.
3. The only Boolean expressions are those which are generated by (1) and (2).

PARENTHESIS CONVENTION: When parentheses are omitted in an expression, the order in which operations are performed is given by the list:

.ABS.
.P.
− (unary)
*, /
+, − (binary)
.E., .NE., .L., .LE., .G., .GE.
.AND.
.OR.

If two operations with the same position in this list occur in an expression, the left-most is executed first.

Chapter 3

THE SIMPLE CONDITIONAL STATEMENT:

WHENEVER $\mathfrak{B}$, $\mathfrak{S}$

where $\mathfrak{B}$ is a Boolean expression and $\mathfrak{S}$ is a statement to be executed if and only if $\mathfrak{B}$ has the value *true*. (Certain statements, such as the iteration statements and the END OF PROGRAM statement, may not be used for $\mathfrak{S}$.)

THE TRANSFER STATEMENT:

TRANSFER TO $\mathfrak{a}$

where $\mathfrak{a}$ is a statement label.

THE END OF PROGRAM STATEMENT:

END OF PROGRAM

This must be the last statement in any program (except in external functions, where END OF FUNCTION is used; see Chapter 6).

AN ITERATION STATEMENT:

THROUGH $\mathfrak{a}$, FOR VALUES OF $\mathfrak{v} = \mathfrak{E}_1, \mathfrak{E}_2$

This indicates that the scope of the iteration statement (i.e., the block of statements following the iteration statement down to and including the statement labeled $\mathfrak{a}$) is executed for each of the values $\mathfrak{E}_1$, $\mathfrak{E}_2$ of the variable $\mathfrak{v}$. (There may be more than two values.)

THE DIMENSION STATEMENT:

DIMENSION Q(50)

A block of storage locations is set aside for Q(0), . . . , Q(50). (This is extended in Chapter 10.)

Chapter 4

THE COMPOUND-CONDITIONAL STATEMENT:

WHENEVER $\mathfrak{B}_1$
. . . . . .
OR WHENEVER $\mathfrak{B}_2$
. . . . . .
OTHERWISE
. . . . . .
END OF CONDITIONAL

If $\mathfrak{B}_1$ has the value *true*, the block of statements which follows is executed, and then a transfer is made to the first statement following the END OF CONDITIONAL statement. If $\mathfrak{B}_1$ is *false*, $\mathfrak{B}_2$ is evaluated, etc. The OTHERWISE statement is treated as an OR WHENEVER statement containing a Boolean expression which is always *true*. (The OR WHENEVER and OTHERWISE sections need not occur.)

Chapter 5

ANOTHER ITERATION STATEMENT:

THROUGH $\mathfrak{A}$, FOR $\mathfrak{V} = \mathfrak{E}_1, \mathfrak{E}_2, \mathfrak{B}$

This indicates that the variable $\mathfrak{V}$ is to be set equal to the value of the expression $\mathfrak{E}_1$. If $\mathfrak{B}$ is *true*, then the scope of this iteration statement is not executed. If $\mathfrak{B}$ is *false*, the scope is executed. Then $\mathfrak{V}$ is incremented by $\mathfrak{E}_2$, and $\mathfrak{B}$ is evaluated again. The scope is repeatedly executed, and $\mathfrak{V}$ is incremented, until $\mathfrak{B}$ has the value *true*.

DEFINITION: *a is congruent to b modulo c* (written $a \equiv b \bmod c$) if $a - b$ is a multiple of c. *a is the residue of b modulo c* if $a \equiv b \bmod c$ and $0 \leq a < c$.

Chapter 6

THE EXTERNAL-FUNCTION STATEMENT:

EXTERNAL FUNCTION $(\mathfrak{V}_1,\mathfrak{V}_1,\mathfrak{F}_1.,\mathfrak{F}_2.)$

This indicates that the program which follows (and which ends with

the statement END OF FUNCTION) is an external-function definition, i.e., a program to be called by another program. The arguments may be variables, names of functions, and/or statement labels. Occurrences of variables in an external function have no relation at all to variables with similar names in any other program. The entry point and name of the function are indicated by the ENTRY TO statement, and the value returned as the value of the function is indicated by the FUNCTION RETURN statement.

THE INTERNAL-FUNCTION STATEMENT:

INTERNAL FUNCTION ($\mathcal{V}_1,\mathcal{V}_2,\mathcal{F}_1.,\mathcal{F}_2.$)

An internal function is similar to an external function except that it is part of another program and may be called only by that program. Also, variables which are referred to in the internal-function definition and which are not arguments are assumed to be part of the calling program as well. There is a one-line form of the internal-function definition:

INTERNAL FUNCTION SQUARE.(X) = X * X

If the one-line form is not used, the definition program is similar in form to that of an external function.

Chapter 7

THE EXECUTE STATEMENT:

EXECUTE $\mathcal{F}_1.(\mathcal{V}_1,\mathcal{V}_2)$

This statement indicates that the function-definition program for $\mathcal{F}_1.$ is to be executed. Then the next statement in the program is executed. There may be several arguments in such a function call, or there may be none. The arguments may be expressions, function names, and/or statement labels.

