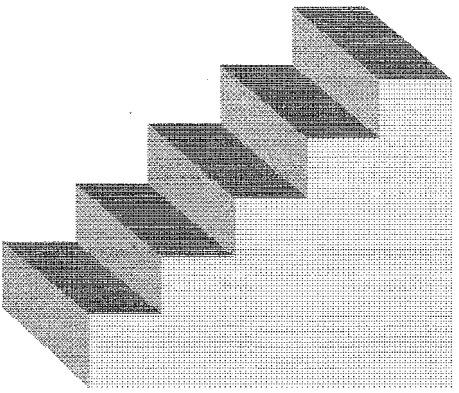

PROGRAMMING IN FORTRAN

Fourth Edition

Ömer Akgiray

PERMISSION TO COPY AND DISTRIBUTE:

This book may be copied and distributed in digital or printed form provided that the front cover that contains the name of the author and the title of the book is included with each copy. Individual chapters may be copied and printed in the same way.

e-mail:

omer.akgiray@marmara.edu.tr

PREFACE

FORTRAN was the first widely used high-level programming language, and it continues to be of great importance in scientific computing. It is an old language, but despite recurrent predictions of its imminent death, it is still going strong and is likely to continue to do so for some time. In particular, many engineers and scientists continue to use FORTRAN as their primary programming language. This is because FORTRAN is particularly well suited for scientific and technological problems, as well as to other problem areas with a significant numerical or computational content.

It is true that the C programming language is superior to FORTRAN 77 and earlier versions of FORTRAN in a number of ways, and it may have become the preferred language for many engineers and scientists in the early 1990s. There are, however, numerous FORTRAN 77 programs in current use in engineering and science. There are also many well-documented, well-tested, and high quality FORTRAN 77 programs in the published literature. As a result, even for those who prefer to use C or another language when writing new programs, there is a need to learn FORTRAN--if for nothing else--just to be able to understand, and, when deemed useful, to adapt and to utilize such programs.

It should be emphasized that the next version of the language, Fortran 90, contains many new added features that arguably make it the most suitable language for scientific computing. Fortran 90, which contains the whole of FORTRAN 77 as a subset, is a powerful and flexible programming language. Fortran 90's array processing features, for instance, are considerably more powerful than those of any other programming language.

Fortran 90 is a rich and large language. This is partly due to the fact that it contains many features which are included for compatibility with earlier versions of the language. A standard-conforming FORTRAN 66 or FORTRAN 77 program written over 20 to 30 years ago is also a perfectly valid Fortran 90 program. Considering how quickly almost anything related to computer hardware and software becomes obsolete, this is a tribute to the longevity of FORTRAN programs.

One of the advantages of FORTRAN 77 over C has been that it is easier to learn, because its "mental model" of the computer is much simpler. For example, in FORTRAN 77 the programmer can avoid learning about pointers and memory addresses, while these are essential in C. Because of this relative simplicity, for many simple yet significant programming tasks, FORTRAN 77 generally requires much less computer science knowledge of the programmer than C does, and is thus much easier to use.

As noted above, however, Fortran 90 is a large language and, as a result, learning the whole of Fortran 90 is not as easy. The primary audience to this book has been the freshman engineering students with no prior programming experience. I think that it is not possible to teach all of Fortran 90 to these students satisfactorily within a single semester. Therefore a proper subset of Fortran 90 must be covered in such a single-semester introductory course. Here an instructor has two choices: (1) Teach FORTRAN 77 as fully as possible and introduce some of the important new features of Fortran 90 to the extent that time allows. (2) Teach a subset of Fortran 90, such as the F programming language, which contains most of the new

and modern features of Fortran 90, but not certain older features of FORTRAN 77 which were retained in Fortran 90 for backward compatibility. Most authors and instructors have adopted the latter approach.

I have decided to continue to teach FORTRAN 77 in the class and to present it separately in this book (by introducing Fortran 90 in the last chapter) for the following reasons:
(1) It is possible to write efficient and structured programs using FORTRAN 77.
(2) FORTRAN 77 is relatively easy to learn and use.
(3) Many engineers and scientists continue to develop new code using FORTRAN 77 despite the availability of commercial Fortran 90/95 compilers for essentially all types of computers.
(4) There are several widely available FORTRAN 77 compilers.
(5) FORTRAN 77 programs are also valid Fortran 90 programs, and they can be compiled using modern Fortran 90/95 compilers.
(6) FORTRAN 77 is a subset of Fortran 90, and learning FORTRAN 77 means also learning a significant fraction of Fortran 90. Such an effort, therefore, is not wasted in any way.
(7) The engineering and scientific literature of the last 30 years is replete with FORTRAN 66 and FORTRAN 77 programs. Ability to understand, use, modify, adapt, and maintain such older code can be expected to remain a valuable skill for years to come.

The engineering student is, of course, strongly encouraged to continue increasing his/her knowledge and skills by learning and adopting Fortran 90/95 (and the next expected version, Fortran 2000) as well as some of the other widely-used programming languages and software tools such as C, Java, Mathcad, Mathematica, etc.

All of the FORTRAN 77 programs given in this book were tested using at least one of the following two compilers: (i) Microsoft FORTRAN PowerStation Version 1.0a. (ii) Lahey Computer Systems' FORTRAN 77 Compiler Version 4.0. Many programs were tested with both compilers. The Fortran 90 programs presented in Chapter 7 were tested using Microsoft Fortran PowerStation Version 4.0.

There are several books that contain comprehensive information about FORTRAN. It would be very difficult to write a new book that betters the existing textbooks. My goal in writing this book was very much unpretentious: I aimed at an easy-to-read textbook on FORTRAN, one that beginning students can be asked to read from cover to cover, and one from which they can learn FORTRAN rather quickly. My approach has been, for the most part, to teach by example. I have tried to avoid "real engineering applications" which tend to distract the reader from the main point being made. The examples are, therefore, quite simple and short. Many subtle details of the language, however, are thoroughly covered and exemplified. You are strongly encouraged to type and run each program in the book, and to compare the results so obtained with those given in the book. In this way, you will quickly become familiar with the process of typing in, compiling, and running FORTRAN programs.

Ömer Akgiray
Marmara University, Göztepe, İstanbul
March 2000

CONTENTS

CHAPTER 1: *BASIC FEATURES OF FORTRAN*

1.1 Introduction

A digital electronic computer is, at the most basic level, a collection of electronic and electromechanical components and devices. What sets a computer apart from other machines which may be built from similar component parts are two characteristics: *memory* and *capability to be programmed*. That part of a computer in which information is stored is referred to as the memory. There are two main types of information stored in a computer's memory, namely, *programs* and *data*. A program is a set of instructions which the computer is to obey. Data (numbers, words, sound, etc.) are processed by the computer in a way defined by a program. A programming language is required to program a computer.

The most basic programming language is **machine language**. A program written in the machine language of a computer is a collection of very detailed, cryptic instructions that control the computer's internal circuitry. This is the native tongue of the computer and is the only language that the computer actually understands. In the earliest days of computers, to program a computer meant to write programs in the machine language of that computer. There are two main disadvantages to using a machine language: First, machine languages are very cumbersome to work with, and second, every type of computer has its own machine language. Thus, a machine language program written for one type of computer cannot be used on another type of computer without significant alterations.

