FORTRAN 77
Principles of PROGRAMMING

Jerrold L. Wagener

FORTRAN 77
Principles of PROGRAMMING

Jerrold L Wagener

Computer Science
State University of New York, Brockport

John Wiley & Sons

New York Chichester Brisbane Toronto

Copyright © 1980, by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of
this work beyond that permitted by Sections
107 and 108 of the 1976 United States Copyright
Act without the permission of the copyright
owner is unlawful. Requests for permission
or further information should be addressed to
the Permissions Department, John Wiley & Sons.

Library of Congress Cataloging in Publication Data:

Wagener, Jerrold L
Fortran 77 Principles Of
Programming
Includes indexes.
1. FORTRAN (Computer program language) I. Title.
QA76.73.F25W32 001.6'424 79-17421
ISBN 0-471-04474-1

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

PREFACE

This book is a comprehensive text/reference on Fortran programming, and completely describes the new standard Fortran language—Fortran 77. In general it presents, in a Fortran 77 context, the important principles of contemporary computer programming practice. In particular it presents structured programming by application as the normal way to program.

Fortran 77 has many features, not contained in the previous Fortran standard, that contribute significantly to its suitability as a modern general-purpose computer language. Among these features are the CHARACTER data type (which provides a good fixed-length string facility), the block-IF control structure (which provides if-then-else and general n-way selection control), and extensive I/O facilities (which include format-free READ and PRINT statements, file connection control, and provisions for I/O error recovery). There are a number of others, and I have tried to show that the resulting Fortran 77 is indeed an excellent general-purpose language for modern software development.

I have treated Fortran 77 as an entirely new and versatile language, with the presentation unencumbered by references to and comparisons with previous versions of Fortran. Experienced Fortran programmers are assured, however, that their old Fortran programs will function unchanged as Fortran 77 programs, almost without exception. Experienced and novice programmers alike will benefit from the emphasis on effective modern use of the features of Fortran 77. Concepts and techniques pertaining to well-structured programs, program modularization, numeric and nonnumeric processing, program correctness, data files, data types and structures, and recursion are discussed in detail.

An important feature of this book is the presentation of a large number of example computer programs, usually complete with output. These examples systematically progress from extremely simple ones in the first chapters to quite sophisticated ones later on, with each new example (in most cases) illustrating one additional feature of Fortran 77. The example programs are all written in the same consistent style, a style that I believe to be in the spirit of contemporary practice and highly effective in impressing on the reader the virtues of well-structured, readable, well-documented programs. One benefit of this, I hope, is that the example programs are so complete and lucid that they are, by themselves, sufficient to initially acquaint the reader with the essence of Fortran 77. They are easily distinguished from the rest of the text by the shading along the left margin.

The book is divided into three major parts. Part 1 (''Fortran Fundamentals'') contains six chapters of material basic to Fortran programming. It includes simple I/O, the declaration and use of data elements in problem solving, selection and repetition control structures, and ends with an introduction to program modularization (subroutines). Preceding Part 1 is an introductory chapter (Chapter 0) supplying background material on the structure of computing machinery and the nature of computer programming. After Part 1 the chapters are largely independent and self-contained, and their order is not especially critical.

Part 2 (''Program Structure'') deals with the structuring of programs and related aspects of programming and program development. This part begins with two chapters on procedures (subroutines and functions), examining—in detail—argument passing and association, local and global data element facilities and their uses, and guidelines pertaining to procedure design and use. A chapter is then devoted to an in-depth examination of control structures and corresponding Fortran 77 facilities, with particular attention paid to loop exits. One application of these concepts occurs in Chapter 10, which deals with program correctness and proofs of correctness. This chapter contains an introduction to proving programs correct, and, to my knowledge, contains the first reasonably general treatment of proving Fortran programs correct. Chapter 11, on recursive procedures, completes Part 2, and discusses another important programming technique largely ignored (in the literature, but increasingly implemented) in the context of Fortran.