DECLARATION:

STATEMENT LABEL $\mathcal{B}$

This declares $\mathcal{B}$ to be a statement label even though it does not appear explicitly as the label of some statement.

Chapter 8

THE INPUT STATEMENT:

READ DATA

This statement causes data to be read into storage until a terminating mark (*) is encountered. Each input value is identified by the name of the variable for which it is the value.

THE OUTPUT STATEMENT:

PRINT RESULTS $\mathcal{E}_1$, $\mathcal{E}_2$

This statement causes the values of the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ to be printed. In this case the definition of an alphabetic constant is extended to include more than six characters between dollar signs to allow headings and comments to be printed.

Chapter 10

AN EXTENDED DIMENSION STATEMENT:

DIMENSION Q(50,B(3))

This declaration indicates that storage is to be set aside for Q(0), . . . , Q(50), but additional dimension information about the array Q will be found in the *dimension vector* for Q, e.g., B(3), B(4), ... In B(3) will be the number of subscripts to be used with the name Q, in B(4) the single subscript to be used for the base element Q(1,1, . . . ,1), in B(5) the number of columns in the array Q, etc. The vector B is automatically declared to be of integer mode.

THE VECTOR-VALUES STATEMENT:

VECTOR VALUES $\mathcal{V} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$

The vector $\mathcal{V}$ is to be preset to the constant values $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ (all of the same mode). The vector $\mathcal{V}$ is automatically declared to have the mode of its values, and it is dimensioned large enough to accept these values. There may be several constants in such a statement.

APPENDIX B

TRANSLATION TO FORTRAN

AS WE POINT OUT in Chapter 12, there are many similarities between the various computer languages, such as MAD, ALGOL, FORTRAN, GAT, and IT. The language which is most often encountered on computers at present is FORTRAN, and in this appendix we shall outline some rules for transforming a program written in the language developed here into FORTRAN. This will enable those readers who have the FORTRAN language available on a computer (but not the MAD language) to write programs in our language, translate them to FORTRAN, and run them on that computer.[1]

If this procedure is to be followed, i.e., programs will be written with a translation to FORTRAN expected, certain precautions should be taken, since some of the more powerful features in our language do not translate easily (or, in many cases, at all) into FORTRAN. We shall list here, by chapters, the rules for translating the various elements of our language into FORTRAN, indicating as we go along which features should be avoided to allow the translation to be made. These rules were chosen to make the translation as automatic as possible, and they may not lead to the most elegant FORTRAN pro-

[1] Although there are many versions of FORTRAN, the most commonly available is FORTRAN II, which we shall use.

gram that one could write. The notes which follow for each chapter assume in each case that the chapter has been read. A complete example is given in Figure B.1 at the end of this appendix.

Chapter 1

The substitution statement may be translated directly to FORTRAN, subject to the rules which follow in the comments on Chapter 2.

Chapter 2

1. Delete INTEGER and NORMAL MODE IS INTEGER statements.

2. All integer variables must begin with one of the letters I, J, K, L, M, or N. All noninteger variables must not begin with one of these letters.

3. Alphabetic constants may be translated to FORTRAN by dropping the dollar signs and prefixing a count of the number of characters in the constant and the letter H. Thus, AB becomes 2HAB. This should be used with great care, if at all, because such alphabetic constants are assumed to be *noninteger* in FORTRAN, and it is easy to use them incorrectly.

4. Integer division agrees with the greatest-integer function on nonnegative quotients, so that $7/3 = 2$, but the fractional part of negative quotients is truncated, so that $-7/3 = -2$.

5. Any one expression may not contain both integer terms and noninteger terms. If at least one variable or constant is noninteger, then each integer subexpression should be enclosed in parentheses and preceded by FLOATF, e.g., FLOATF(I1). Also, in this case, each integer constant should be followed by a period to make it a noninteger constant.

6. Replace .P. everywhere by two asterisks, so that A .P. B becomes A **B. (If the value of A happens to be negative, FORTRAN computes $|A|^B$.) Exponentiation is an exception to (5) above, in that B may be an integer expression even if A is not. If A is an integer expression, however, B must also be an integer expression.

7. At each occurrence of .ABS., enclose in parentheses the expression to which .ABS. applies, if it is not already enclosed in paren-

theses, and replace .ABS. by ABSF if the expression is noninteger, otherwise by XABSF.

8. There are no Boolean expressions in FORTRAN. Rules for translating them from our language are given below (see notes on Chapter 3).

Examples of the translation of expressions are:

From	*To*
A(5 * J + 10)	A(5 * J + 10)
B .P. I + J	B ** I + FLOATF(J)
C + .ABS.(X + Y)	C + ABSF(X + Y)

where only I and J are assumed to be of integer mode in MAD.

Chapter 3

1. The TRANSFER TO statement translates directly into the statement GO TO, but FORTRAN only allows integer statement labels less than 32,768. (If translation to FORTRAN is anticipated, labels should be chosen of the form S12, so that in the translation the first letter may be dropped.)