Machine language instructions are usually written in binary code (i.e. strings of 0's and 1's), and each statement corresponds to one machine action. Higher in the hierarchy of programming languages are **assembly languages**. Each statement of an assembly language program corresponds to one machine language statement, but the statements themselves are written in symbolic code that is easier for people to read. As an hypothetical example, the following three instructions constitute a tiny machine language program:

```
00010100000000000000000000001100100
00010000000000000000000000001100101
00011000000100000000000000001100110
```

The same three steps written in an assembly language may look like

```
CLA 100
ADD 101
STO 102
```

where `CLA` and `STO` stand for "clear accumulator and add" and "store," respectively. The decimal numbers 100, 101, and 102 (and the corresponding binary numbers

1100100, 1100101, and 1100110[1]) refer to storage locations in the memory. The first step (of both programs) causes the number in the storage location 100 to be brought to the accumulator. (Roughly speaking, the accumulator is the storage location where a computer stores the result of an arithmetic operation.) In the second step, the number in location 101 is added to the number that has just been placed into the accumulator. The result of this addition is stored in the accumulator. In the third step, the number currently present in the accumulator (which is the sum of the numbers in storage locations 100 and 101) is stored in the storage location 102. A programmer can usually assign names (such as `A`, `B`, and `C`) to memory locations such as 100, 101, and 102. The program above may in that case be written as follows:

```
CLA A
ADD B
STO C
```

An **assembler** program is required to translate assembly language programs into machine language instructions.

A **high-level language** is a computer programming language that allows people to write programs without having to understand the inner-workings of a computer. FORTRAN, BASIC, COBOL, Pascal, Ada, C, C++, and Java are examples of high-level languages. A machine language is at the lowest level, since machine language programming requires detailed knowledge of the computer's inner-workings. An assembly language is at a slightly higher level than a machine language. Assembly language programs are lengthy and hard to read, and the same assembly language cannot be used on different types of computers. A high-level language, on the other hand, resembles an everyday language (English) and may be used on different types of computer systems. Furthermore, while a statement of an assembly language program corresponds to exactly one machine language statement, a single statement written using a high-level language may correspond to several machine-level instructions. For example, to add the values stored in the memory locations named `A` and `B`, and to store the result in the memory location named `C`, a single FORTRAN statement is used:

```
C = A + B
```

A high-level language program must first be translated into machine language instructions before the program can be executed. A special program, namely a **compiler**, is employed to carry out this translation. The high-level language program is called the **source program**, and the translation is referred to as the **object program**.

---

[1] Calculation of the decimal code corresponding to a binary number is best explained with an example. Consider the binary code **1100100**. Then $\mathbf{1}x2^6+\mathbf{1}x2^5+\mathbf{0}x2^4+\mathbf{0}x2^3+\mathbf{1}x2^2+\mathbf{0}x2^1+\mathbf{0}x2^0 = 100$ is its decimal equivalent. Verify that the decimal equivalents of 1100101 and 1100110 are 101 and 102, respectively.

A high-level language program can be run on any type of computer provided that a compiler exists for that type of computer. Thus, in high-level languages, differences between different types of computers are handled by the compiler, which is specially written for each type of machine. Many computer manufacturers provide compilers for the commonly used high-level languages such as FORTRAN and C. For "IBM-compatible" PC's, several compilers (developed by different software companies) for each such language are commercially available.

FORTRAN (short for "IBM Mathematical **FOR**mula **TRAN**slation System") was the first widely used high-level programming language. The FORTRAN programming language was developed by IBM (International Business Machines Corporation) in the 1950s, and its first commercial version (FORTRAN I) was announced in 1957. This was followed a year later by FORTRAN II (1958), an improved version of the language.

While FORTRAN II established the FORTRAN language among a large body of users, its successor FORTRAN III (1958) was never released to the public. It made it possible to use assembly language code right in the middle of FORTRAN code. This allowed writing more efficient programs, but the advantages of a high-level programming language are lost (e.g. portability, ease of use).

The general spread of the FORTRAN language was due, in part, to IBM's free distribution of its FORTRAN compilers with its mainframe computers. In the meanwhile, other manufactures started to write compilers for FORTRAN, and by 1963 there were more than 40 different FORTRAN compilers in existence. This development was of great importance, because once a program had been written for one computer using FORTRAN, it could be moved to another computer with little or no change.

FORTRAN II released by IBM and other manufacturers' FORTRAN compilers, however, contained certain machine-dependent features. Furthermore, there were some incompatibilities between different compilers. In response to these problems, IBM developed and released **FORTRAN IV** (in 1962) which did away with the machine-dependent features of FORTRAN II. This new version, FORTRAN IV, was highly successful because programs written in FORTRAN IV were almost totally independent of the type of computer on which they were to be run. Such programs could easily be transferred to another type of computer, as long as that computer had a FORTRAN IV compiler.

The next significant development was the definition of **FORTRAN 66** (USASI standard FORTRAN, March 1966) which was based largely on IBM's FORTRAN IV. FORTRAN 66 was the first high-level language standard in the world.

Standard **FORTRAN 77** (ANSI standard developed in 1977, approved in April 1978)[2] replaced the older FORTRAN 66. The FORTRAN 77 programming language was ratified as an international standard in 1980. An important new international standard was published in August of 1991 (ISO/IEC[3], 1991) and this version was named **Fortran 90**[4].

The development of Fortran has not ceased. The most recent version of the language is a minor revision of Fortran 90 known as **Fortran 95** (ISO/IEC, 1997). A further revision of the language, named **Fortran 2000**, is expected to be published by the end of 2002.

"FORTRAN is not a dead language, the majority of programs used and developed in the scientific and engineering communities are still written in FORTRAN 77 or Fortran 90. High-performance computing is mostly done in one of the parallel dialects of FORTRAN. FORTRAN 77, and of course the much improved Fortran 90, are better suited for numerical computation than most programming languages. Fortran is expected to further improve in this respect."[5]

Mention of the word FORTRAN in the first part (Chapters 1 through 6) of this text refers to FORTRAN 77, which is the most widely used version of the language. Certain important new features of Fortran 90 are mentioned in the footnotes. The student using a Fortran 90/95 compiler should read these footnotes carefully. A separate discussion of Fortran 90 is presented in the second part of this book.

All standard FORTRAN 66 programs will run under FORTRAN 77, but not vice versa. Similarly, any standard-conforming FORTRAN 77 program is also a standard-conforming Fortran 90 program, and the existing investment in FORTRAN 77 programs is fully preserved. This means that you can compile your FORTRAN 77 programs using a Fortran 90/95 compiler.

FORTRAN standards are designed to make it possible to run the same FORTRAN program on any computer without worrying about the variations in the details of the language. Typically, a particular compiler manufacturer's version of FORTRAN 77 will include all of the features defined in the ANSI standard, plus additional features devised by the manufacturer. To be easily transportable from one type of computer to another (or, from one compiler to another on the same computer), a program should not use any features that are not in the standard. Certain non-standard features which are recognized by the majority of modern FORTRAN 77 compilers, however, are utilized in this book.

---

[2] American Standards Association (ASA) became the United States of America Standards Institute (USASI) in 1966. USASI adopted its present name, the American National Standards Institute (ANSI), in 1969. ANSI is the U.S. member body of the International Standards Organization (ISO).

[3] IEC is short for International Electrotechnical Commission.

[4] Lower case letters are used to spell Fortran 90, unlike the upper case letters used for its predecessors.