Part 3 (''Data Structure'') deals with data elements and Fortran's facilities for

representing and processing data. The intrinsic data types of DOUBLEPRECISION and COMPLEX are introduced, and illustrated with program examples using each. Techniques are developed for utilizing either INTEGER or CHARACTER data types for simulating bit strings, enumerated data types, lists, trees, and records, and program examples illustrate the use of each of these kinds of data elements. The major focus in this part, however, is on data files and the Fortran 77 provisions for opening, closing, reading, writing, and inquiring about sequential, direct, and internal data files. Appendices B and C illustrate typical ways in which Fortran's data elements are implemented on computing machinery.

Several computer courses could use this book as a text. No prerequisites are necessary, and each course should include all six chapters of Part 1. A minimal introductory course in Fortran programming could reasonably terminate at this point. A very ambitious course in Fortran programming or introductory computer science could include most of the material in the book, as could a more leisurely two-course sequence. Most courses in Fortran programming will include, in addition to Part 1, Chapters 7 and 8 ("Subroutines" and "Functions") from Part 2 and then an appropriate selection from the last seven chapters (9–15). For example, a course emphasizing business applications of computing would also include Chapters 14 and 15 ("Data Files" and "File I/O"); a course emphasizing scientific/engineering applications would include Chapters 12 and 13 (sophisticated use of data elements). The end-of-chapter programming exercises constitute an unusually large and diversified set of problems. An instructor's manual, containing solutions to many of these exercises, is available from Wiley.

Programming-oriented courses in computer science fundamentals, for which this book would be an excellent text or language supplement, normally include all of those topics in Part 2 and some of those in Part 3. In general the choice of which chapters to include from the last seven should be governed by the interests and objectives of the class and the time available. Whichever chapters are chosen will be useful for future reference, since the programming practitioner will most assuredly eventually encounter these areas. Experienced programmers will need only to quickly review Part 1 (perhaps glancing mostly at the example programs and Appendix A), and then concentrate on those chapters of interest in Parts 2 and 3.

The example programs have all been processed with a commercially available Fortran 77 compiler, and I thank the Prime Computer Co., Framingham, MA, for making an early version of their Fortran 77 compiler available to me. Most commercial compilers contain useful extensions to the standard language. I have carefully avoided any such extensions so that the material in this book will apply directly to any implementation of Fortran 77 that conforms to the standard language. The official description of the Fortran 77 standard may be obtained from American National Standards Institute, Inc., 1430 Broadway, New York, NY, 10018, as document ANSI X3.9-1978 Programming Language Fortran.

As previously indicated, I believe that Fortran 77 is an excellent general-purpose language. For the most part I have endured its deficiencies, often developing alternative techniques that can be used to advantage by the programmer. Fortran 77 has one deficiency, however, although it is easily remedied: it does not have a general loop control structure with exiting. Therefore toward the end of Chapter 5 ("Loop Control") I define a simple do-repeat-exit loop control structure with a single-level exit facility. Thenceforth I use do-repeat-exit when describing loops. When this nonstandard feature appears in a program it is clearly identified as such, and the conversion to standard Fortran 77 is simple. In fact subroutine DOREPX in Chapter 8 (and program F PLUS in Chapter 15) performs this conversion automatically (and has been tested on all of the program examples containing do-repeat-exit). This program (which works for do-repeat-exit in either lower- or upper-case letters) may be used in lieu of the given manual conversion technique.

I am indebted to a number of individuals who reviewed the original manuscript and made suggestions for improvement. I especially thank Anthony Ralston for his

此为试读,需要完整PDF请访问: www.ertongbook.com

many valuable suggestions. Others deserving special thanks for their suggestions are Harry Gross, Charles Hughes, Larry Humm, Charles Pfleeger, and Jean Wagener. These individuals, and others, deserve much credit for spotting errors and recommending an occasional different tack. Any remaining deficiencies in the finished product are, of course, solely my responsibility.