2. In FORTRAN only one conditional statement is available, of the form

IF (A − B) 6, 7, 8

where the expression in parentheses (in this example, A − B) is compared to zero and the next statement executed is (in this example) statement 6, 7, or 8, depending on whether the expression is less than, equal to, or greater than zero. Table B.1 shows how to translate our simple conditional to FORTRAN in case the Boolean expression is a *basic Boolean expression*, i.e., the connectives .AND. and .OR. are not involved. In each case, $\S_1$ is the statement to be executed if the Boolean expression is *true* and $\S_2$ is the next statement in the program. In the FORTRAN version, T and F represent the statement labels (which must be numeric) of the *true* and *false* jumps, respectively. If the Boolean expression involves the .AND. and .OR. connectives, Table B.2 shows how to reduce it to its component parts. By applying these rules (and those in Table B.1 for basic Boolean expressions),

one may translate every simple conditional statement into FORTRAN. In Table B.2, statement T_1 represents the *true* situation for the Boolean expression $\mathfrak{B}_1$. If the *true* outlet for $\mathfrak{B}_2$ must be different from T_1, T_2 is used. The same is true for F_1 and F_2. (In an actual case, these would be replaced by statement numbers.) The

Table B.1

WHENEVER X .L. Y, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF (X − Y) T, F, F T $\mathfrak{S}_1$ F $\mathfrak{S}_2$
WHENEVER X .LE. Y, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF (X − Y) T, T, F T $\mathfrak{S}_1$ F $\mathfrak{S}_2$
WHENEVER X .G. Y, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF (X − Y) F, F, T T $\mathfrak{S}_1$ F $\mathfrak{S}_2$
WHENEVER X .GE. Y, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF (X − Y) F, T, T T $\mathfrak{S}_1$ F $\mathfrak{S}_2$
WHENEVER X .E. Y, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF (X − Y) F, T, F T $\mathfrak{S}_1$ F $\mathfrak{S}_2$
WHENEVER X .NE. Y, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF (X − Y) T, F, T T $\mathfrak{S}_1$ F $\mathfrak{S}_2$

jumps represented by TF, TF, TF in Table B.2 are to be chosen from the appropriate entry in the table for basic Boolean expressions, Table B.1.

Table B.2

WHENEVER $\mathfrak{B}_1$ .OR. $\mathfrak{B}_2$, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF ($\mathfrak{B}_1$) TF, TF, TF F_1 IF ($\mathfrak{B}_2$) TF, TF, TF T_1 $\mathfrak{S}_1$ F_2 $\mathfrak{S}_2$
WHENEVER $\mathfrak{B}_1$ .AND. $\mathfrak{B}_2$, $\mathfrak{S}_1$ $\mathfrak{S}_2$		IF ($\mathfrak{B}_1$) TF, TF, TF T_1 IF ($\mathfrak{B}_2$) TF, TF, TF T_2 $\mathfrak{S}_1$ F_1 $\mathfrak{S}_2$

As an illustration of the use of these tables, we shall translate into FORTRAN the statements

```
WHENEVER X .L. Y + 3 .AND. I .GE. R, I = I + 1
TRANSFER TO S13
```

where X, Y, and R are noninteger variables and I is an integer variable. The result is

```
    IF (X - Y - 3.) 1, 2, 2
1   IF (FLOATF(I) - R) 2, 3, 3
3   I = I + 1
2   GO TO 13
```

3. The iteration statement which contains a sequence of values of the iteration variable may be translated by using the *computed* GO TO statement in FORTRAN. The following illustration will show what is done in the translation. Here S_1 and S_2 form the scope of the iteration, and S_3 is the next statement to be executed.

```
    THROUGH S12, FOR VALUES OF V = X, X + 4., X - 4.
    S1
S12 S2
    S3
```

This is translated to the following FORTRAN sequence:

```
    V = X
    I = 1
    GO TO 30
10  V = X + 4.
    I = 2
    GO TO 30
20  V = X - 4.
    I = 3
30  S1
12  S2
    GO TO (10,20,40), I
40  S3
```

Here it is assumed that I is an integer variable which is introduced into the FORTRAN program just for this purpose. Its value picks out which statement in its list is to be executed next after the GO TO

statement. Every statement that may be selected by I must appear in the list.

4. The DIMENSION statement need not be changed in the translation to FORTRAN, but the zero subscript *must not be used* for any vector. Moreover, the only legal subscript expressions in FORTRAN are those of the form $c_1 * v + c_2$ or $c_1 * v - c_2$, where c_1 and c_2 are integer constants or are omitted and v is a nonsubscripted integer variable. Thus $3 * I1 + 5$ is a legal subscript, but $5 + 3 * I1$ is not. The DIMENSION statement must precede the first use of the array.

5. The END OF PROGRAM statement is translated into the statement END. (Individual computing installations may require slight variations of this statement.)