[5] *User Notes on FORTRAN Programming* (1996-1998), http://sunsite.icm.edu.pl/fortran/.

1.2 Typing FORTRAN Statements

All FORTRAN compilers recognize the following list of characters:

Upper case alphabetic characters:	A to Z
Digits:	0 to 9
Symbols:	+ - * / . , ' = $ ( ) :

These characters, together with the *blank space*, constitute the **FORTRAN character set**. The digits and the alphabetic characters are collectively referred to as **alphanumeric characters**.

FORTRAN statements are normally formed using characters from the FORTRAN character set defined as above. It may be noted that the standard FORTRAN character set does not include lower case letters[6]. However, most FORTRAN 77 compilers in current use permit the use of lower case letters inside character strings, and some allow lower case everywhere in a program. Furthermore, with many compilers, certain characters not included in the FORTRAN character set can be used as continuation marks or inside strings (see Example 1.3). Lower case letters can safely be used in comments. (What we mean by "comment," "variable," and "character string" will be clear as you study this chapter.)

In writing FORTRAN statements to a file, there are certain rules that must be followed. These rules can be outlined as follows[7]:

1. Not all 80 columns of a line are used for a FORTRAN statement. The statement must be typed within the **statement field**, i.e. columns 7 through 72.
2. Columns 73 through 80 are ignored by the FORTRAN compiler. Some programmers use these columns for numbering their FORTRAN statements, while others leave these columns unused.
3. Columns 1 to 5 of a line can be used for **labels**. A label in FORTRAN is an unsigned positive integer with a maximum of five digits and is also referred to as a *statement number*. Different statements cannot be assigned the same label.
4. Column 6 is called the **continuation column**. If a FORTRAN statement is too long for the 66 columns in the statement field, it can be continued in the statement fields of subsequent lines (called **continuation lines**) in the file. To do this, any symbol other than 0 (zero) is typed in column 6 of each continuation line.

---

[6] The Standard Fortran 90 Character Set includes, in addition to those listed above, the lower case letters, and the following characters: % > < ? ; & " ! _

[7] Fortran 90 allows the use of a **free form** in typing programs. In the free form, statements can be written anywhere on the line. The relevant rules are explained in Chapter 7.

Typically, at most 19 continuation lines are allowed[8]. The first line of a statement is called the **initial line**. A zero (or blank) in column 6 indicates an initial line.

5. A **comment** can be written on a line by typing "C" or "*" (an asterisk) in column 1 of that line[9]. Comments are ignored by the compiler.
6. Blanks in FORTRAN programs are ignored by the compiler and may be used to improve the appearance and readability of a program.

The application of these rules will be illustrated with several example programs given in the sections that follow.

1.3 Writing Programs in FORTRAN

Taking a look at an actual program written in FORTRAN is probably the quickest way to gain an appreciation of this programming language. As a beginning, a very simple example will be considered: A program that displays the phrase "FORTRAN is beautiful." on the computer screen. Here is a FORTRAN program that accomplishes this task:

Example 1.1

```
PROGRAM FIRST
PRINT*, 'FORTRAN is beautiful.'
END
```

When this program is compiled and executed, the following output appears on the screen of the computer:

```
FORTRAN is beautiful.
```

We shall now take a close look at our first program. The first line of the program informs us that the name of the program is FIRST. Any name that can be used as a FORTRAN variable name can also be used as a program name. (We shall examine rules for the formation of variable names later in this chapter.)

It is actually not mandatory to give names to FORTRAN programs, and you could omit the first line in the above program. It is, however, good practice to name each program for easy identification, especially if you write a large number of programs. You should choose a meaningful program name, e.g. a name that reminds you what the program does.

The file that contains the program may be named *programname*.for (e.g. FIRST.FOR in this case). On many systems, however, the use of a different file name is allowed.

The second line of the program contains the PRINT statement. This statement instructs the computer to display the string of characters contained within the apostrophes.

The END statement terminates the execution of the program. The END statement also marks the physical end of the list of FORTRAN statements.

Example 1.2:

Next we modify our first program to display an additional string of characters which is too long to fit within the statement field of a single line.

---

[8] Up to 39 continuation lines are allowed in Fortran 90.

[9] An exclamation mark, i.e. ! can be used to start a comment in Fortran 90. See chapter 7.

Typically, at most 19 continuation lines are allowed[8]. The first line of a statement is called the **initial line**. A zero (or blank) in column 6 indicates an initial line.

5. A **comment** can be written on a line by typing "C" or "*" (an asterisk) in column 1 of that line[9]. Comments are ignored by the compiler.
6. Blanks in FORTRAN programs are ignored by the compiler and may be used to improve the appearance and readability of a program.

The application of these rules will be illustrated with several example programs given in the sections that follow.

1.3 Writing Programs in FORTRAN

Taking a look at an actual program written in FORTRAN is probably the quickest way to gain an appreciation of this programming language. As a beginning, a very simple example will be considered: A program that displays the phrase "FORTRAN is beautiful." on the computer screen. Here is a FORTRAN program that accomplishes this task:

Example 1.1

```
PROGRAM FIRST
PRINT*, 'FORTRAN is beautiful.'
END
```

When this program is compiled and executed, the following output appears on the screen of the computer:

```
FORTRAN is beautiful.
```

We shall now take a close look at our first program. The first line of the program informs us that the name of the program is FIRST. Any name that can be used as a FORTRAN variable name can also be used as a program name. (We shall examine rules for the formation of variable names later in this chapter.)

It is actually not mandatory to give names to FORTRAN programs, and you could omit the first line in the above program. It is, however, good practice to name each program for easy identification, especially if you write a large number of programs. You should choose a meaningful program name, e.g. a name that reminds you what the program does.

The file that contains the program may be named *programname*.for (e.g. FIRST.FOR in this case). On many systems, however, the use of a different file name is allowed.

The second line of the program contains the PRINT statement. This statement instructs the computer to display the string of characters contained within the apostrophes.

The END statement terminates the execution of the program. The END statement also marks the physical end of the list of FORTRAN statements.

Example 1.2:

Next we modify our first program to display an additional string of characters which is too long to fit within the statement field of a single line.

---

[8] Up to 39 continuation lines are allowed in Fortran 90.

[9] An exclamation mark, i.e. ! can be used to start a comment in Fortran 90. See chapter 7.

```
      PROGRAM FIRST2
C*************************************************************************
C  This is a modification of our first FORTRAN program.
C  In this program, we learn how to use a continuation line.
C  A continuation mark (any symbol other than 0) is typed
C  on the 6th column of the continuation line.
C*************************************************************************
      PRINT*, 'FORTRAN is beautiful.'
      PRINT*, 'Programming in FORTRAN can be fun-if you are organized, c
     &areful and patient.'
      END
```

The expected output is the following:

```
FORTRAN is beautiful.
Programming in FORTRAN can be fun-if you are organized, careful and patient.
```

Note that the first letter of the word "careful" is in the 72nd column of the file, and anything typed beyond this column would be ignored by the FORTRAN compiler. To complete the PRINT statement, we have used a continuation line which is recognized by the compiler by means of a continuation mark ("&" in this case) in the 6th column.

Example 1.3:

Type, compile and run the following program. You should, for convenience, type the characters in the PRINT statements in the order they are placed on your keyboard.