Jerrold L. Wagener

Brockport, New York
September 1979

CONTENTS

Chapter 0 Programming Fundamentals

This book describes completely the elements of the Fortran computer programming language, and in considerable detail, illustrates effective, contemporary use of Fortran. While the treatment of Fortran is comprehensive, and many aspects of contemporary programming technology are described, no prior knowledge of programming is assumed. Because of the comprehensive nature of this book, and the resulting inclusion in places of somewhat specialized material, the presentation of Fortran is organized into three parts. Part 1 (Fortan Fundamentals) describes a minimal subset of Fortran that is useful for writing programs. All six chapters of Part 1 are fundamental, and these chapters contain only general material, important to all programmers. Parts 2 and 3 contain more specialized material, and the chapters in these two parts will be of varying relevance, depending on the interests and objectives of the reader. For this reason the chapters in Parts 2 and 3 have been designed so that their order is not critical and the interdependence among these chapters is minimal.

The reader who is already familiar with the fundamental concepts of computer programming, and wants to turn immediately to the description and use of Fortran, may proceed directly to Part 1. This chapter provides the background concerning computers and the nature of programming necessary to prepare the reader for the art and science of Fortran programming.

0.1 Computing Systems

The principal elements of a typical computing system are shown schematically in Figure 0.1. (Large computing systems are often considerably more complex than shown in Figure 0.1, but even in such systems the following fundamental concepts are still valid.)

These elements fall into three general categories:

1 Processing units (the CPU box).

2 Memory units (the elements labeled "Main memory" and "Secondary memory").

3 Input/output units (the "Line printer", "Card reader", and "Terminals").

In general, the CPU (Central Processing Unit) performs the actual computations and data processing, and controls the interaction of the various parts of the system. The memory units are those devices that "hold" the data that is to be processed. The arrows in Figure 0.1 represent possible paths over which data can move. Thus data can be moved between the CPU and main memory (MM). Input/output units provide the interface for the flow of information between the computer and the outside world (e.g., humans). In the following sections each of these three categories of computer system elements are described in greater detail.

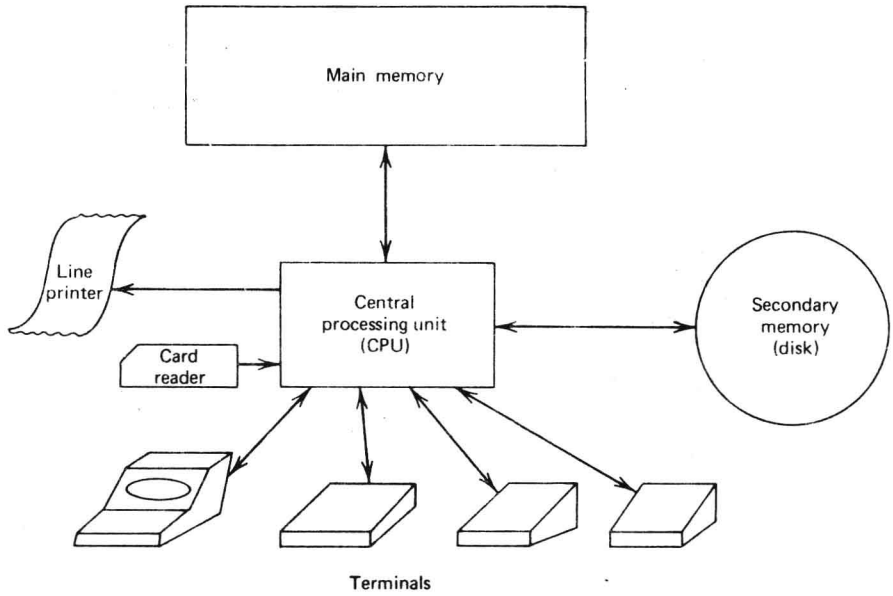

Figure 0.1. Elements of a computing system.

