FORTRAN

Samuel L. Marateck

FOR TRAN

FOR TRAN

Samuel L. Marateck
New York University

ACADEMIC PRESS NEW YORK SAN FRANCISCO LONDON
A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1

Library of Congress Cataloging in Publication Data

Marateck, Samuel L
FORTRAN.

Includes index.
1. Fortran (Computer program language). I. Title.
QA76.73.F25M37 001.6'424 76-45991
ISBN 0-12-470460-3

PRINTED IN THE UNITED STATES OF AMERICA

To my parents Harold and Rita

Contents

Preface

This book is an outgrowth of notes the author uses in a course in FORTRAN that he teaches to undergraduates at New York University. Its purpose is to teach the student how to program using the FORTRAN language, and it is written for students who have no prior knowledge of computers or programming.

The design of the book has special significance and deserves special comment. The right-hand pages contain pictorial material on programming which most students will readily understand; this is described in detail in "To The Reader." The left-hand pages contain the explanatory text. You will see that strict adherence to this design has resulted in a number of partially filled pages. It has been the author's experience with a book on BASIC, which he wrote and designed in the same way, that in many cases students who visit the computer center and have studied only the right-hand pages, have been able to write and run programs on their first visit. The student should, in order to understand all facets of the programming techniques described, also read the text on the left-hand pages.

There are other features that should help the beginning student. The author usually introduces only one new programming concept per program. Thus many of the programs in the book are first written as a series of smaller programs, each of which serves as a step in understanding the entire larger program.

Easy to understand programs have been used to illustrate the various programming techniques discussed in the book. These programs are the solutions to problems drawn from various disciplines, and all students, whatever their major field, should understand them without difficulty. After students have been presented these programs, they will have the ability to read optional sections in which these same techniques are applied to more advanced programs.

The author also includes examples of common programming mistakes made by beginning students when they are not explicit enough in translating their thoughts into programming instructions. The author has found this type of example to be an effective teaching technique.

Since one of the most difficult parts of learning FORTRAN is the mastering of the input/output statements, these statements are introduced early in the book. Their ramifications are explained as the programs demand it. Thus the reader is exposed to input/output concepts gradually. Then in Chapter 8 the author explains the input/output capability of FORTRAN in great detail.

Sound programming techniques, including programming style and its logical extension, structured programming, aid the programmer in all stages of program writing: the design, writing, debugging, and maintenance of programs. Sound programming techniques are emphasized from the beginning, and they are introduced as they are needed and as the statements being discussed allow their use. Chapter 10 is devoted to the application of structured programming in the design and writing of programs. Also discussed in that chapter are top-down design, top-down testing of programs, and the HIPO diagram.

At some point the FORTRAN programmer should understand round-off errors and significance as they relate to the bit configuration of the computer being used. The greater part of Chapter 12 is devoted to a discussion of these concepts, although the effect of round-off errors is introduced early in the book.

Almost the entire book is devoted to a discussion of American National Standard (ANS) FORTRAN. Since some students will be using Standard Basic FORTRAN, which has fewer instructions and which consequently can be used on machines which have less memory, any statement that is described and is not available in Standard Basic FORTRAN is footnoted as such.

The features of the WATFOR and WATFIV compilers are described as well as features of WATFIV-S (the IF-THEN-ELSE and the WHILE-DO) along with their ANS analogs. Many features of WATFOR/WATFIV/WATFIV-S have been incorporated into the proposed ANS X3.9 FORTRAN language revision (1977), and thus even the reader who will not be using WATFOR/WATFIV/WATFIV-S might want to take more than a casual interest in sections describing these compilers because of their applications to the proposed ANS compilers.

Those teachers who do not intend to use all the chapters in the book may be interested in knowing that the last four chapters—Chapter 10 (structured programming), Chapter 11 (The COMMON and the EQUIVALENCE statements), Chapter 12 (Significance, DOUBLE PRECISION, and COMPLEX numbers), and Chapter 13 (More Input/Output)—can be read in any order.

We have included in the appendices material that all readers might not have use for, e.g., control cards for the IBM 360/370 and time sharing.

It is a pleasure to thank Professor J. T. Schwartz and Professor Max Goldstein for their friendship, their many kindnesses and for their constant support while I was writing this book; and H. David Abrams and Professor Carl F. R. Weiman for acting as a sounding board for many of my ideas. It is also a pleasure to thank Jeffrey Akner and his computer facility staff for their cooperation and Professor Robert Richardson and Professor George Basbas for granting me free time on their computer.

To the Reader

This book has been written on the premise that it is at times easier to learn a subject from pictorial representations supported by text than from text supported by pictorial representations. With this in mind, beginning with Chapter 2 we have used a double page format for our presentation. On the left-hand page (we call it the text page) appears the text, and on the right-hand page (we call it the picture page) appears the pictorial representation, consisting mostly of programs and tables.

Each picture page was written to be as self-contained as possible, so that the reader, if he so desires, may read that page first and absorb the essence of the contents of the entire double page before going on to read the text. The text page consists of a very thorough discussion of the programming techniques presented on the picture page. It refers to parts of the programs and tables on the picture page; when reference is made on the text page to a given line of print on the picture page, that line—whenever it is feasible to do so—is reproduced in the text to promote readability. Students who have a previous background in programming languages and others who understand the picture page completely may find that in some chapters they can skip the text (left-hand) pages and concentrate on the picture pages.

The following techniques are used as aids in making the picture page self-contained:

1. As many as possible of the ideas discussed in the text are illustrated in the programs and tables. The captions beneath these capsulate much of what is said in the text.

2. Words underlined in the captions describe lines underlined in the figures. To illustrate this, part of Fig. 2.2a is reproduced below.

```
VAR1 = 11.4
VAR2 = 20.2
STOP
END
```

Figure 2.2a. In the program the number 11.4 is assigned to the variable VAR1 and 20.2 to VAR2 in the assignment statement.

The statements VAR1 = 11.4 and VAR2 = 20.2 are underlined to show that they are described by the words underlined in the caption. Thus they are both assignment statements.