Chapter 4

1. The compound conditional does not exist in FORTRAN, but it may be translated by using the same tables and conventions given above for the simple conditional. The following example will illustrate the translation procedure. We wish to translate the compound conditional:

WHENEVER $\mathfrak{B}_1$
$\mathfrak{S}_1$
OR WHENEVER $\mathfrak{B}_2$
$\mathfrak{S}_2$
OTHERWISE
$\mathfrak{S}_3$
END OF CONDITIONAL
$\mathfrak{S}_4$

The corresponding FORTRAN sequence is

```
      IF (B1) TF, TF, TF
T1    S1
      GO TO 3
F1    IF (B2) TF, TF, TF
T2    S2
      GO TO 3
F2    S3
3     S4
```

A specific example of the treatment of the compound conditional would appear as follows:

```
WHENEVER X .L. Y + 3.
    X = B + 1.
    TRANSFER TO S7
OR WHENEVER X .E. Y + 3. .OR. I .GE. R
    X = 2. * B
    TRANSFER TO S8
OTHERWISE
    TRANSFER TO S9
END OF CONDITIONAL
```

This is translated to the following FORTRAN sequence:

```
      IF (X - Y - 3.) 1, 2, 2
1         X = B + 1.
          GO TO 7
2     IF (X - Y - 3.) 3, 4, 3
3     IF (FLOATF(I) - R) 5, 4, 4
4         X = 2. * B
          GO TO 8
5     GO TO 9
```

If the END OF CONDITIONAL statement has a label, replace it by a CONTINUE statement bearing that label (in numeric form).

Chapter 5

The iteration statement of the form

```
      THROUGH S6, FOR I = ε1, ε2, ℬ
      S1
S6    S2
```

where ε_1 and ε_2 are arithmetic expressions and $\mathcal{B}$ is a Boolean expression, may be translated exactly as if it were the sequence

```
      I = ε1
S5    WHENEVER ℬ, TRANSFER TO S1
      S1
S6    S2
      I = I + ε2
      TRANSFER TO S5
S1
```

Although it is good practice to arrange it so that $\mathcal{E}_1$ and $\mathcal{E}_2$ have the same mode as the iteration variable I, i.e., all integer, or all noninteger, we have not insisted on it in our language. FORTRAN does not allow mixed modes in expressions, however; so $\mathcal{E}_2$ must have the same mode as I. Using the same conventions on the conditional as above, the following FORTRAN sequence would result (where T and F would actually be numeric):

```
      I = ℰ1
5     IF(ℬ) TF, TF, TF
F     𝒮1
6     𝒮2
      I = I + ℰ2
      GO TO 5
T
```

As a specific illustration, we consider the following sequence, which is similar to one which occurs in Figure 5.3*b*. Since zero subscripts are not allowed in FORTRAN, however, the example has been modified.

```
S2    THROUGH S2, FOR I = 1, 1, LETTER(K) .E. STAND(I)
      P = I + S
```

This would be translated first into the sequence in our language (see Figure 5.3*a*):

```
      I = 1
S2    WHENEVER LETTER(K) .E. STAND(I), TRANSFER TO S1
      I = I + 1
      TRANSFER TO S2
S1    P = I + S
```

Translating to FORTRAN, we obtain the following:

```
      I = 1
2     IF (LETTER(K) − LSTAND(I)) 3, 1, 3
3     I = I + 1
      GO TO 2
1     P = I + LS
```

Note that some of the names have been changed here to make them conform to the convention on names of integer variables.

One common form of this iteration statement may be translated directly into the FORTRAN DO statement. If I is a nonsubscripted integer variable, *and* if $\mathcal{E}_1$ *and* $\mathcal{E}_2$ are each either unsigned integer constants or nonsubscripted integer variables, *and* if $\mathcal{B}$ is the basic Boolean expression I .G. M, where M is either an unsigned integer constant or a nonsubscripted integer variable, *and* if the value of I is not changed in the loop, then the statement

THROUGH S2, FOR I = $\mathcal{E}_1$, $\mathcal{E}_2$, I .G. M

may be translated to the FORTRAN statement

DO 2 I = $\mathcal{E}_1$, M, $\mathcal{E}_2$

If $\mathcal{E}_2$ is omitted, it is assumed to be 1. Thus, the statement

THROUGH S1, FOR K = 1, 1, K .G. N

which is similar to the third statement in Figure 5.3*b*, is translated to

DO 1 K = 1, N

It must be noted that in our language the Boolean expression (i.e., termination condition) is tested before the scope of the loop is executed, so that in some cases the scope may not be executed at all. In FORTRAN the test is made after the scope is executed, so that the scope *must* be executed at least once. Moreover, if a FORTRAN DO statement executes until the termination condition is satisfied, the iteration variable may not be assumed to have any particular value; i.e., it is undefined.

Chapter 6

Internal functions, other than the one-line internal function, may not be used if translation to FORTRAN is needed, since only the one-line form occurs in FORTRAN. The one-line form is translated according to the rules:

1. Delete the words INTERNAL FUNCTION and the period after the name of the function.
2. Move the one-line definition to the beginning of the program or at least in front of the first executable statement.
3. Names of functions and statement labels may not be used as arguments.