Remember that, strictly speaking, only the characters in the FORTRAN character set may be used in a FORTRAN program. Lower case letters, however, are allowed by many (if not all) of the commercially available compilers. On the other hand, the use of characters such as the "Turkish characters" (ç, ğ, ı, ö, ş, ü, Ç, Ğ, İ, Ö, Ş, Ü) may lead to unexpected results. You may modify this program to try to print characters such as ç, ş, etc. that are not in the FORTRAN character set to see if your FORTRAN system can handle them correctly.

```
      PROGRAM CHARS
C*************************************************************************
C   A FORTRAN source file may usually contain:
C   1. All upper case and lower case letters
C   2. Digits 0 thru 9
C   3. Many of the symbols that appear on the keyboard
C*************************************************************************
      PRINT*, 'The characters on my keyboard:'
      PRINT*, '------------------------------'
      PRINT*,
     % 'Symbols: ~ ! @ # $ % ^ & * ( ) _ + ` - = , . / < > ? { } | [ ] \
     % : " ; '' '
      PRINT*,
     * 'Digits: 0 1 2 3 4 5 6 7 8 9'
      PRINT*,
     & 'Letters:  QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm'
      END
```

The output of this program is as follows:

```
The characters on my keyboard:
------------------------------
Symbols: ~ ! @ # $ % ^ & * ( ) _ + ` - = , . / < > ? { } | [ ] \ : " ; '
Digits: 0 1 2 3 4 5 6 7 8 9
```

```
Letters:  QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm
```

Notice the effect of the

```
PRINT*, ' '
```

statement. Make sure that you understand how the continuation marks and continuation lines are used. Note also that two adjacent apostrophes within a character string appear as a single apostrophe in the output. This is explained in the next section.

Exercises:

1. Delete the first line of the program in Example 1.1 and rerun the program. Is there any difference in the output of the modified program?

2. It is said that the simplest FORTRAN program consists of a single line containing the `END` statement. What happens when you run such a program? (Try it.)

3. Retype the program in Example 1.1 using lower case letters for the FORTRAN statements. (For example, type `end` instead of `END`, `print` instead of `PRINT`, etc.) What happens? Try also mixing lower and upper case, e.g. `End` or `eND`, etc. What happens?

4. Consider the `PRINT` statement in Example 1.1, i.e.

```
PRINT*, 'FORTRAN is beautiful.'
```

(note the blank space after the comma)

Try each of the following modifications of this statement:

```
PRINT*,        'FORTRAN is beautiful.'
PRINT*,'FORTRAN is beautiful.'
PRINT*,'FORTRAN     is     beautiful.'
```

Compile and run the program again after each modification. Observe how the output is affected (if at all) by adding or removing blank spaces. Are the blank spaces within character strings ignored?

5. In Example 1.2, the symbol `&` was used as a continuation mark. Try using other characters for this purpose. What happens if you use `0` (number zero)?

1.4 Constants and Variables in FORTRAN

In FORTRAN, there are six basic **data types**: *integer, real, double precision, complex, character,* and *logical.* In this section, we shall restrict attention mainly to integer, real, and character data. Double precision and complex data types are briefly explained. The logical data type is discussed in Chapter 2.

An **integer constant** is any signed or unsigned whole number without a decimal point or any other punctuation. Unsigned numbers are assumed to be positive. The following are some acceptable integer constants:

```
-7        +13        0        189675        -78
```

The following, on the other hand, are unacceptable as integer constants:

```
68.0    (contains a decimal point)
13,234  (contains a comma)
```

`-75.` (contains a decimal point)
`--10` (two minus signs)

A **real constant** can be written in two forms: the **decimal form** and the **exponential form**. In both forms, there is a finite number of digits and a decimal point. In the exponential form, a real constant contains a second part starting with the character `E` followed by a signed or unsigned integer with at most two digits. The following are valid as real constants:

```
-8.5     37.     +1345.01     0.0     -0.012
```

The following are all valid representations of the same real constant (`0.027`):

```
0.027   2.7E-02   27.0E-03   +.27E-01   0.0027E01   0.0027E+01
```

The following are not acceptable as real constants:

`123` (no decimal point)
`1,345` (contains a comma)
`.34E982` (contains three digits after `E`)

All types of data (integers, real numbers, characters, etc.) are internally represented as strings of binary digits (**bits**) in a computer's memory. Internal representation of character data will be discussed later (cf. Chapter 5) in some detail.

Consider here the representation of integers by strings of binary digits. There are only two possible binary digits: 0 and 1. Using a single bit, therefore, only two decimal values (e.g. 0 and 1) can be represented. A group of two bits, on the other hand, has four different combinations (00, 01, 10, 11) and therefore can represent four (2^2) different decimal values e.g. 0 to 3. Similarly, a string of three bits can be used to represent eight (2^3) different values, e.g. from 0 to 7. A string of 8 bits (a **byte**) can represent $2^8 = 256$ different values, whereas a 32-bit string can be used to represent $2^{32} = 4{,}294{,}967{,}296$ possible values.

Normally, half of all possible values are reserved for representing negative numbers, one value for representing 0 (zero), and the remaining values (half of all values minus one) for representing positive values. Thus, a group of 8 bits can be used to represent numbers from –128 to +127. There are several different schemes

for representing negative numbers in a computer's memory. These will not be described here; the interested reader may consult texts on computer science[10].

The amount of memory devoted to storing an integer will vary from one type of computer to another, but it will usually be 1, 2, 4, or 8 **bytes**[11]. The most common type of integer in modern computers is a 4-byte integer. Since a byte consists of 8 bits, a 4-byte integer is a 32-bit integer. In general, the smallest and the largest numbers that can be stored in an n-bit integer are -2^{n-1} and $2^{n-1}-1$, respectively. For a 4-byte integer, these values are –2,147,483,648 and +2,147,483,647, respectively. Attempts to use an integer larger than the largest possible value or smaller than the smallest possible value result in an error condition called **integer overflow**.

A real number is stored in two parts: the **mantissa** and the **exponent**. This is similar to scientific notation used in science and engineering. For example, a number like 189,300,000 is conveniently written as 0.1893×10^9. Here the mantissa is 0.1893, and the exponent (in the base 10 system) is 9. A computer uses the base 2 (binary) system instead of the base 10 (decimal) system. In the internal representation of a real number, the mantissa contains a number between –1.0 and +1.0, and the exponent contains the power of 2 needed to scale the number to its actual value.

To store real numbers, most modern computers employ 32 bits (4 bytes) of memory divided into two components: a 24-bit mantissa and an 8-bit exponent. There are some computers that use a slightly different division of bits, e.g. 23 bits for the mantissa and 9 bits for the exponent. The number of bits in the mantissa determines the **precision** of the number (i.e. the number of significant digits to which the number can be represented), while the number of bits in the exponent determines the **range** (i.e. the largest and the smallest values that can be represented). For a given size (e.g. 4 bytes) allocated to store real numbers, the higher the precision is, the smaller the allowed range is, and vice versa.

A **double precision constant** is similar to a real constant, but it contains approximately twice the number of digits as an ordinary real constant. (A real constant is sometimes referred to as a *single precision real constant*, whereas a double precision constant is termed a *double precision real constant.*) . A double precision constant has the same form as the exponential form of a real constant, except that the letter `D` is used instead of `E`, and--on most processors--up to 3 digits may be used in the exponent. For example, `-1.54D-7` and `1.2983D+129` are valid double precision constants. Double precision constants must be written in the exponential form. If the exponent is omitted, a constant is interpreted as a (single precision) real constant.