3. To the right of most programs appears a table that describes what effect certain statements in the program have on the computer's memory. For instance, the following table describes the effect that VAR1 = 11.4 has on the memory:

DESCRIPTION	VAR1
VAR1 =11.4	11.4

We see from the table that this statement caused the number 11.4 to be associated with VAR1 in the computer's memory. The line-by-line analysis afforded by these tables should aid the reader in understanding the program.

1

Introduction to Computers and Programming

1.1. General Remarks

If you wished to categorize the times we live in, in terms of the technological advance that most affects our lives, you would call our age the age of the computer; the computer is all encompassing. For instance, in business some of the uses of computers are for billing, check writing, and inventory control; every large organization uses computers to process its records. In another realm—science and engineering—there are two spectacular examples of the use of computers: (1) The multitude of calculations that enabled the space program to put a man on the moon were done by computers. (2) The pictures taken on Mars by the Viking lander were reconstructed on earth by computers.

In order to solve a problem, the computer follows a set of instructions called a *program*. The people who write these programs are called, appropriately, *programmers*. The form that the instructions in the program take depends on the programming language used. It is the purpose of this book to teach you to program in FORTRAN. The name FORTRAN is taken from FORmula TRANslation. FORTRAN was developed in the 1950s; as the name implies, it was devised as a language for solving mathematical and mathematics-related problems. FORTRAN is used today to solve problems in mathematics, the physical sciences and engineering, the social sciences and linguistics, and other related fields.

The type of computer used in an overwhelming number of applications solves problems and processes information by manipulating digits; hence, this type of computer is called a *digital computer*. A FORTRAN program is run on a digital computer.

So that you may understand the relation of FORTRAN to the computer, we first briefly describe computers. One way of picturing a computer is as a maze of on-off electrical switches connected by wires. Thus you might imagine that if a programmer wished to instruct a computer to do something, he would have to feed it a program composed of a series of on-off types of instructions. As a matter of fact, the first programs were written like this. The type of language that uses this form of instruction is called *machine language*. In its most primitive form, a program written in machine language consists of strings of 0's and 1's, where a zero represents an open switch, and a one represents a closed switch. As you can surmise from this brief description, learning to write programs in machine language can be very difficult. Moreover, even once you master it, writing in machine language can be very tedious. For this reason, computer languages closer in form to the spoken word—in our case, English—and to algebra, have been devised. The most widely used of these languages is FORTRAN. Computer languages closer to the machine are called *low level languages*. An example of a low level language is machine language. Computer languages similar in form to how we express ourselves, either by the spoken word or by mathematical symbols, are called *high level languages*. FORTRAN is a high level language.

A program written in FORTRAN cannot be directly understood by the computer; it must first be translated into machine language. A special program does this. It is called a *compiler*, and is already present in the machine when we feed the computer our program. Once a program has been translated, we say that it has been *compiled*. The original FORTRAN program is called the *source program*, and the translated program is called the *object* program.

FORTRAN has grammatical rules that must be followed by the programmer. These rules are similar to those in English that govern the sequence of words in a sentence, the punctuation, and the spelling—we shall learn these rules in later chapters. Before the compiler translates your program, it checks whether you have written your program instructions according to the grammatical rules. If you make grammatical errors in writing an instruction, don't worry; the compiler has been written so that it will inform you of these. Programmers refer to grammatical errors as *compilation errors* or *compile-time errors*. Your program must be free of compile-time errors before the compiler will translate your program.

We now describe the main components of the computer. Essentially, the computer consists of an *input unit*, a *memory unit*, a *logical unit* (or *arithmetic unit*), an *output unit*, and a *control unit*. Our program is communicated to the computer through the input unit. All the mathematics and decisions in the program are done in the logical unit. The program itself and the numbers it processes are stored in the memory unit. The computer communicates the results of our program to us through the output unit. The control unit directs the activities of the other four units. The control unit and the logical unit are referred to collectively as the *central processing unit* (abbreviated as CPU). The word *hardware* is used to describe the physical components of the computer, such as these units, whereas the word *software* is used to describe the programs. We shall use the word *system* to describe the programs, such as the compiler, that process the programs you write.

1.2. The Keypunch

The first, and still the most widely used, means of communicating a program to the computer is to punch the program instructions on a card. The device we use to do this is called a keypunch. In Fig. 1.1 we show a typical keypunch; and in Fig. 1.2, the keypunch keyboard.

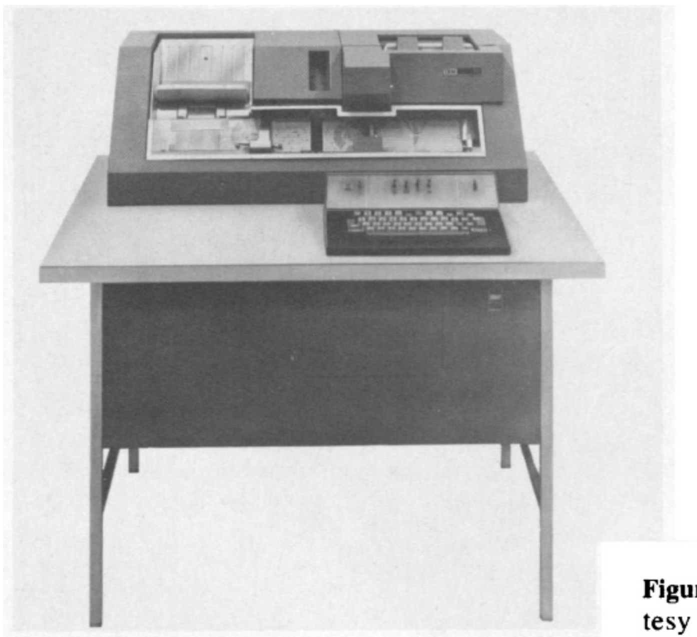


Figure 1.1. The IBM 029 keypunch. (Courtesy of IBM.)