0.1.1 Processing Units

All of the elements of a computer system are important to its functioning, but the CPU is the "heart" of the system. It is an intricate electrical structure, composed of large numbers of electronic components, which provides the electrical control for the movement of data between the CPU and any other part of the computing system. Note from Figure 0.1 that all data flow goes through the CPU (which—at least partially—accounts for the word "Central" in the term Central Processing Unit).

In the computer all data is actually represented as series of pulses of electricity. The presence of a pulse can be thought of as a "1", and the absence of a pulse as a "0". Thus the data can be thought of as packets of 1's and 0's. All numeric information (e.g., integer numbers) is coded in 1's and 0's for the purposes of processing by the computer. The nature of such coding is shown in Appendix C, although the Fortran programmer normally need not be concerned with the details of this coding. Similarly, nonnumeric information (e.g., character symbols, English text, etc.) is coded in "bits" (as the 1's and 0's are called) and Appendix B shows the most common forms this coding takes. Thus the transmission of data performed by the CPU is the transmission of 1's and 0's in the form of electrical pulses. Such transmission can occur at the very high rates typical of electrical/electronic phenomena.

The other function of the CPU is the modification of data. Certain electronic parts of the CPU, called "registers," can accept the bits of data and the CPU can modify data in its registers. For example, suppose that the bits in two different registers represent two numeric values. The CPU is capable of adding these two values and placing the resulting pattern of bits in a register. This (newly obtained) data may then be transmitted to another part of the system (such as one of the Input/Output units). Figure 0.2 illustrates schematically two registers containing bits that represent integers (5 and 9) to be added and the sum placed in a third register. The CPU in most computing systems is capable of a large variety of data modification operations, so that virtually any desired processing of data can be readily accomplished. Such processing occurs at very high electrical/electronic speeds.

The CPU is the only place in the computing system where data can be modified. Thus the CPU really is central to the functioning of the computer—it's where the action is. It's where data is processed, and/or transferred to some other part of the system. The registers of the CPU are where the data are when it's being processed (modified) or in the process of being transferred. Although the number of registers, and size of each register, vary tremendously from computer to computer, typically a

此为试读,需要完整PDF请访问: www.ertongbook.com

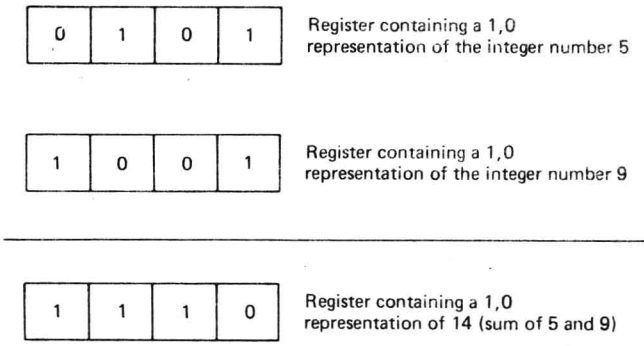

Figure 0.2. Concept of a register, containing the 1's and 0's of data.

CPU has perhaps 20 registers with 16 bits in each register (other typical register sizes are 8, 32, and 60).

0.1.2 Memory Units

The CPU is a beehive of activity and volatility. It cannot hold much data at any one time (because there aren't a great many registers), and what data it does have is probably in the process of being changed or moved (or both). A typical computer application involves the processing of a great deal of data. Provision must be made to store this mass of data, and make it available a little bit at a time to the CPU for processing. Memory units are used for this purpose. Memory units are places where large amounts of data can await their turn for processing in the CPU. No changes of data take place in memory. In order for a piece of data in memory to change, it must be transferred to the CPU, changed in the CPU, and then transferred back to the memory.

A memory unit can be thought of as merely a large collection of registers, with provision for transferring the contents of any register, or cell, to and from the CPU (see Figure 0.3).

Figure 0.1 shows two memory units, called Main Memory and Secondary Memory. The reason for this situation, which is typical in contemporary computing systems, is a limitation of present-day technology—high-speed memories are very expensive and low-speed memories are, in comparison, cheap. Thus a small amount (often less than 1,000,000 cells) of high-speed memory (Main Memory) is used for most of the CPU-Memory data transfers, and a large amount of low-speed memory (Secondary Memory) is used to hold data that are not currently being processed.