---

[10] See, for example, *Introduction to Computer Science* (1981) by V. Zwass.

[11] A *byte* is the amount of space needed to store one character (cf. Chapter 5). One kilobyte = 1024 bytes, whereas one megabyte (1 MB) = 1024x1024 bytes = 1,048,576 bytes ≈ 1,000,000 bytes.

On a typical PC, a real constant occupies 4 bytes (32 bits) of memory, whereas a double precision constant occupies 8 bytes (64 bits) of memory. Most new computer systems devote 53 bits to the mantissa and 11 bits to the exponent. In that case, 15 to 16 significant decimal digits can be accommodated, and numbers with absolute values as large as $+1.7x10^{+308}$ or as small as $+2.2x10^{-308}$ can be represented in the computer. Remember, however, that some computers may allocate their bits in a different fashion. Older VAX machines, for example, allocated 56 bits to the mantissa and 8 bits to the exponent of their double precision numbers. This gives a range of $+1.18x10^{-38}$ to $+3.4x10^{+38}$ (same as that of real numbers) and 16 to 17 significant decimal digits of precision[12].

While the value ranges and the storage requirements of the various data types vary from one type of computer to another, the following table shows these values on a typical PC.

Type	Bytes	Range
Integer	4	-2,147,483,648 to 2,147,483,647
Real	4	$-3.4\times10^{+38}$ to -1.18×10^{-38} The number 0 $+1.18\times10^{-38}$ to $+3.4\times10^{+38}$
Double precision	8	$-1.7\times10^{+308}$ to -2.2×10^{-308} The number 0 $+2.2\times10^{-308}$ to $+1.7\times10^{+308}$

Attempts to use a number larger than the largest possible value for that type (e.g. $+3.4x10^{38}$ for reals) or smaller than the smallest possible value (e.g. $-3.4x10^{38}$ for reals) result in an error called **overflow** condition. This condition will normally cause a program to abort. Attempts to use numbers with absolute values smaller than the smallest possible positive value that can be represented for that type (e.g. $1.18x10^{-38}$ for reals) will cause an **underflow** condition. Such a number will be set to 0 (zero) by most FORTRAN systems.

A **complex constant** is an ordered pair of real constants. The first constant represents the real part of the complex number, and the second represents the imaginary part. The two real constants are enclosed in parentheses and separated by a comma. For example, `(3.0,2.5)` represents the complex number *3+2.5i*. While not part of standard FORTRAN, a **double complex** data type is defined on many compilers: a double complex constant is an ordered pair of double precision constants.

[12] Chapman (1998), p.419.

A **character constant** consists of a string of characters enclosed in apostrophes[13]. Any character taken from the FORTRAN character set may be included in the string. The following are valid character constants:

```
'PDO-546'  'FORTRAN is beautiful.'  'I don''t like Pascal.'
```

Blanks are permitted in character constants and are significant. Thus, `'STR ING'` and `'STRING'` are different constants.

Earlier it was noted that the standard FORTRAN character set does not include lower case letters. However, most FORTRAN 77 compilers permit lower case letters to be used inside strings. If lower case is allowed, then the case of letters is also significant within character constants. For example, the character constants `'Name'` and `'NAME'` are different.

Furthermore, any particular compiler will almost certainly have codes for characters not included in the standard FORTRAN character set, and these may be used inside character strings. If you are working on a PC, for example, you are probably allowed use any printable ASCII character inside character strings. It should be remembered, however, that a program that uses a non-standard feature may not work on a different type of computer. See Chapter 5 for more information.

When character data are printed, the enclosing apostrophes are omitted. Note that two adjacent apostrophes are used in a string to indicate contraction or possession. The following are examples of invalid character data:

```
'I'm going.'    (missing apostrophe: should be 'I''m going.')
'FORTRAN is beautiful.    (missing apostrophe)
100          (integer constant)
```

FORTRAN permits the use of descriptive symbolic names (called variable names, or simply **variables**) to designate memory cells. The term variable in computer terminology means a memory cell with variable contents. That is, the value stored in such a memory cell can be changed during program execution. (Constants are also stored in memory cells. The content of such a memory cell is not intended to be modified.) If a variable (memory cell) is used to store integer values, it is called an *integer variable*; if it is used to store real values, it is called a *real variable*, and so on.

The rules that must be followed in the formation of FORTRAN variable names are as follows: *A variable name may contain only six alphanumeric characters*[14] *(*`A` *to*

---

[13] In Fortran 90, quotation marks can also be used to delimit character strings. Thus, "`string`" and `'string'` are equivalent.

[14] Fortran 90 allows longer variable names (up to thirty-one characters), and the use of the underscore character and lower case letters in variable names.

Z and 0 to 9), and must always begin with an alphabetic character (A to Z). Standard FORTRAN 77 allows upper case letters only.

Note that the same restrictions apply to program names. Furthermore, a program name must be unique and cannot be used as the name of a variable within the same program unit[15].

A word (such as END, PRINT, REAL, etc.) with a special, predefined meaning for the compiler is termed a **keyword**. Using keywords as variable names makes programs harder to understand. Therefore, although the rules of FORTRAN do not prohibit it, the use of keywords as variable names should be avoided.

Some FORTRAN 77 compilers allow the use of more than six characters, and/or the inclusion of lower case letters, the underscore (_), and the dollar sign ($) in variable names. (You should check whether your FORTRAN system includes these extensions.) In this text, however, we shall take a cautious approach with regard to variable names and assume that the restrictions stated above apply without change. Hence,

```
GOOD        NEW        RESULT        Z21
```

are acceptable variable names, but the following are not:

```
    2XYZ    (The first letter is not alphabetic)
    A*BC    (* is not alphanumeric)
INCOMETAX   (More than six characters)
     END    (Has a special meaning in FORTRAN)
```

If the use of lower case letters in variable names is allowed by your FORTRAN system, remember that the case of letters is insignificant in variable names. Thus, for example, the names NUMBER and Number would refer to the same variable[16].

1.5 Implicit Typing of Variable Names and Type Declarations

FORTRAN uses the following **implicit first-letter typing convention** for variable names: *If the first character of a variable name is* I, J, K, L, M, *or* N, *then that variable is typed as integer. If the first character is one of the other letters (*A *to* H *and* O *to* Z*) then the variable is typed as real.* No variable names are implicitly typed as double precision, complex, logical, or character. Hence, IXY, MONDAY, KOOL are

---

[15] The only program unit we have seen so far is a main program, i.e. a program that starts with the PROGRAM statement and ends with the END statement. Other types of program unit such as functions and subroutines will be discussed later (see Ch.4).

[16] The lower case and upper case letters are distinct in the C programming language. Thus the names number, Number, and NUMBER each refer to a different variable in C.

valid integer variable names, whereas BAY, ZERO, HEY are valid as real variable names.