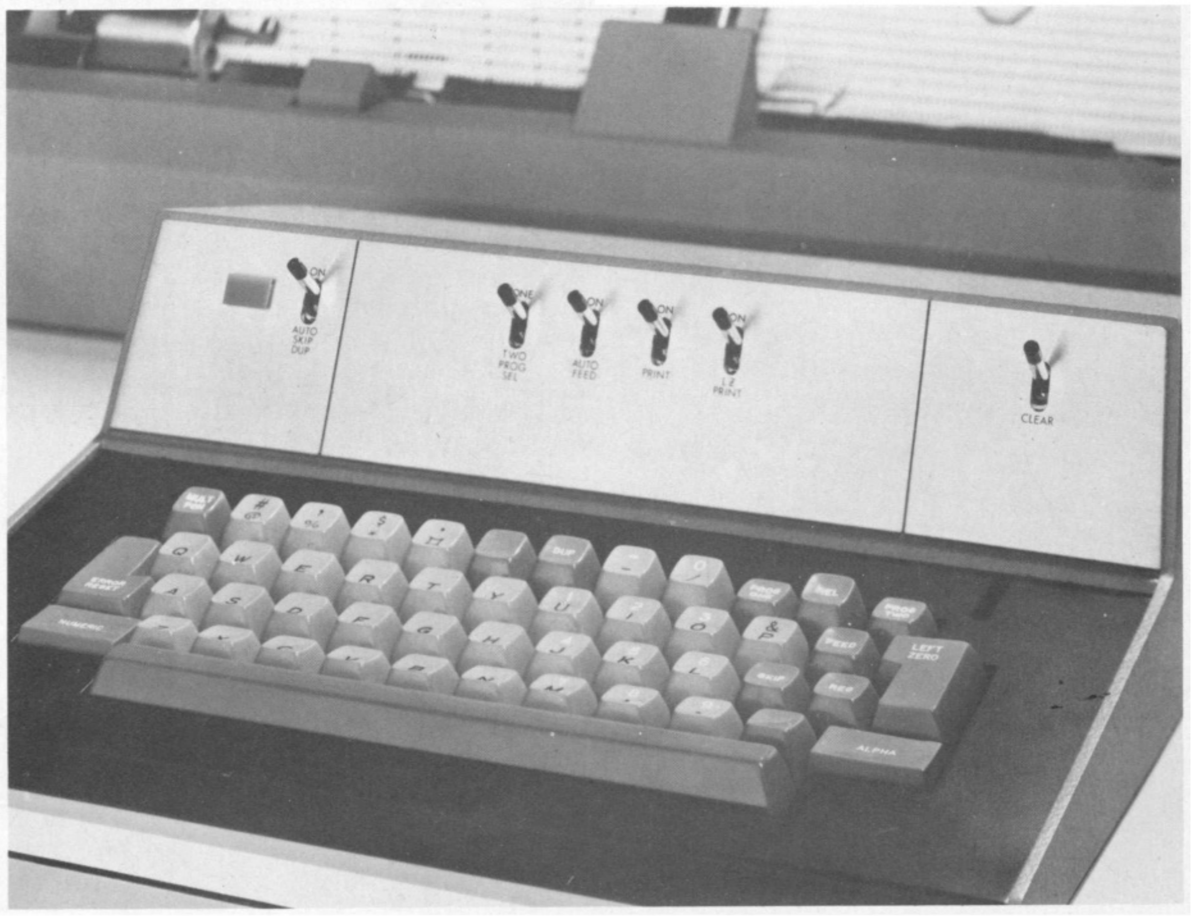


Figure 1.2. The IBM 029 keypunch keyboard. (Courtesy of IBM.)

In order to operate a keypunch, we must first place a batch of computer cards in the hopper—which is on the right of the keypunch. There is a clamp that will press the cards against the front of the hopper. When we press the FEED button, one card will be released from the hopper into the window directly beneath the hopper. If we press the FEED buttton again, a second card will be released from the hopper, and the first card will be properly positioned in the window so that we can start punching the card. An alternative approach for readying a card so that we can punch it is to press the FEED button and then the REG button.

After this preliminary procedure has been carried out, each time we press a key that has a symbol on it not only will that symbol be printed on the top of the card, but, more important, one or more rectangular holes corresponding to the symbol will be punched in the same column in which the symbol is printed. The printing on the card is of no consequence to the computer; only the holes are important. As we punch symbols on the card, the card will be advanced farther and farther into the middle window. In Fig. 1.3 we show some symbols that can be punched on a card and the holes that correspond to these symbols. As we see, a card is subdivided into 80 columns.

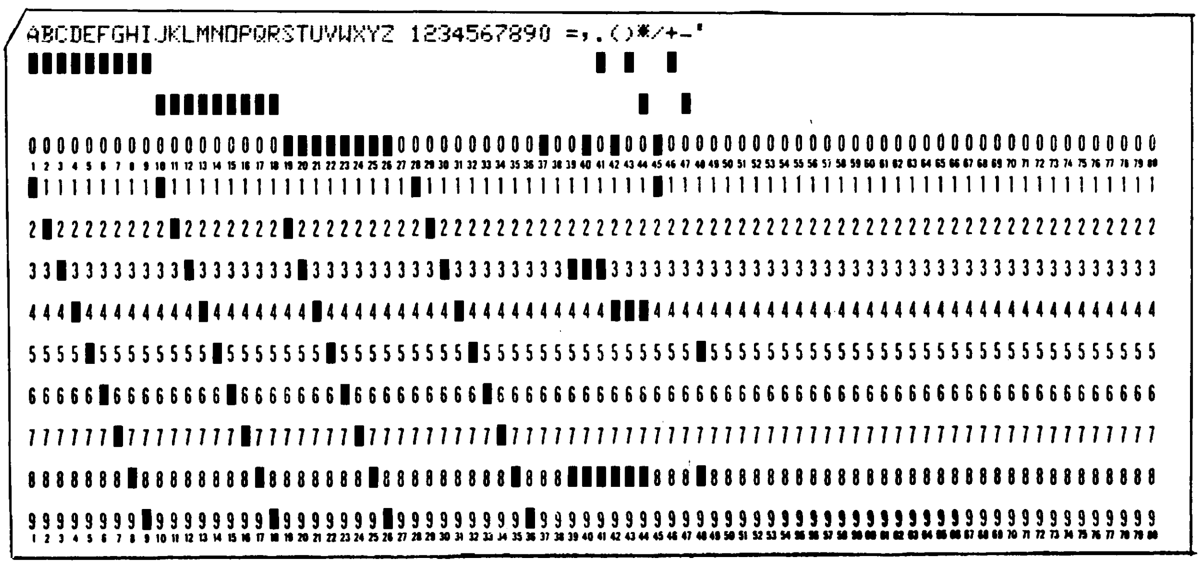
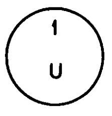


Figure 1.3. The characters punched on a card, and the holes that correspond to these characters.