The individual cells in a Main Memory unit may be processed in a random fashion. That is, the cells may be accessed in any order. For example, immediately after main memory cell number 478,256 is accessed, cell number 93,401 may be accessed. In order to make such random access possible, each memory cell must have a unique identification, and this identification must be given when that cell is to be accessed. Such identification is called the cell "address", and in Figure 0.3 the cell addresses are given as 1, 2, 3, 4, . . ., n-1, n. The bits in the cell representing data, are called the cell "contents."

Thus each memory cell has two items associated with it—its address and its contents. When the CPU needs to access a certain memory cell it specifies the address of that cell; when the CPU transfers data to or from a cell it is the contents of the cell that are moved. The address of a cell always remains the same, and constitutes the unique identification of that cell. The contents of a cell may be any pattern of bits (which represent data), may change from time to time (under the control of the CPU), and has no relation to that cell's address. Memory cell addresses and contents are very important aspects of computer programming. Fortran's provisions concerning these aspects of programming are introduced in Chapter 2.

0.1.3 Input/ Output Units

A CPU and a memory unit constitutes a "complete" computing system, in the sense that if the original data is somehow placed in the memory, virtually any processing of

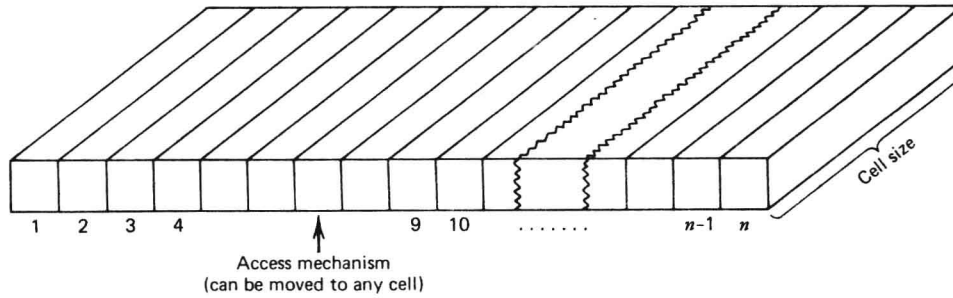

Figure 0.3. The structure of a memory unit. Each cell (register) contains a set of bits (e.g., 8, 16, 32, 60) representing bit-coded data. The value of n for main memory units can vary considerably, from computer to computer, from about 10,000 for very small systems to about 10,000,000 for large ones. Secondary memory units are much longer (have many times more cells). The access mechanism is capable of rapidly accessing individual cells, one after the other, usually in any order, for the purpose of transferring data between the cells and the CPU.

that data may be performed. In order to be practical, however, results of the processing must be communicated to the outside world (i.e., outside of the CPU-memory combination). Moreover, practical means must be available for placing the initial data in the memory. Thus communication links are needed between humans and the CPU-memory combination. These links are provided by the input/output units.

A typical computing system has (or may have provisions for) a number of input/output units by which humans can communicate with the system. Three different such units are shown in Figure 0.1. The line printer is a high-speed printing device that can be used to display the data contents of (any group of) memory cells. The card reader is a high-speed reading device that can be used to input data into memory cells (the data is first placed, by humans, on punched cards, and the punched cards are then placed in the card reader). An increasingly used input/output unit is the *terminal*, four of which are shown in Figure 0.1. This device has a keyboard, similar to that of a typewriter, for inputting data into the computer, and either a typewriter printing mechanism or a television-type screen for displaying output from the computer. The human user can use any or all of these devices for communicating with the computer.