External functions may be translated to FORTRAN by using the following rules:

1. Replace the words EXTERNAL FUNCTION by the word FUNCTION followed by the name of the function (without the period). The arguments remain as before.
2. There may be only one name for the function.
3. Delete the ENTRY TO statement. The entry point will be assumed to be at the first executable statement.
4. The statement FUNCTION RETURN ε, where ε is the expression whose value is to be returned, may be translated to the FORTRAN sequence:

```
NAME = ε
RETURN
```

where NAME is the name of the function.

5. The last letter of the name must not be an F if there are more than three letters in the name.
6. The END OF FUNCTION statement is translated into the statement END.
7. All vectors and arrays (see Chapter 10) used as arguments must be dimensioned in the function definition exactly as the variables which are to be substituted for them are dimensioned in the calling program.
8. No statement labels or function names may be used as arguments.
9. If the value of the function is an integer, the name of the function must begin with I, J, K, L, M, or N. If the value of the function is a noninteger, the first letter of the name must not be one of these letters.

Calls for functions (internal or external) are translated to FORTRAN by deleting the period after the function name. Remarks (recognized by the letter R just before the statement) should be translated to *comments*, indicated by a letter C in the first column of the statement number field. The letter R should be deleted.

Chapter 7

The EXECUTE statement is translated by replacing the word EXECUTE by the word CALL and deleting the period after the name of the function.

If a function-definition program contains the statement FUNCTION RETURN without an expression following, it should be replaced by the FORTRAN statement

```
RETURN
```

The CONTINUE statement is not changed in the translation.

Chapter 8

Alphabetic data may not be used. The statement READ DATA should be translated into the FORTRAN sequence

```
    READ 5, A, B, C, (D(I), I = 1, 6)
5   FORMAT (9F8.0)
```

or into the sequence

```
    READ INPUT TAPE 7, 5, A, B, C, (D(I), I = 1, 6)
5   FORMAT (9F8.0)
```

where the first assumes that a card reader is used and the second assumes that the data have been transcribed to a magnetic tape called tape "7." In either case, it is assumed that the data are punched into cards originally in nine eight-column fields, with decimal points punched. The notation (D(I), I = 1, 6) is equivalent to our notation D(1), . . . , D(6), which may not be used. Note that the names of the variables receiving values are written into the statement and are not part of the data. If some of the data values are integers (without decimal points), the FORMAT statement must indicate this, as in the example:

```
    READ INPUT TAPE 7, 5, I, A, B, N, M, D(3), D(4)
5   FORMAT (I8, 2F8.0, 2I8, 2F8.0)
```

The PRINT RESULTS statement is exactly analogous to the input statements just described, except that the word READ is replaced by PRINT, the words READ INPUT TAPE are replaced by WRITE OUTPUT TAPE, and the format information should be preceded by the characters 1H0, to control the page spacing. Thus, the statement

```
PRINT RESULTS A, B, C(3), . . . , C(8), J
```

might be translated into

```
      WRITE OUTPUT TAPE 6, 4, A, B, (C(I), I = 3, 8), J
4     FORMAT (1H0, 8F8.0, I8)
```

Alphabetic comments enclosed in dollar signs are translated by entering them directly into the format information, preceded by a count of the number of characters (including blanks) and the letter H, as in the following, where

```
PRINT RESULTS $NO SOLUTION, INPUT WAS A = $,
                                    A, $, B = $, B, $, C = $, C
```

is translated to

```
      WRITE OUTPUT TAPE 6, 4, A, B, C
4     FORMAT (1H0, 27HNO SOLUTION, INPUT WAS A =  ,
                          F8.0, 6H, B =  , F8.0, 6H, C =  , F8.0)
```

In FORTRAN one may not include expressions in the output list. If an expression occurs in a PRINT RESULTS statement, it should be computed in an earlier statement. Thus, our statement

```
PRINT RESULTS A + B, GCD.(I,J), I, J
```

should be translated into the sequence

```
      X = A + B
      I1 = GCD(I,J)
      WRITE OUTPUT TAPE 6, 4, X, I1, I, J
4     FORMAT (1H0, F8.0, 3I8)
```

Chapter 9

Boolean variables and constants should not be used, since they have no analogue in FORTRAN. Also, the statement NORMAL MODE IS BOOLEAN must not be used.

Chapter 10

The DIMENSION statement in FORTRAN does not provide the flexibility that ours does. If translation to FORTRAN is anticipated, the following rules should be observed.

1. An array (or vector) may have one, two, or three (maximum

of three) subscripts of the form $c_1 * v + c_2$ or $c_1 * v - c_2$ as indicated above. All references to that array must have the same number of subscripts, and this number may not change. (An array name which normally has two or three subscripts may be used with one subscript, however.)

2. Zero or negative subscripts may not be used, nor may $c_1 * v$ be zero or negative in the above subscript form. The base element V(1,1) may not be moved, nor is it ever different from V(1).

3. The DIMENSION statement indicates the highest value that *each subscript* may assume. No dimension vector is used.

4. The DIMENSION statement must precede the first use of the name of the array.

As an example, if the array A is to have two subscripts and is to have no more than 20 rows and 15 columns, we would write

```
DIMENSION A(20,15)
```

In our language, this would probably have been written as follows:

```
DIMENSION A(300,ADIM)
VECTOR VALUES ADIM = 2, 1, 15
```

although the last value (i.e., 15) would probably change as the particular set of data came in.