The keypunch keyboard differs somewhat from a typewriter keyboard. For instance, although you can type lower-case letters—for example, a—on a typewriter, you cannot type them on a keypunch; when you depress the keypunch key marked A, a capital letter A is punched on the card.

Some keys have two symbols on them, for instance,



In order to punch the upper symbol on such a key, you must simultaneously depress the key and the NUM key.

After you have finished punching all the information you want on the card, you press the REL key. This transfers the card from the middle window to the left window. Then, if the card is not automatically ejected into the hopper above this window, press REG.

Above the middle window is a drum that rotates as the card is punched. By placing a specially punched card—called a control card—on this drum and then pressing the lever below the drum to the left, you can, for instance, instruct the keypunch to automatically make the card that you are punching skip to certain columns. If the control card is completely blank, then when you press a key that has two symbols on it, the upper symbol will be punched on the card. The keypunch is then said to be in *numeric mode*.

We now describe the function of the other keys.

ALPHA: If the keypunch is in numeric mode, then, when a given key and the ALPHA key are depressed simultaneously, the lower symbol on the key will be punched.

DUP: This allows you to duplicate the punching from the card in the middle window onto the card in the right-hand window.

BACKSPACE: This moves the card one column to the right, allowing you, for instance, to punch a symbol in a column that you accidentally skipped. The BACKSPACE key is below the middle window.

MULT PCH: This key allows you to punch more than one character in a column. If you simultaneously depress this key and a second key, the card will not advance to the left, and the symbol on the second key will be punched on the card in the same column in which you punched the last symbol.

SKIP: If you are using an appropriate control card, then when this key is depressed the keypunch advances the card to a predetermined column.

We shall explain the function of symbols that are neither numeric nor alphabetic as we encounter them in the book. In the rest of the book, we shall refer to the symbols on the keyboard as characters. There are six levers (or toggle switches) above the keyboard. Under normal conditions, the five leftmost levers should be in the "ON" position. The rightmost lever is used to clear the cards from all the windows and place them in the left hopper.

1.3. Input and Output Devices

Once all the instructions constituting a program are punched on cards, we can submit the deck of cards so that they can be processed by the computer. We take the deck of cards consisting of the program and the data and submit it to the computer center where it is read into a *card reader*. In Fig. 1.4 we show a card reader. As the card reader reads each card, it examines the holes in each column and thus determines which characters we have punched on each card. It transmits this information to the central processing unit, where the compiler determines what instruction we have punched on the card.

Once the program is compiled, unless we do not wish it, the instructions in the program will be printed. If we include the proper instructions, the results of the program will also be printed. The printing is done on the output device called a *line printer*. In Fig. 1.5 we show a line printer.

One device that can be used as both an input and output device is called a magnetic tape drive; it is shown in Fig. 1.6. It is very much like a tape recorder, but instead of recording voice patterns, it records arithmetic data in binary on magnetic tape. We can write information on a magnetic tape and then at some later time read the information as well.

The last device we discuss is called a disk. It is a metal disk on which we store arithmetic information in binary on concentric rings on the disk. We can both read information from a disk and write information on the disk. It is thus another example of devices that can be used both as input and as output devices. A disk unit is shown in Fig. 1.7.

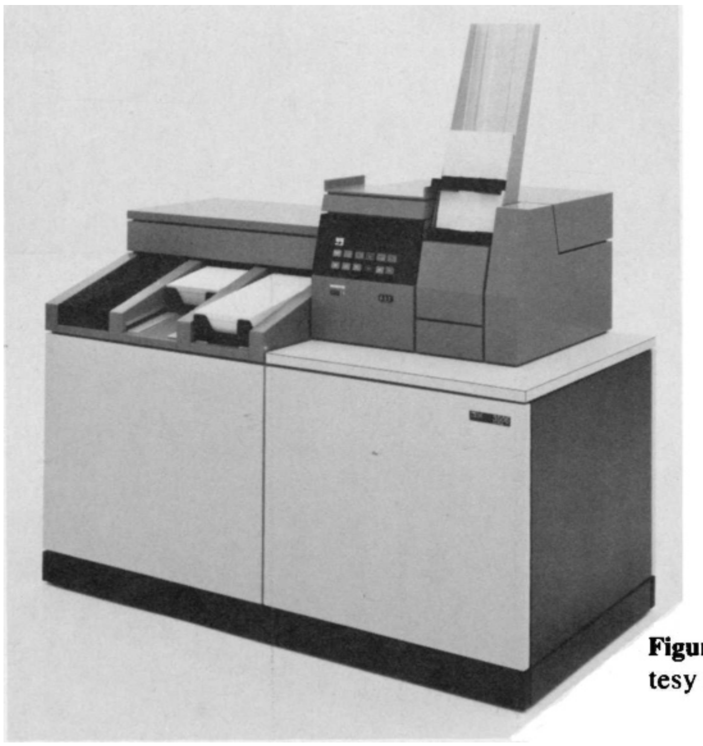


Figure 1.4. The IBM 3505 card reader. (Courtesy of IBM.)

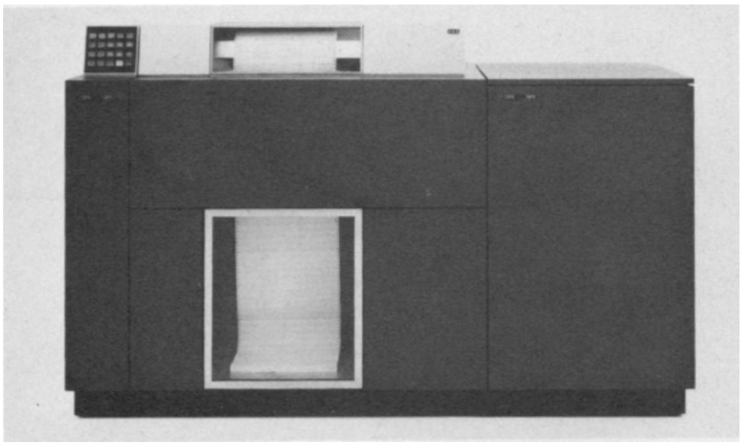


Figure 1.5. The IBM 3211 line printer. (Courtesy of IBM.)