When a human depresses a key on the terminal, for example, a group of bits, representing that character, is generated and transmitted to the CPU. The action that the CPU takes then depends on what the computer is currently *programmed* to do—often the data bits are simply moved to a cell in memory to await further processing. Similarly, when data from memory is to be displayed on a device (e.g., the line printer or a terminal), the bits representing the data are first transferred by the CPU from the memory cell(s) in which they reside to the CPU. From there they are transferred to the output device, where they are converted from bit packets to the corresponding characters, and printed. That is, the input/output units not only provide communication links between human users and the computing system, but they also provide the necessary conversion of the communicated data between the human-oriented forms and the bit-coded forms.

0.2 Computer Programs

In order for data to be processed in the desired manner, the CPU must transfer data between its parts and make the appropriate modifications of data in its registers, all in a proper sequence. However, the CPU does not automatically know which actions to take next—it must be *instructed* each step of the way. One of the properties of the CPU is its ability to accept, and obey, certain *instructions*. Each of these instructions results in the modification of the data within a register of the CPU or the transfer of data from or to the CPU. Each instruction must be one selected from a predefined set, called the *instruction set*, associated with that CPU.

A particular data processing task is accomplished when the CPU follows, or *executes*, a sequence of instructions. The human desiring such processing must specify an appropriate sequence of instructions. Such a sequence is called a *computer program*, and the process of designing a computer program is called *programming*. This section introduces instructions, programs, programming, and related concepts. Programming is the topic of the next section also, and indeed is the major concern of this book.

0.2.1 Concept of an Instruction

Whatever the data processing task, the computer cannot do it all at once. It is done in tiny steps, one after the other. Each step is an extremely simple one, such as adding 1 to a number in a CPU register, or transferring one piece of data from the CPU to the main memory. Not only is each individual step very simple and limited, the CPU is capable of performing only a limited number of different kinds of such simple steps. Moreover, the CPU performs one of these steps only when it is instructed to do so.

For each individual step or action that the computer is capable of there is a corresponding instruction that can be used to cause that action to occur. In considering the kinds of actions that the computer is capable of, one must think in terms of the various instructions associated with (supplied with) that computer. Examples of such instructions are shown in Figure 0.4.

In Figure 0.4 the <things in pointed brackets> indicate a choice of options. For example <register id> refers to the identification of any one of the registers in the CPU. Suppose that a certain computer has three CPU registers, identified respectively as R1, R2, and R3. Then the specific instruction

ADD REGISTER R3 TO REGISTER R1

could be used to cause the current contents of register R3 to be added to the current contents of R1, with the result replacing the original value in R1.

There are three general classes of instructions represented in Figure 0.4. Instructions (*a*) and (*b*) specify the transfer of data between the CPU and other parts of the system. The first of these transfers data from <source id> to one of the CPU registers; the second transfers data from a CPU register to <destination id>. The <source id> may be a memory cell, I/O device, or another CPU register, for example:

MEMORY CELL M27
I/O DEVICE D4
REGISTER R2

Similarly, <destination id> may be a memory cell, I/O device, or another CPU register.

Instructions (*c*) and (*d*) of Figure 0.4 specify the modification of contents in CPU registers. The ADD instruction in (*c*) is typical of the several instructions available to perform simple arithmetic. The COMPLEMENT instruction in (*d*) is typical of the several "logic" instructions usually available. Logic instructions involve modification of a

1 MOVE <source id> TO REGISTER <register id>

2 MOVE REGISTER <register id> TO <destination id>

3 ADD REGISTER <register id> TO REGISTER <register id>

4 COMPLEMENT REGISTER <register id>

5 GOTO <instruction id>

6 IF REGISTER <register id> ALL ZEROS, GOTO <instruction id>

Figure 0.4. Examples of instructions.

register's bits in ways that are often quite useful but not easily achieved with arithmetic operations. The COMPLEMENT instruction, for example, "flips" all the bits (changes 0's to 1's and 1's to 0's) in the specified register. If register R2 contains the bits 01101000 before execution of the instruction

COMPLEMENT REGISTER R2

it contains 10010111 afterward.