Frequently, however, it is not desirable to follow the implicit typing rule for variable names. One may, for example, want to use LENGTH as a real variable that represents the length of a specified object. The implicit typing rule in FORTRAN can be overridden by using an explicit **type declaration**. To inform the compiler that LENGTH will be used as a real variable name, the following type declaration should be inserted at the top of the program:

```
REAL LENGTH
```

As another example, assume that a program contains the following statements:

```
REAL NEW, K, MAXIM
INTEGER  AGE, C, D2
```

Thus, NEW, K, and MAXIM are typed as real variables, whereas AGE, C, and D2 are typed as integers, and they will be used as such throughout the program. The following points regarding type declarations and implicit typing should be noted:

1. Type declarations are non-executable FORTRAN statements. As such they provide information to the compiler. They must appear at the beginning of the program before all executable statements.
2. The general format of type declarations is as follows:

   REAL *list of variables*
   INTEGER *list of variables*
   COMPLEX *list of variables*
   DOUBLE PRECISION *list of variables*
   LOGICAL *list of variables*
   CHARACTER*n *list of variables*

   Commas are used to separate the variable names in the *list of variables*. Here n (a positive whole number) is the number of characters stored in a character variable declared in this manner. If character variables of different lengths (say, n_1 and n_2) are to be used within the same program, a separate declaration can be included for each distinct length:

   CHARACTER*n_1 *list of variables*
   CHARACTER*n_2 *list of variables*

   Alternatively, the length specifier for each variable can be placed after the name of that variable in the CHARACTER declaration:

   CHARACTER var_1*n_1, var_2*n_2, ..., var_m*n_m

   The symbols *1 may be omitted when declaring a variable for storing a single character. If the "double complex" data type is recognized by the compiler being used, variables of that type may be declared as follows:

   COMPLEX*16 *list of variables*

3. Any variable that is not explicitly declared in a type declaration is automatically associated with a computer memory cell when it is first encountered in the program. The type of such a variable is determined by the first-letter typing convention.

From what has been said above, it is clear that FORTRAN allows the use of variables not explicitly declared with a type declaration. In the early days of programming, many programmers were unhappy about having to declare variables before using them. As a result, FORTRAN provided this alternative form of determining the type of a variable based on its initial letter. The first-letter typing rule saves time by reducing the number of statements the programmer has to type.

Utilization of this implicit first-letter typing convention, however, can sometimes lead to programming errors which are very difficult to track. To make sure that all the variables used in a program are declared explicitly, the following statement should be inserted at the top of the program (before all type declarations but after the `PROGRAM` statement):

```
IMPLICIT NONE
```

This statement effectively removes the first-letter typing convention and prevents the use of variables not declared explicitly[17].

Example 1.4:

Type, compile and run the following program exactly as it is written here:

```
      PROGRAM ERROR
C*************************************************************************
C   This program illustrates one kind of error that can occur
C   when the use of undeclared variables is allowed.
C*************************************************************************
      NUMBER = 27
      PRINT*, 'The number is ', NUMBR
      END
```

The output of this program will probably be

```
The number is       0
```

Note that although the integer variable `NUMBER` is assigned the value 27, the value displayed is zero. This happens because, the value printed is that of `NUMBR` which is a distinct variable that is introduced as a result of a typing error. The statement

```
      PRINT*, 'The number is ', NUMBR
```

[17] While the `IMPLICIT NONE` statement is not in the ANSI standard, it is recognized by the majority of the present FORTRAN 77 compilers. Furthermore, standard Fortran 90 includes this statement.

is translated using a new memory cell associated with the variable NUMBR, although the intention is to print the value of NUMBER. Inconsistent spelling of variable names in this manner is a common error but may not be detected by the compiler.

A variable is said to be **defined** if a value has been placed in it by the program. Although the program output given above shows the value of NUMBR as 0 (zero), the value of any undefined variable is simply the value that happens to be sitting around inside the computer's memory at the time that the program is executed. As a result, no assumption should ever be made as to the value of an undefined variable.

Since it makes no sense to use or print the value of a variable that has never been defined previously in the program, some compilers give a warning message like

```
WARNING - INTEGER VARIABLE (NUMBR) NEVER ASSIGNED A VALUE
```

when such a mistake is made. This feature is useful, but it does not provide a protection against all the potential hazards associated with the use of undeclared variables. For example, if NUMBR appears on the left of an assignment (NUMBR = ...), there will be no warning and the expression value will be incorrectly assigned to NUMBR instead of NUMBER. Furthermore, warning messages are informational only; they do not prevent compilation and execution. Next, modify the program as follows and compile it:

```
      PROGRAM BETTER
C   This program illustrates the use of IMPLICIT NONE
      IMPLICIT NONE
      INTEGER NUMBER
      NUMBER = 27
      PRINT*, 'The number is ', NUMBR
      END
```

This program still contains the original error, i.e. NUMBER has been typed as NUMBR in the PRINT statement. When you attempt to compile this program, however, the compiler will prompt you with a **fatal error** message concerning the undeclared variable NUMBR. You cannot execute this program without correcting the typing error. (If a statement has a syntax error, the compiler cannot translate it and the program cannot be executed.) Next, carry out the necessary correction and compile/run the program.

Note that, in this very simple example, taking a quick look at the program would suffice to discover the error we have made: We have typed NUMBR instead of NUMBER. Consider, however, a program which contains thousands of lines and hundreds of variables. (Such a program size is typical in many practical programming projects.) It would be very difficult to find a typographical error such as this by a mere visual inspection of the program listing. The IMPLICIT NONE statement is therefore very useful when working with long programs containing many variables.

A complete list of all variables appears in the type declaration section of a program if the IMPLICIT NONE statement is used. If the program is to be modified at a later time, the programmer can easily check this list to avoid using variable names that are already used in the program. This helps to prevent a common error which occurs when modifications to a program inadvertently change the values of some variables used elsewhere in the program.

Based on these considerations, it is recommended that the IMPLICIT NONE statement be included in all FORTRAN programs.

1.6 Arithmetic Operations and Expressions

In Example 1.4, we have used the *assignment statement*

```
NUMBER = 27
```

which assigns the value `27` to the integer variable `NUMBER`. The value of a variable (that is, the value stored in the memory cell associated with that variable name) is often determined by one of the following two ways: (1) an arithmetic assignment statement is used, or (2) the value is read in from a data file or entered at a computer terminal during program execution. The second method employs the `READ` statement which is discussed in the next section.

An **arithmetic assignment statement** has the following general form:

Variable Name = *Arithmetic Expression*

The result (i.e. value) of the *Arithmetic Expression* is stored in the memory cell specified by the variable on the left side of the **assignment operator** (=). The previous value of the variable is erased when the expression value is stored. The arithmetic expression can be a single constant or variable, or a computation involving constants, variables, and the **arithmetic operators** listed below:

Arithmetic Operator	Meaning
+	Addition
-	Subtraction
/	Division
*	Multiplication
**	Exponentiation

The operators listed above are **binary operators**, appearing between two **operands**. In an expression like `3+4`, for example, `3` and `4` are the two operands and the plus sign is the binary operator. The plus and the minus signs may also be employed as **unary operators** (operators with a single operand), the minus sign standing for **negation** and the plus sign acting as the **identity operator**. Thus, `-4` is the negative number "minus four," and `+5` is identical to `5`.