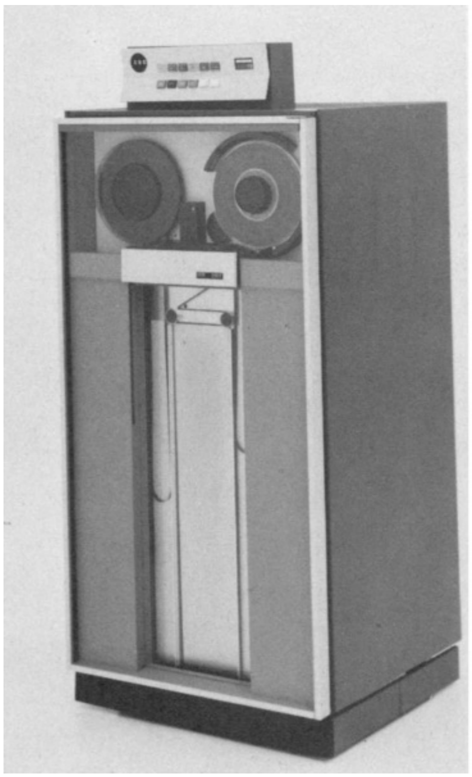


Figure 1.6. The IBM 3420 magnetic tape unit. (Courtesy of IBM.)

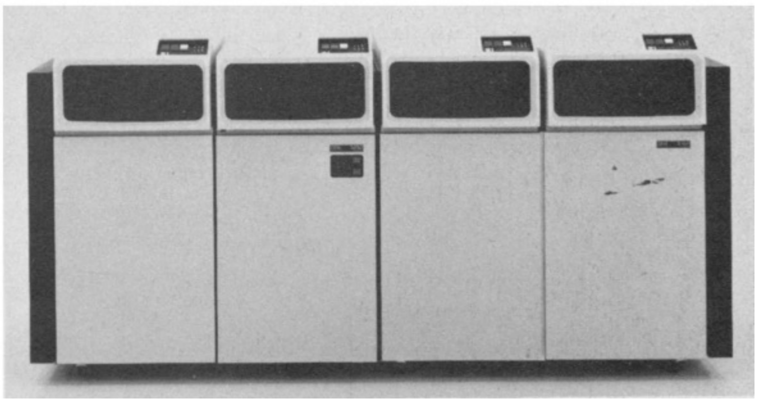


Figure 1.7. The IBM 3350 disk storage unit. (Courtesy of IBM.)

1.4. Solving a Problem

We now describe the solution of a problem that will give us a further insight into computers and programming. The problem is to determine the regional distribution of people standing in line in a certain city. That is, we wish to determine how many people come from the north, south, east, and west sides of the city.

We begin by drawing on a piece of paper four boxes, which we label N, S, E, and W, as shown in Fig. 1.8a. These letters represent north, south, east, and west, respectively. We shall place in the appropriate box a number indicating the number of people we have encountered from the north, south, east, and west sides of the city, as we question the people in line.

The composition of the line is shown in Fig. 1.8a. The fact that the first letter on the line is a W means that the first person comes from the west; the same type of correspondence is applied to the rest of the line. The arrow indicates which person we are questioning. Before reading the next paragraph, please study all of Fig. 1.8.

We first place a zero in each of the four boxes, as shown in Fig. 1.8a. This indicates that we have not as yet encountered anyone from any section of the city. We ask the first person in line which part of the city he is from. We are told "the west." We record this by adding 1 to the 0 that is in the box marked W, obtaining 1. In order to record this result, we erase the zero that was originally in this box and replace it by the digit 1. The box for the west now contains 1, and the other three boxes still contain 0, as shown in Fig. 1.8b. We ask the second person where he is from; he answers, "the north." We thus add 1 to the 0 that is in the box marked N, obtaining 1. To record this, we erase the 0 in this box and replace it by a 1, as shown in Fig. 1.8c. We see here that the rest of the boxes still contain the numbers they contained in Fig. 1.8b. We ask the third person where he is from; he answers "the west." We now add 1 to the 1 already in the box marked W, obtaining 2. We then erase 1, the previous entry in the box, and replace it with 2, as shown in Fig. 1.8d. We continue to the end of the line, applying this same procedure to each person we question. After the last person has told us where he is from and the information has been recorded, the boxes contain the numbers shown in Fig. 1.8e.

If this problem is to be programmed, the process we just described is called *running* or *executing* the program. When the program is run on a computer, the function of the boxes—that is, to contain or "store" the numbers—is played by the computer's memory locations. Like the boxes, a memory location can store only one number at a time; moreover, when the computer places a number in a memory location, it automatically erases the location's previous contents—that is, the number that was previously stored there. The placing of zeros in all the boxes at the beginning of the problem and the addition of 1's to the correct boxes is done by the computer's logical unit. The people standing in line are called the *data* or *input* to the program. We type on the keypunch both the program that solves the problem and the data for the program. After the program is done—if we so instruct—the computer prints the results of the program on the line printer. The control unit supervises the activities of all the units (except, of course, the keypunch) as they process this problem.

Indeed, all the power of the computer is not exhibited in our example. For instance, the computer can multiply many numbers together and get the results almost immediately. It can do a multitude of complex calculations that man, without the aid of a computer, would hesitate to undertake. In the following chapters we shall give many examples of what computers can do.