Instructions (*e*) and (*f*) are examples of "branching" instructions, and will be discussed in the next section. The three classes of instructions—data transfer, data modification, and branching—represent the principal actions that any computer can take. A specific computer usually has many variations of instructions in each of these classes, rather than just the two shown in Figure 0.4, including some quite specialized instructions and even instructions combining some aspect of two or more of these classes. The total size of a typical actual instruction set is in the vicinity of 100, and often more. While each computer has instructions falling into these three general classes, and has instructions equivalent to those in Figure 0.4, the number of instructions and nature of each instruction vary considerably from computer to computer.

0.2.2 Instruction Sequences

The processing accomplished by executing any one instruction is really quite small. A practical data processing task can be accomplished only by the execution of a great many individual instructions. Moreover the order in which various instructions are executed determines what processing is actually done. Therefore any practical computer task requires the specification of a sequence of instructions to be executed—that is, requires a computer program. The programmer (person designing the sequence of instructions) writes down a list of instructions in the order that the instructions are to be executed.

Suppose, for example, that a number representing the PRICE of an item is stored in memory cell M146, that the sales TAX is stored in memory cell M23, and that the total COST of the item (PRICE + TAX) is to be stored in memory cell M5914. The following sequence of instructions, executed in the order shown, will accomplish this:

1 MOVE MEMORY CELL M146 TO REGISTER R1
2 MOVE MEMORY CELL M23 TO REGISTER R2
3 ADD REGISTER R2 TO REGISTER R1
4 MOVE REGISTER R1 TO MEMORY CELL M5914

Actually a more abbreviated form of each instruction is used, such as the following:

1 MOVE M146, R1
2 MOVE M23, R2
3 ADD R2, R1
4 MOVE R1, M5914

This sequence of four instructions constitutes a simple computer program.

The normal sequence of instruction execution in a program is the order in which the instructions are listed. Departure from this normal sequence is often needed, however, and the purpose of the branching instructions is to specify such departure. Consider, for example, the task of inputting a series of numbers from a terminal (I/O DEVICE D2), adding them together, and transferring the sum back out to the terminal. An input value of zero is to signify the end of the input values. The following program accomplishes this (assume that register R3 initially contains the value zero):

```
1 MOVE D2, R2
2 IF R2 IS 0, GOTO 5
3 ADD R2, R3
4 GOTO 1
5 MOVE R3, D2
```

In this program instruction (1) represents the inputting of the next number, instruction (5) represents the outputting of the sum, and instruction (3) adds the next number to the sum. Instructions (2) and (4) are branching instructions, and specify which instruction is to be executed next. After instruction (3) is executed, instruction (4) is encountered. Instruction (4)'s action is to cause instruction execution to resume at instruction (1)—instruction (4) is called an *unconditional branch*. Instruction (2) is also a branch instruction, but the branch occurs only if the value of register R2 is zero at the time of execution of instruction (2), otherwise execution continues with instruction (3). Instruction (2) is an example of a *conditional branch*.

Note that instructions (1), (2), (3), and (4) of the preceding example are executed repetitively—perhaps many, many times—until R2 contains the value zero when instruction (2) is executed. Most computer programs involve branching instructions to help achieve the desired processing. When branching is used to specify reexecution of a group of instructions, then execution of even a very short program can result in the execution of a great many—perhaps millions or billions—individual instruction steps. Whereas many individual instructions may be executed for a certain task, each instruction takes perhaps only a millionth of a second to execute. Therefore processing by computer may proceed very rapidly, even for complex tasks requiring millions of steps.

0.2.3 Programming Languages

It should be clear from the preceding sections that computers are basically very simple-minded devices, capable only of performing a few very simple operations, and then only as instructed. To achieve any sort of practical processing, a human must devise a sequence of instructions which, when executed by the machine, will result in the desired processsing. Because each instruction is so simple, and so limited in its actions, a practical program typically consists of a very long sequence of individual instructions. Devising such a program usually is no simple task and is susceptible to errors because the programmer must think in terms of the capabilities (and limitations) of the machine's primitive instruction set rather than in the high-level logic terms of the problem. In terms of the first example of the previous section, for example, the programmer must devise the following sequence of instructions

```
1 MOVE M146, R1
2 MOVE M23, R2
3 ADD R2, R1
4 MOVE R1, M5914
```

when what is wanted is simply PRICE = COST + TAX.