5. The VECTOR VALUES statement does not exist in FORTRAN. To translate the statement

```
VECTOR VALUES R(6) = 1.2, .3, -4.1
```

into FORTRAN, the following sequence should be used at the beginning of the program (but after any one-line internal-function definitions):

```
R(6) =  1.2
R(7) =   .3
R(8) = -4.1
```

6. If V is a vector but the symbol V is used without a subscript, we have agreed to interpret it as V(0). In FORTRAN it is interpreted as V(1). To avoid this conflict, the subscript should not be omitted if translation to FORTRAN is expected. Figure B.1 shows an example of the translation from our language to FORTRAN.

Figure B.1 is the translation of Figure 10.12, the Jordan algorithm for a system of simultaneous linear equations (ignoring division by zero). The system is assumed to have, at most, 19 equations. This is reflected in the DIMENSION statement.

```
C   THIS STATEMENT SETS UP THE
C   DIMENSION INFORMATION FOR THE MATRIX
C   OF COEFFICIENTS
    DIMENSION A(19,20)
C   THE INPUT STATEMENTS WHICH FOLLOW
C   BRING IN THE NUMBER OF EQUATIONS N
C   AND THE MATRIX OF COEFFICIENTS A(1,1), . . . , A(N,N + 1)
 1  READ INPUT TAPE 7, 2, N
 2  FORMAT (I8)
    NP1 = N + 1
    READ INPUT TAPE 7, 3, ((A(I,J), J = 1, NP1), I = 1, N)
 3  FORMAT (9F8.0)
C   THE NEXT STATEMENT BEGINS THE JORDAN ALGORITHM
    DO 4 K = 1, N
    J = N + 1
 7  IF (J - K) 6, 5, 5
 5  A(K,J) = A(K,J)/A(K,K)
    J = J - 1
    GO TO 7
 6  DO 4 I = 1, N
    IF (I - K) 8, 4, 8
 8  J = N + 1
12  IF (J - K) 4, 10, 10
10  A(I,J) = A(I,J) - A(I,K) * A(K,J)
    J = J - 1
    GO TO 12
 4  CONTINUE
    DO 13 I = 1, N
13  WRITE OUTPUT TAPE 6, 14, A(I,N + 1)
14  FORMAT (1H0, F8.0)
    GO TO 1
    END
```

Figure B.1

APPENDIX C

TRANSLATION TO ALGOL

IN THIS APPENDIX, as in Appendix B, we shall consider the rules for translating programs from our language into another language which is quite similar. The ALGOL language (and in particular, ALGOL 60, see Chapter 12) is, or soon will be, available on several computers. We shall use ALGOL 60 as the language to which we shall translate, and the rules will be stated by chapter. In each case, it will be assumed that the chapter has already been read. Anything not mentioned explicitly below need not be changed in the translation to ALGOL.

Chapter 1

Substitution statements are translated by replacing the = sign by the "colon-equal" sign (:=). Expressions on the right side and subscript expressions on the left side of the statement are translated subject to the rules given below (see comments on Chapter 2).

Chapter 2

Numeric constants need not be changed in the translation, except that no noninteger constant may end with a period. Alphabetic

constants are translated to ALGOL by replacing the dollar signs by left and right (single) quotation marks. Operations are unchanged, except that .P. is replaced by an upward arrow (↑), .ABS.(X) by abs(X), * by ×, and I/J by I ÷ J whenever I and J are both of integer mode. Subject to these replacements, arithmetic expressions are translated without further change.

There is no statement in ALGOL corresponding to the statement NORMAL MODE IS INTEGER; so this statement must be deleted. Each variable and function with integer values must be declared **integer** at the beginning of the program, and every Boolean variable (see Chapter 9) must be declared **Boolean.**

Boolean expressions are translated by replacing .E., .NE., .L., .LE., .G., .GE., .OR., and .AND. by =, ≠, <, ≤, >, ≥, ∨, and ∧, respectively.

Chapter 3

In an ALGOL program all statements must be separated by semicolons, even if they appear on separate lines. A statement label must be separated from the statement to which it is attached by a colon. The statement TRANSFER TO A1 is translated to **go to** A1.

The simple conditional

WHENEVER 𝔅, 𝔖

is translated to the ALGOL statement

if 𝔅 **then** 𝔖;

The iteration statement which lists a sequence of values, such as

THROUGH LOOP1, FOR VALUES OF D = 50, 25, 10, 5, 1
LOOP1 Q(D) = 0

would be translated to

for D := 50, 25, 10, 5, 1 **do** Q[D] := 0;