SOLVING A PROBLEM

Problem: Determining the regional distribution of people standing in line in a certain city. The letter N denotes north, S denotes south, E denotes east, W denotes west. The arrow indicates which person is being questioned.

$$\begin{array}{ccccccc} {\downarrow}W & N & W & S & S & E & W \\ \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & & & \\ N & S & E & W & & & \end{array}$$

Figure 1.8a. Composition of the line and the contents of the boxes before questioning begins. The arrow is at left of line, indicating that questioning has not begun.

$$\begin{array}{ccccccc} \overset{\downarrow}{W} & N & W & S & S & E & W \\ \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} & & & \\ N & S & E & W & & & \end{array}$$

Figure 1.8b. We add 1 to the box for west after questioning the first person.

$$\begin{array}{ccccccc} W & \overset{\downarrow}{N} & W & S & S & E & W \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & & & \\ N & S & E & W & & & \end{array}$$

Figure 1.8c. We add 1 to the box for north after questioning the second person.

$$\begin{array}{ccccccc} W & N & \overset{\downarrow}{W} & S & S & E & W \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{2} & & & \\ N & S & E & W & & & \end{array}$$

Figure 1.8d. We add 1 to the box for west after questioning the third person. It now contains 2.

$$\begin{array}{ccccccc} W & N & W & S & S & E & W{\downarrow} \\ \boxed{1} & \boxed{2} & \boxed{1} & \boxed{3} & & & \\ N & S & E & W & & & \end{array}$$

Figure 1.8e. The contents of all boxes after we have questioned each person in the line.

1.5. Algorithms

If we wanted to instruct someone how to conduct the survey discussed in Section 1.4, we would have to write a set of instructions for him to follow. In mathematics, a set of instructions written to solve a problem is called an algorithm. Assuming that the boxes have been already drawn, the algorithm for performing the survey is:

1. Place a zero in each box.
2. Ask a person which part of the city he is from.
3. Add 1 to the proper box.
4. If the person is the last one in line, stop the survey; otherwise repeat instruction 2.

Instruction 3 always follows instruction 2; and instruction 4 always follows instruction 3.

The sequence of how the instructions are performed for the first three people in line and the results of performing these instructions are as follows:

Instruction	Result
1	$\boxed{0}\boxed{0}\boxed{0}\boxed{0}$
	N E S W
2	We are told W
3	$\boxed{0}\boxed{0}\boxed{0}\boxed{1}$
	N E S W
4	Not the last person, so perform instruction 2
2	We are told N
3	$\boxed{1}\boxed{0}\boxed{0}\boxed{1}$
	N E S W
4	Not the last person, so perform instruction 2
2	We are told W
3	$\boxed{1}\boxed{0}\boxed{0}\boxed{2}$
	N E S W
4	Not the last person, so perform instruction 2

This process is continued up to and including the last person in line. Since he is from the west, a 1 is added to the box marked "W". The algorithm then terminates as directed in instruction 4.

A computer program is simply an algorithm written in a language the compiler can understand. If the algorithm on which the program is based is incorrect, the program will produce incorrect results.

1.6. FORTRAN, WATFOR, and WATFIV

The FORTRAN language has gone through several stages. The current stage, called FORTRAN IV, incorporates practically all the features of the preceding stages. Most of the computer manufacturers have their own version of the FORTRAN IV compiler in use on their computers; in fact, it is possible to have more than one version of the FORTRAN compiler used on a given computer. All these versions are similar except perhaps for some auxiliary features.

What most compilers have in common are the programming instructions incorporated in what is called ANS (American National Standards) Standard FORTRAN. If you wish to write a program that can be used on most compilers, you should write your instructions so that they conform to ANS Standard FORTRAN. When a program can be used by many compilers, we say that it is transportable. We have devoted almost this entire book to a discussion of ANS Standard FORTRAN.

Another compiler, ANS Standard Basic FORTRAN, was written with the smaller computers in mind. Many of the features of Standard FORTRAN are not included in Standard Basic FORTRAN. We note this in footnotes to the text.

A compiler developed in Canada at the University of Waterloo simplifies some of the features of FORTRAN that beginners find difficult—notably the instructions that accept data from the card reader and prepare results for the line printer. This compiler is called WATFOR. The name is taken from WATerloo FORtran. A more flexible version of WATFOR is called WATFIV (WATerloo Fortran IV). We describe both WATFOR and WATFIV.

All the instructions in ANS Standard FORTRAN were incorporated into WATFOR. And all the instructions in WATFOR were incorporated into WATFIV.

A new compiler called WATFIV-S incorporates the features of WATFIV; moreover, what is important is that it enables you to write your programs in a more elegant way using what is called structured programming. We describe structured programming features of WATFIV-S in appropriate places in this book. As of January 1977 one of these features* has been incorporated into the proposed ANS FORTRAN revision.

The reader might wish to take more than a casual interest in the parts of the book that discuss WATFOR/WATFIV and the other non-Standard features because they have been incorporated into the proposed ANS FORTRAN revision.

* This feature is the IF-THEN-ELSE statement.

2

Introduction to FORTRAN

2.1. General Remarks

A FORTRAN program is a series of statements that are instructions to the computer. Each statement is keypunched on a separate card.

In Fig. 2.1a, we see a simple FORTRAN program written on a sheet of paper called a coding form. This sheet will indicate to the person who keypunches the program, in which columns to keypunch the different characters that comprise the statements in the program. The first line of this particular program is a comment. If a C is keypunched in column 1 of a program card, the computer will not process the card any further, but will simply print what is on that card when it prints the statements in the program. Comment cards are important in that they may aid someone reading the program to understand the program; however, they are completely overlooked by the compiler. Thus a workable FORTRAN program can be written without comment cards.

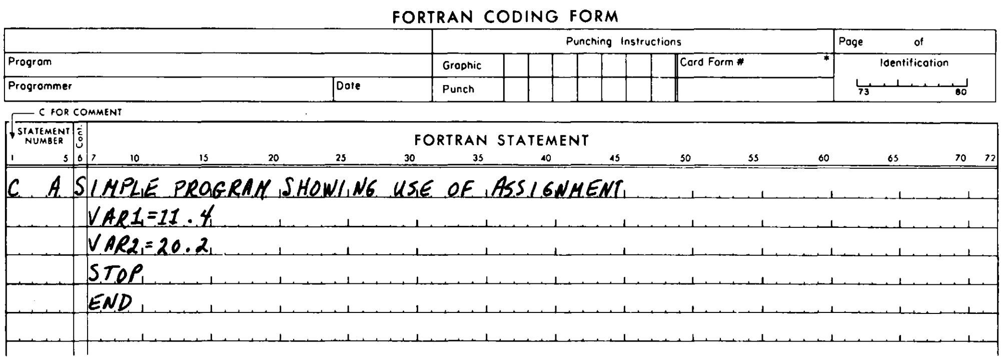
The rest of the lines represent FORTRAN statements that *are* processed. These are, of course, the statements that are of primary importance in a program. (They will be discussed in this chapter and other chapters of the book.) These statements may be keypunched anywhere between columns 7 and 72 on the card; however, under normal circumstances we begin simple statements in column 7 as shown in Fig. 2.1b. Figure 2.1b indicates how the program of Fig. 2.1a would appear when keypunched on cards.