In the mid-1950s it was recognized that a (very complex) computer program could accept (as input) a statement such as PRICE = COST + TAX, analyze its sequence of characters, and subsequently generate (as output) the corresponding four machine instructions shown above. When such a program was available, programmers could then write sequences of statements in a high-level language, more like the mathematics and natural language they were familiar with, and in terms of which the data processing problem of interest could be more easily stated. The computer itself could then be used to translate the high-level language program into an equivalent sequence of ma-

chine instructions; this sequence of machine instructions could then be executed to perform the desired processing. The process of converting the high-level statements to sequences of machine instructions was known as formula translation, and that original high-level language was therefore called Fortran. Warily at first, not fully trusting the translation process, programmers began writing their programs in Fortran.

Quickly Fortran became a roaring success. Programmers found that they could write programs more quickly and reliably in Fortran than in machine language, and found the use of Fortran more interesting and satisfying. And, while that first translator program was far from perfect, a development had occurred that would change the nature of programming dramatically and permanently—large-scale programming in a high-level language had been born. In the succeeding decades hundreds of high-level languages have been devised, each having its own translator; most, however, are conceptually similar to Fortran in most respects. And today, the vast majority of programs are written in Fortran and other high-level languages, rather than in machine-level instructions. That initial Fortran result has been confirmed many times over—that programs can be written more quickly and reliably in a high-level language.

In the days since that first Fortran translating program, a very great deal has occurred in the field of computing. Knowledge about the translating process, and the writing of translating programs, has increased tremendously. In the early days of Fortran all of the computations were of a numerical nature—solving complex numerical problems of science and engineering. Steadily nonnumerical processing applications, such as text analysis and process automation, have increased, to the point where they now account for more computer usage than numerical processing. And computing systems themselves have undergone a tremendous series of developments. New since those first days of Fortran, for example, are the major developments of disk storage systems and time sharing.

That first Fortran, known subsequently as Fortran I (vintage about 1955), soon became inadequate in the face of the rapid development within the computing field. Fortran therefore underwent several stages of evolution until, in 1966, a version, known as Fortran IV, was standardized by the American National Standards Institute. Thus Fortran IV became the first officially standardized language, and translators for this language appeared on virtually every computer model manufactured. The computing field continued to develop rapidly, and so, too, did Fortran continue to evolve. Today, the 1966 Fortran standard has been superceded by a more modern version, called Fortran 77 (it was completed in 1977). While Fortran 77 is a much more sophisticated language than the 1955 version, most of the programs written in the 1955, 1966, and other versions will work properly in the 1977 version.

Today the computing world teems with high-level languages. Very few, however, have been as successful and as venerable as Fortran. And it remains, in Fortran 77, a good language in which to program. It will no doubt continue to be a good language—continuing its evolution to correct any deficiences that it might still have, and adapting to the ever-changing scene in computing.

0.2.4 Program Translation

A program written in a high-level language, such as Fortran, is not directly *machine readable*. That is, the machine does not directly "understand" Fortran statements. A Fortran program must first be converted to an equivalent sequence of machine instructions before that program can be executed. This conversion—the translation from Fortran to machine instructions—is usually called *compiling*, or *compilation*, and the program that does the translating is called the *compiler*. Thus, before a Fortran program can be executed it must be compiled.

Compilation is therefore an important phase in the processing of a Fortran program. Much will be said in this book about the compilation phase and some of the actions taking place during program compilation. The most important of these actions, of course, is the generation of sequences of machine language instructions that correspond to the Fortran statements. Another important action is that of allocating main memory storage for data used in the program, such as for COST and TAX of a pre-

此为试读,需要完整PDF请访问: www.ertongbook.com