FORTRAN does not allow two arithmetic operators to appear next to each other. Thus, the mathematical formula x^{-y} should be written as `X**(-Y)`, and not as `X**-Y`. The following are other examples of valid arithmetic expressions:

Expression	Value
`2*5`	`10`
`1+3`	`4`
`4/2`	`2`
`3**2`	`9`
`6-2+3`	`7`
`N + 2`	Depends on the value of `N`
`A * B`	Depends on the values of `A` and `B`

The meaning of each expression given above is unambiguous, but what do we mean when we write `2+4/2` ? The value of this expression depends on the order of evaluation of the arithmetic operations of addition and division: If division is performed first, then the value of the expression is `2+2`, i.e. `4`. If addition is performed first, the value is `6/2`, i.e. `3`. In order to write complicated expressions correctly, the following **rules of expression evaluation** should be remembered:

1. All subexpressions within parentheses are evaluated first. In the case of nested parenthesized subexpressions, the innermost subexpression is evaluated first.
2. The computer evaluates a parenthesis-free subexpression using the following hierarchy:
   First precedence: `**`
   Second precedence: `/` and `*`
   Last precedence: + and -
3. Operators within the same parenthesis-free subexpression and at the same level of hierarchy (such as `/` and `*`) are evaluated from left to right. The exception to this rule is `A**B**C` which is evaluated from right to left, i.e. as `A**(B**C)`.

Consider the following examples:

`A+B/C`	`B/C` is evaluated first, the value of `B/C` is then added to `A`.
`(A+B)/C`	`A+B` is evaluated first, the result is divided by `C`.
`M*K/J`	`M` is multiplied by `K` first, the result is divided by `J` next.
`M*(K/J)`	`K/J` is calculated first and its value is multiplied by `M`.
`A/B**2`	`B**2` is evaluated first, `A` is then divided by `B**2`.
`(A/B)**2`	`A/B` is calculated first, then its square is computed.

It is clear that inserting parentheses in an expression may change the order of operator evaluation. If you are not sure about the order of evaluation that will be followed by the compiler in a particular case, you can use extra parentheses to clearly specify the order of evaluation you want:

Expression	Alternative Equivalent Expression
`A + B*C - C**2`	`A + (B*C) - (C**2)`
`D**E*F`	`(D**E)*F`
`X + Z / X / Y`	`X + ((Z /X)/Y)`
`A*B**2/(C*D)`	`(A*(B**2))/(C*D)`
`-2.5**3`	`-(2.5**3)`

As you become more experienced in FORTRAN, however, you should try to avoid using redundant parentheses as they tend to make rather simple expressions look unnecessarily complicated. While `(((A)+(B))+((C)-(D)))` is syntactically a perfectly correct FORTRAN expression, it is awkward when compared to the equivalent expression `A+B+C-D`.

So far in our discussion of arithmetic expressions, we have not focused on the types of the variables and constants in these expressions. When both of the operands of an arithmetic operation are of type integer, the arithmetic involved is referred to as **integer arithmetic** and the operation generates an integer value. When the operands are real, then **real arithmetic** is performed to yield a real value. For example, integer arithmetic is performed to evaluate

```
6+7          3-2          3*4
```

yielding integers `13`, `1`, and `12`, respectively. **Integer division** also yields an integer, the integral part of the quotient. Hence, the fractional part of the quotient is truncated in integer division. For example,

`3/2` yields `1` `-3/2` yields `-1` `5/10` yields `0`

It is very important to understand the consequences of using integer division. For example, since division has higher precedence than addition, the expression

```
1/3 + 1/3 + 1/3
```

is equivalent to

```
(1/3) + (1/3) + (1/3)
```

which evaluates to `0` (zero), since `1/3` yields `0`. On the other hand, real arithmetic is used to evaluate

```
6.+7.          3.-2.          3.*4.
```

yielding reals `13.`, `1.` and `12.`, respectively. Real division is similar to ordinary division. Thus

`3./2.` yields `1.5`, `-3./2.` yields `-1.5` `5./10.` yields `0.5`

The examples above illustrate a general rule: *When both operands of a binary operation are of the same type, the result of the operation is also of that type.* When the two operands are of different data types, then we speak of a **mixed-mode of operation**. For example, one operand may be of type real, while the other is of type integer. In such a case, the integer is first converted to real and then real arithmetic is performed to yield a real value. Therefore

`6+7.` and `6.+7` yield the real value `13.0`
`3/2.` and `3./2` yield `1.5`

Consider now the following **mixed-mode expression**:

`13/2 + 2.5` yields the real value `8.5`

In this expression, the integer division `13/2` is carried out first (division has precedence over addition) yielding the integer value `6` which is then converted to the real value `6.0` and added to the real value `2.5`, yielding the real number `8.5`. As this example illustrates, it is possible for one part of an expression to be evaluated using integer arithmetic, followed by another part evaluated using real arithmetic.

The general rule that applies in the evaluation of mixed-mode expressions is the following: *the type of the value returned by an expression is the type of the "highest-ranked" operand.* The ranking of operands is: (1) complex, (2) double precision (3) real (4) integer. For example, if a complex number and a real number are added, the real number is first converted to a complex number, and the result of the addition is also complex:

```
(3.5,4.0) + 5.2
```

is equivalent to

```
(3.5,4.0) + (5.2,0.0)
```

which yields `(8.7,4.0)`. Note that when a real number is converted to a complex number, `0.0` (zero) is appended as the imaginary part.

Mixed-mode of operations involving both a double precision operand and a complex operand are not permitted in standard FORTRAN. If the "double complex" data type is defined on the FORTRAN compiler you are using, then you may use mixed-mode expressions involving both complex numbers and double precision numbers. The evaluation of such an expression, however, constitutes an exception to the above stated rule: both operands are first converted to double complex, and the expression yields a double complex result.

There are a couple of subtle points regarding the exponentiation operation that should be mentioned here. Consider the expression `X**N`, where `X` is real and `N` is integer. This actually is not a mixed-mode expression because the computer multiplies `X` by itself `N` times when this expression is evaluated. That is, this expression is evaluated as `X*X*...*X` where `X` occurs `N` times.

Next, consider the expression `X**Z`, where `Z` is real. This expression will not be evaluated by multiplying `X` by itself; it will probably be evaluated by using the algebraic equality $X^Z = e^{Z(\ln X)}$. This method takes longer to perform and is less accurate than a series of multiplications[18]. Therefore, an integer exponent should be used instead of a real exponent whenever possible. For example, write `X**3` instead of `X**3.0`.

---

[18] Chapman (1998), p.43.

It should also be noted that a negative number cannot be raised to a real power. For example, `(-3.)**2.5` and `(-2.)**3.0` are not valid expressions. Note that raising a negative number to an integer power e.g. `(-2.)**3` is allowed.

We now return to our discussion of *arithmetic assignment statements*. When the two sides of = are of different types, the statement is a **mixed assignment statement**.

It is important to understand that in an assignment statement, two different actions take place: (1) The evaluation of the arithmetic expression on the right side of =. (2) The assignment of the result of this evaluation to the memory location represented by the variable name on the left of =.

The first step is carried out in accordance with the principles discussed above. In the second step, if the value of the expression is an integer and the variable on the left is real, then the expression value is converted to real and then assigned to the real variable. Thus, in the assignment statement

```
A = 2 + 1/2 + 3
```

the expression on the right side of = is first evaluated and the integer value `5` is found. Then, (assuming `A` is of type real) `5` is converted to `5.0` and then assigned to `A`. It should be understood that the expression is evaluated before the assignment is made, and the type of the variable being assigned has no effect whatsoever on the expression value. Consequently, integer arithmetic is employed to evaluate the arithmetic expression `2 + 1/2 + 3` despite the fact that `A` is real.