2.2. The Assignment Statement

In order to store a number in a location in the computer's memory, we give these locations names, which are called *variables,* and assign numbers to them in the program. In the program of Fig. 2.1a, the assignment of the numbers to variables is made by a statement that includes an equals sign. It is called an *Assignment* statement. In the assignment statements in this program, VAR1 and VAR2 are the variables. In the first line of the program, we assign the number 11.4 to VAR1 by keypunching

```
VAR1=11.4
```

WRITING A SIMPLE PROGRAM ON A CODING FORM


FORTRAN CODING FORM

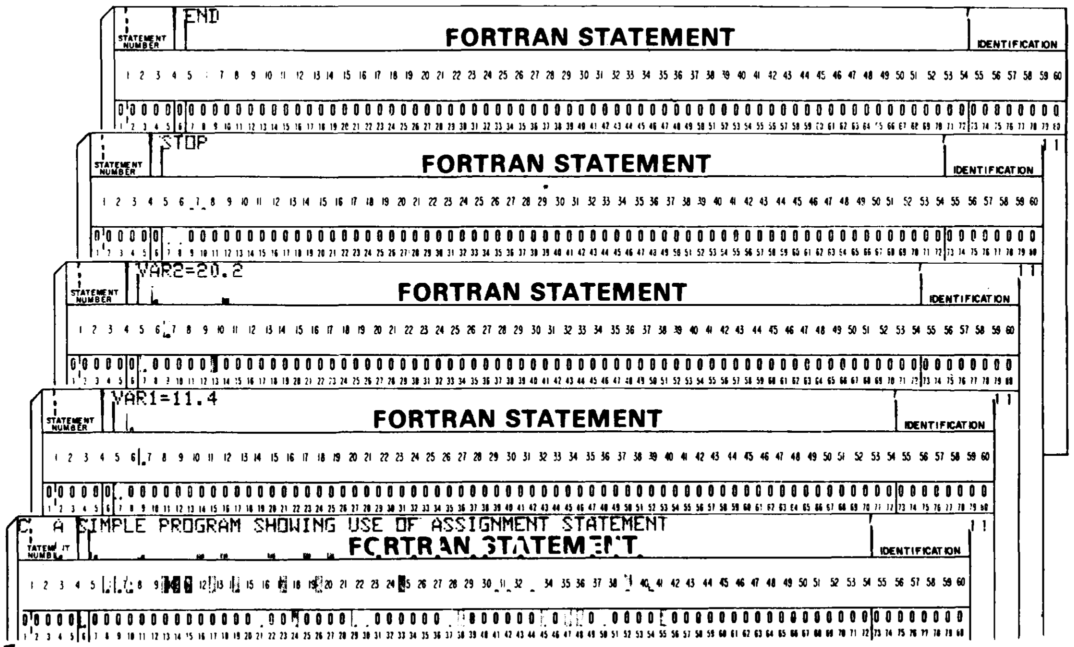
	Punching Instructions		Page of
Program	Graphic	Card Form #	Identification
Programmer / Date	Punch		73 80

C FOR COMMENT — STATEMENT NUMBER — FORTRAN STATEMENT

```
C A SIMPLE PROGRAM SHOWING USE OF ASSIGNMENT
      VAR1=11.4
      VAR2=20.2
      STOP
      END
```

Figure 2.1a. A simple FORTRAN program written on a coding form. A program consists of statements. When a C appears in column 1, that statement is not processed by the computer. It is just used by the programmer to write comments about the program. Hence it is called a comment card.

KEYPUNCHING A SIMPLE PROGRAM


```
      END
      STOP
      VAR2=20.2
      VAR1=11.4
C A SIMPLE PROGRAM SHOWING USE OF ASSIGNMENT STATEMENT
```

STATEMENT NUMBER
FORTRAN STATEMENT
IDENTIFICATION

These are the column numbers

Figure 2.1b. The program that first appeared on the coding form of Fig. 2.1a, now keypunched on cards. All statements but the comment card are punched anywhere between column 7 and 72 inclusive, on the program card. The END statement instructs the compiler that the program is finished.

Thus we have instructed the computer to place the number 11.4 in the memory location named VAR1. In a similar way, we next assign the number 20.2 to VAR2 by typing

```
VAR2=20.2
```

Another way of describing what has happened in these two statements is to say that the VAR1 has the value 11.4 and VAR2 the value 20.2.

We instruct the computer to compile and then execute (run) the program by typing certain instructions on cards that are not part of the program. These cards are called *control cards*, and in general they are used to instruct the compiler how to process our program. Since different compilers require different types of control cards, we will, throughout the text, simply describe a given program and then the results of executing it. In the Appendix however, we describe how to write control cards for two systems: FORTRAN IV on the IBM system 360/370; and WATFIV on the IBM system 360/370.

2.3. The END Statement

Sometime after we have submitted our deck of cards to the computer, the computer operator will run our program. The compiler first checks each statement in our program for grammatical errors until it reaches the END statement. The END statement indicates to the compiler that we have finished writing the program, i.e., that no more cards will follow. The END statement must appear as the last statement in any FORTRAN program.

2.4. The Listing of the Program; the STOP Statement

As the compiler checks each statement for errors it prints or *lists* the statement on the line printer. In Fig. 2.2a we see a listing of the program. The program is listed exactly as we keypunched it on the program cards. Since the first card in the program is a comment card, the compiler (after it detects the C in column 1 of the card) simply prints the contents of the card and does no checking; however, the computer does check the other statements in the program of Fig. 2.2a. If a statement has an error in it, the compiler will indicate that it has found an error by printing an error message. Since none of the statements contain errors, the computer prints no error messages.

When the computer performs the task that a program statement has instructed it to perform, the computer is said to *execute* the statement. We will soon see how the computer executes the statements in the program.