(Note that parentheses *used for subscription* are replaced by brackets.) If the scope of the iteration contains more than one statement, the scope is preceded by **begin** and followed by **end.** (Any time a sequence of statements must be considered as a single block, they

must be preceded and followed by **begin** and **end,** respectively. In particular, the entire program being translated, after the initial declarations such as mode and dimension, should be contained between a **begin-end** pair.) For example, the iteration sequence (involving only integer variables):

```
        THROUGH LOOP2, FOR VALUES OF D = 50, 25, 10, 5, 1
        WHENEVER R .E. 0, TRANSFER TO FINISH
        Q(D) = R/D
LOOP2   R = R - D * Q(D)
```

would be translated into the ALGOL sequence:

```
for D := 50, 25, 10, 5, 1 do
    begin if R = 0 then go to FINISH;
        Q[D] := R ÷ D;
        R := R - D × Q[D] end;
```

Dimension declarations for vectors, such as

```
DIMENSION Q(50)
```

are translated to

```
array Q [0:50]
```

indicating in brackets the lowest and highest subscripts to be used for Q. If Q is an integer variable, the declaration is written

```
integer array Q[0:50]
```

Chapter 4

To translate the compound conditional to ALGOL, we replace the word WHENEVER by **if . . . then,** the words OR WHENEVER by **else if . . . then,** and OTHERWISE by **else.** The END OF CONDITIONAL statement is then deleted. Thus, the compound conditional

```
WHENEVER I .G. K .AND. Z .NE. Y
    I = I - 1
    K = K + 1
OR WHENEVER I .E. K
    I = I + B(I)
```

```
OTHERWISE
    I = I + C(I)
    K = K − C(I)
END OF CONDITIONAL
```

would appear as follows when translated into the ALGOL sequence:

```
if I > K ∧ Z ≠ Y then
    begin I := I − 1;  K := K + 1 end
else if I = K then
    I := I + B[I]
else
    begin I := I + C[I];  K := K − C[I] end;
```

Note that there is no semicolon after the first **end** in this example, since the **else if** which follows is part of the same conditional statement.

Chapter 5

The iteration sequence of the form

```
        THROUGH A, FOR V = ε1, ε2, ℬ
        §1
A       §2
```

may be translated into an ALGOL sequence of the form

```
for V := ε1, V + ε2 while ℬ̄ do
    begin §1; §2 end;
```

where $\bar{\mathcal{B}}$ is the *negation* of the Boolean expression $\mathcal{B}$; i.e., $\bar{\mathcal{B}}$ is obtained from $\mathcal{B}$ by replacing each element of the first column of Table C.1 with the corresponding element in the second column.

Table C.1

.E.	$\neq$
.NE.	$=$
.L.	$\geq$
.LE.	$>$
.G.	$\leq$
.GE.	$<$
.OR.	$\wedge$
.AND.	$\vee$

In the special case where ℬ is V. G. N and $\mathcal{E}_2$ is positive, or ℬ is V. L. N and $\mathcal{E}_2$ is negative, one may write

for V := $\mathcal{E}_1$ **step** $\mathcal{E}_2$ **until** N **do**
begin $\mathcal{S}_1$; $\mathcal{S}_2$ **end;**

For example, the iteration sequence in Figure 5.3*b*,

```
        THROUGH LOOP1, FOR K = 1, 1, K .G. N
LOOP2   THROUGH LOOP2, FOR I = 0, 1, LETTER(K) .E. STAND(I)
        P = I + S
        CODE(K) = KEY(P - 39 * (P/39))
LOOP1   S = S + 5
```

may be translated into the ALGOL sentence

```
for K := 1 step 1 until N do
    begin for I := 0, I + 1 while true do
        if LETTER[K] = STAND[I] then go to next;
    next: P := I + S; CODE[K] := KEY[P - 39 × (P ÷ 39)];
        S := S + 5 end;
```

Note that the inner iteration on I would normally have been translated into the ALGOL statement

for I := 0, I + 1 **while** LETTER[K] ≠ STAND[I] **do**

but in ALGOL any iteration which terminates because the termination condition is satisfied leaves the iteration variable undefined; i.e., we may not assume that it has any particular value. The **true** condition which we used above guarantees that the iteration will continue until the **if** condition is satisfied, thus leaving I with the desired value.

Chapter 6

Internal and external functions are combined into one kind of function in ALGOL called a **procedure.** (The one-line form of internal function does not exist in ALGOL.) To translate either an internal-function (not the one-line form) or an external-function-definition program to ALGOL, the following rules should be applied.

1. If the function returns a value to the calling program, the mode

of the value must be the first word of the procedure. The mode is **integer, Boolean,** or **real** (noninteger).

2. The words INTERNAL FUNCTION or EXTERNAL FUNCTION should be replaced by the word **procedure,** followed by the arguments, enclosed in parentheses.

3. Following the arguments should be a sequence of declarative statements (separated as usual by semicolons) indicating those arguments which are input values to the procedure (e.g., **value** *x*, *y*) and, for every argument that is itself a procedure name, the mode of the value, if any, followed by the word **procedure** and the argument itself. Any argument whose value is changed by the function (i.e., is an output variable) should *not* be declared in a **value** statement, even if it is also an input variable.

4. There may be only one entry to a procedure, from the beginning of the computational sequence. All ENTRY TO statements should be deleted in the translation to ALGOL.