After the compiler has checked the FORTRAN program for errors and has found none, it translates the program ultimately into machine language (each FORTRAN statement is translated typically into many machine language instructions). The computer will then execute the translated program, beginning with the machine language equivalent of the first noncomment card in our program and then executing the machine language equivalent of each successive card until it reaches the STOP statement. The computer follows this order unless we specifically instruct it in the program to follow a different order; we will describe how to change this order later in the book. The STOP statement instructs the computer to terminate the program.

USING THE ASSIGNMENT STATEMENT

```
C   A SIMPLE PROGRAM SHOWING USE OF ASSIGNMENT STATEMENT
      VAR1=11.4
      VAR2=20.2
      STOP
      END
```

Figure 2.2a. In the program the number 11.4 is assigned to the variable VAR1 and 20.2 to VAR2 in the assignment statements.

On most compilers, if you do not include a STOP statement in the program, the compiler will insert one directly before the END statement when it translates the program.

2.5. Executing the Program

We now follow the execution of the program in Fig. 2.2a with the help of the Table for Fig. 2.2a. This table shows the values of the variables as each card of the program is executed—the table is meant to be used as a learning aid; it is not part of the program.

Card	Description	VAR1	VAR2
1st	Comment	NOT EXECUTED	
2nd	VAR1=11.4	11.4	Undef
3rd	VAR2=20.2	11.4	20.2

We see from the table that the first card—the comment card—is not executed. When the second card is executed, the value associated with VAR1 in the computer's memory is 11.4; however, VAR2 has not yet been assigned a number. In such a case, we say that VAR2 is *undefined*. This is reflected in the table by the entry Undef. When a variable (such as VAR1 here) has been assigned a number, we say that it has been defined. When the third card is executed, we see from the third line of the table that the value associated with VAR2 is 20.2.

Once a variable is assigned a given number, it retains that assignment until the end of the program unless we assign it a new number. Thus, we see from the third line of the table for Fig. 2.2a that when the third card is executed, the computer remembers that 11.4 was assigned to VAR1.

When the computer encounters the STOP statement, it finishes execution. It usually signifies this by printing some message on the line printer, such as EXECUTION COMPLETE, or some computer memory bookkeeping about the run of the program.

The results of the execution are shown in Fig. 2.2b. We note to our disappointment that the computer has printed nothing in Fig. 2.2b except some computer memory bookkeeping information indicating that the computer has finished execution.*

```
CORE USAGE          OBJECT CODE=        192 BYTES,
```

Nothing else was printed because the computer executed the two assignment statements, encountered the STOP statement, and stopped. Since we did not include a statement in the program instructing the computer to communicate the assignments to us, it did not do so. In the next section, we remedy this situation.

* This is only the beginning of the bookkeeping information as printed by the WATFIV compiler.

EXECUTING THE PROGRAM

```
C  A SIMPLE PROGRAM SHOWING USE OF ASSIGNMENT STATEMENT
      VAR1=11.4
      VAR2=20.2
      STOP
      END
```

Figure 2.2a. This figure is reproduced for the reader's convenience.

Card	Description	VAR1	VAR2
1st	Comment	NOT EXECUTED	
2nd	VAR1 = 11.4	11.4	Undef
3rd	VAR2 = 20.2	11.4	20.2

Table for Fig. 2.2a. The status of the variable in Fig. 2.2a given by card number during execution. A comment card is never executed. In the first assignment statement, the value of VAR2 has not yet been defined. When the second assignment statement is executed, the computer remembers that 11.4 was assigned to VAR1.

```
CORE USAGE          OBJECT CODE=        192 BYTES,
```

Figure 2.2b. The results of running the program of Fig. 2.2a. Since we did not include in the program, a statement instructing the computer to print the results, it did not. It simply prints some bookkeeping information indicating that it has finished executing our program. The STOP statement instructs the computer to terminate the program.

2.6. The WRITE and FORMAT Statements

We now rewrite the program of Fig. 2.2a and add a statement instructing the computer to print the numbers that have been assigned to VAR1 and VAR2. To do this on a FORTRAN system or a WATFOR/WATFIV system, we add to the program the statement

```
      WRITE(6,10) VAR1,VAR2
```

as shown in Fig. 2.3a. The first number—here 6—in the WRITE statement refers to the output unit we want information printed on; the 6 is the number most commonly designated at computer centers to represent the line printer. The 10 refers to the statement that describes the form (or format) in which we want our results to be printed. Appropriately enough, this type of statement is called a FORMAT statement. In order to make it possible for a statement (here a FORMAT statement) to be referred to by another statement (here the WRITE statement), the "referred to" statement must have a statement number—also called a label. We have chosen 10 to be the FORMAT's statement number:

```
 10   FORMAT(1X, F10.3, F9.4)
```

However, we could have chosen any integer between 1 and 99999. The statement number must be punched anywhere in the first 5 columns of the program card, as we have shown in Fig. 2.3b for the FORMAT statement. We punch these numbers, however, starting in column 2 in order to improve readability; we have not printed them starting in column 1, since we want the person reading the program to be able to immediately distinguish between statement numbers and the C in the comment statements.

The terms appearing between the parentheses in the FORMAT are called *field specifications*. Here, 1X is the first field specification; F10.3, the second; and F9.4, the third. When 1X appears at the extreme left of the FORMAT, as it does here, it instructs the line printer to vertically advance the page one line before the line printer starts printing. The 1X is said to be used for "carriage control." The F10.3 and F9.4 each describe a group of columns on a line to be printed on the line printer. These groups of columns are called *fields*. F10.3 describes the first field appearing on the left of the line printed; F9.4 describes the second field from the left. Each of the two field specifications beginning with the letter F refer to one of the variables in the associated WRITE statement:

```
      WRITE(6,10) VAR1,VAR2
 10   FORMAT(1X, F10.3, F9.4)
```

Since F10.3 immediately follows 1X, it refers to the first variable in the WRITE statement, i.e., VAR1. Since F9.4 is the second field specification to follow the 1X, it refers to VAR2, the second variable in the WRITE statement. The F in both of these field specifications indicates that the numbers that will be printed each have a decimal point in them; these