5. The computational sequence following the declarations described above should be preceded by **begin** and followed by LAST: **end,** where LAST is a statement label not used elsewhere in the procedure. Those variables referred to in the procedure, but not intended to be common to the calling program, must be declared immediately following **begin** by giving their modes and/or dimensions.

6. The statement FUNCTION RETURN ε, where ε is the expression whose value is to be returned, is translated into the ALGOL sequence

begin NAME :=ε; **go to** LAST **end;**

where NAME is the name of the procedure.

To translate a one-line internal function, replace the words INTERNAL FUNCTION by the words **integer procedure, real procedure,** or **Boolean procedure,** indicating the mode of the value of the procedure. After the name (without the period) and the arguments enclosed in parentheses, one writes the mode declarations for the arguments and then one statement in the form of a substitution statement, where the left side is just the name of the procedure. Thus,

```
INTERNAL FUNCTION EXCESS.(X,Y)
                    = (X - Y + .ABS.(X -- Y))/2.
```

would be translated into the ALGOL sequence

```
real procedure EXCESS(X,Y); value X, Y; real X, Y;
EXCESS := (X − Y + abs(X − Y))/2.0;
```

As an illustration of the rules given here for translating internal and external functions to ALGOL, Figure C.1 is a translation of Figure 6.12, without the remarks.

```
integer procedure RAND (DIST, R1, XBAR, SIGMA);
value DIST, R1, XBAR, SIGMA; integer DIST, R1;
real XBAR, SIGMA;
begin
    real procedure SIGN(X); value X; real X;
        begin if X ≥ 0 then SIGN := 1.0
            else SIGN := −1.0 end;
    integer procedure MODULO(Y,Z); value Y, Z;
        integer Y, Z; MODULO := Y − (Y ÷ Z) × Z;
    real procedure UNIF;
        begin R1 := MODULO(5 ↑ 15 × R1, 2 ↑ 35);
        UNIF := R1/(2.0 ↑ 35) end;
    if DIST = 0 then begin RAND := UNIF; go to LAST end
    else begin R := UNIF;
        V := sqrt(−2.0 × ln(.5 × (1.0 − abs(1.0 − 2.0 × R))));
        RAND := XBAR + SIGMA × (SIGN(R − .5) ×
            (V − ((.010328 × V + .802853) × V +
            2.515517)/(((.001308 × V + .189269) × V
            + 1.432788) × V + 1.0))) end;
    LAST: end
```

Figure C.1

Chapter 7

To translate the EXECUTE statement into ALGOL, delete the word EXECUTE and the period after the name of the function. If the statement FUNCTION RETURN occurs in a function-definition program without an expression following it, translate it into the ALGOL statement

go to LAST

where LAST is the identifier (i.e., statement label) provided at the final **end** of the procedure.

Chapter 8

There is no provision for input or output statements in ALGOL. In any particular computer representation of ALGOL, there would be provided some definite way to write input and output statements, just as we have developed our own in this book. We shall stipulate then, that the statement READ DATA should remain unchanged in the translation to ALGOL (so that it becomes a call for a procedure named READ DATA, which has no arguments). The PRINT RESULTS statement should be translated by enclosing in parentheses the list of expressions which follows the words PRINT RESULTS. This amounts to a call for a procedure named PRINT RESULTS. Since there is no analogue of the block notation [e.g., Q(1), . . . , Q(17)], this notation should be avoided if translation to ALGOL is anticipated.

Chapter 9

There is no provision in ALGOL for the statement NORMAL MODE IS BOOLEAN, but variables may be declared to be **Boolean.** The Boolean constants 1B and 0B are translated into **true** and **false,** respectively.

Chapter 10

Dimension information for arrays is declared in a manner similar to that used for vectors (see comments on Chapter 3 above), except that the lowest and highest values expected must be specified for each subscript separately. Any of these values may be represented by an integer expression, but the values of all variables in the expression must have been computed before calling on the procedure involved. If the dimension declaration occurs in a main program (i.e., not a procedure), then the subscript values must be integer constants. An example of a declaration is the following:

integer array[−5:20, 0:*m*]

The dimension vector is thus not used at all, and the number of subscripts which must be written for any array is fixed throughout the program.

There is no analogue in ALGOL of the VECTOR VALUES statement. The statement

VECTOR VALUES R(6) = 1.2, .3, −4.1

should be translated into the following sequence (at the beginning of the ALGOL program):

begin R[6] := 1.2; R[7] := .3; R[8] := −4.1 **end;**

The program shown in Figure C.2 is the translation of Figure 10.12 according to the rules given above. Remarks have been omitted, although they could have been carried over preceded by the word **comment.**

```
real array A[1:19, 1:20]; integer N, K, I, J;
begin INPUT: READ DATA;
for K := 1 step 1 until N do
    begin for J := N + 1 step − 1 until K do
        A[K,J] := A[K,J]/A[K,K];
        for I := 1 step 1 until N do
            if I ≠ K then
                for J := N + 1 step − 1 until K do
                A[I,J] := A[I,J] − A[I,K] × A[K,J]
            end;
    for I := 1 step 1 until N do
        PRINT RESULTS (A[I,N + 1]);
    go to INPUT
end
```

Figure C.2

INDEX