If the value of the expression is of type real and the variable on the left is of type integer, then the real value of the expression is converted to integer by truncating the fractional part before it is assigned to the integer location. Assuming that `NUM` denotes an integer variable, the assignment statement

```
NUM =  7.0 / 2
```

results in the storage of the value `3` in `NUM`.

Example 1.5:

This program illustrates some of the rules associated with mixed-mode operations and arithmetic assignment statements in FORTRAN. Type and run the program, and examine its output. Make sure that you completely understand the results of the program.

```
      PROGRAM ARITH
C*************************************************************************
C  Program to observe the results of certain mixed-mode
C  arithmetic operations and mixed-mode assignment statements.
C*************************************************************************
      IMPLICIT NONE
```

```
REAL X, Y, Z, Q
INTEGER B, C
B = 5 / 2
C = 5. / 2.
X = 5 / 2
Y = 5. / 2
Z = 5 / 2.
Q = 5. / 2.
PRINT*, 'B and C are integer variables;'
PRINT*, 'X, Y, Z and Q are of type real:'
PRINT*, ' '
PRINT*, 'B = 5 / 2     gives B =', B
PRINT*, 'C = 5. / 2.   gives C =', C
PRINT*, 'X = 5 / 2     gives X =', X
PRINT*, 'Y = 5. / 2    gives Y =', Y
PRINT*, 'Z = 5 / 2.    gives Z =', Z
PRINT*, 'Q = 5. / 2.   gives Q =', Q
END
```

The output should look like the following:

```
B and C are integer variables;
X, Y, Z and Q are of type real:

B = 5 / 2     gives B =           2
C = 5. / 2.   gives C =           2
X = 5 / 2     gives X =    2.00000
Y = 5. / 2    gives Y =    2.50000
Z = 5 / 2.    gives Z =    2.50000
Q = 5. / 2.   gives Q =    2.50000
```

Finally, it should be emphasized that, although the assignment operator (=) looks like the "equals sign" in algebra, it has an entirely different meaning in FORTRAN. Thus, while a statement like

```
I = I + 1
```

would make no sense in algebra, it is a perfectly valid statement in FORTRAN. It means: take the value of the variable named `I`, add `1` to it, and assign (store) the result of the summation in (the memory location represented by) the variable `I`.

1.7 List-Directed `READ` and `PRINT` Statements

We have been using the `PRINT` statement although we have not formally discussed its general syntax and properties. Here is the general form of the list-directed `PRINT` statement:

`PRINT*,` *output list*

where the *output list* is a list of constants, variables, and expressions. Commas separate the items in the *output list.* Each PRINT statement initiates a new line of output. Similarly, the general form of the list-directed READ statement is as follows:

READ*, *input list*

Commas are used to separate the items in the *input list* which is a list of variables.

When a READ statement is reached during execution, the program pauses and waits for data. (It is always helpful to print a **prompting message** just before each READ statement is executed.) You then type in one data value for each variable in the input list. The data items must be separated by one or more blanks, or a comma. After the required data are typed, you press the ENTER (or RETURN) key on your keyboard. Alternatively, you can press ENTER after typing each data item. The program will wait until all items are entered. If the terminating character is a slash (/), however, then no more data items are read. If there are any remaining items in the *input list,* their values will remain unchanged.

Thus, there are four **value separators** in list-directed input: A comma, a blank, a slash, or the end of the line. Any of these value separators may be preceded or followed by any number of consecutive blanks. If there are two consecutive commas, the effect is to leave the corresponding variable in the *input list* unchanged. For example, consider the following program segment:

```
INTEGER K, L, M
K = 1
L = 2
READ*, K, L, M
```

If the user inputs (try this):

```
5,  , /  3
```

K is assigned the value 5, the value of L remains 2, and the input record is terminated with the slash (/) before a value for M is read, so M remains undefined.

The list-directed READ and PRINT statements are sometimes referred to as *unformatted* READ and PRINT statements, respectively. This terminology is used by many authors, because in list-directed output the programmer has no control over the format (the number of blanks between data values, number of digits after the decimal point, etc.) of the output data. Similarly, the exact format of the input data is not specified by the programmer when list-directed input is used. As we shall learn in Chapter 5, however, the term "unformatted" has a different meaning in I/O (input/output) operations, and list-directed I/O is also a type of formatted input/output.

Example 1.6:

The following is a program that calculates and prints out the surface area and volume of a box with dimensions H, L, and W:

```
      PROGRAM BOX
C*************************************************************************
C   Program to calculate the surface area and the volume of a box
C   H       : height
C   W       : width
C   L       : length
C   AREA    : surface area of the box
C   VOLUME  : volume of the box
C*************************************************************************
      IMPLICIT NONE
      REAL H, L, W, AREA, VOLUME
      PRINT*, 'Enter length, width and height: '
      READ*, L, W, H
      AREA = 2 * (L*W + L*H + H*W)
      VOLUME = L*H*W
      PRINT*, 'Surface area =', AREA
      PRINT*, 'Volume       =', VOLUME
      END
```

Note that the *prompting message*

```
Enter length, width and height:
```

tells the user of the program which data values must be entered and the order in which they must be entered. Such a prompting message can be very helpful and minimizes the possibility of making mistakes when entering data. A typical run will look like the following:

```
Enter length, width and height: 1.5  1.0  2.0

Surface area =    13.0000
Volume       =    3.00000
```

Exercises:

6. Develop a program to read the diameter of a circle. Compute the radius, circumference, and area of the circle. Print the results.

7. Develop a program to read the length and the width of a rectangle, and compute and print its perimeter and area.

8. Write a program to read three real numbers into variables A, B, and C, and compute and print their sum, product, and average.

9. Write a program that reads in a temperature value in degrees Celsius, and computes and prints the temperature in degrees Fahrenheit.

10. Write a program to compute the volume of the shell of a hollow ball. The program should read the outside radius of the ball and the thickness of the shell. Hint: The volume of a sphere of radius r is $4\pi r^3/3$.

1.8 Library Functions in FORTRAN[19]

The FORTRAN language has a set of *built-in functions* which can be very helpful in programming. These functions are also referred to as **library functions** or **intrinsic functions**. The mathematical functions of FORTRAN are particularly useful in scientific and engineering computations. A table containing the intrinsic mathematical functions of FORTRAN 77 is given below.

Name	Allowed Argument Types	Description	Function Type
SQRT(X)	Real, complex, double	Square root	Same as argument
EXP(X)	Real, complex, double	Exponential (e^x)	Same as argument
LOG(X)	Real, complex, double	Natural logarithm	Same as argument
LOG10(X)	Real or double	Base-10 logarithm	Same as argument
ABS(X)	Integer, real, or double	Absolute value	Same as argument
ABS(X)	Complex, e.g. X = *(a,b)*	Magnitude $\sqrt{a^2+b^2}$	Real
SIGN(X,Y)	Integer, real, or double	Sign transfer	Same as arguments
MOD(X,Y)	Integer, real, or double	Remainder	Same as arguments
DIM(X,Y)	Integer, real, or double	Positive difference	Same as arguments
MAX(X,Y,...)	Integer, real, or double	Select largest argument	Same as arguments
MIN(X,Y,...)	Integer, real, or double	Select smallest argument	Same as arguments
COS(X)	Real, complex, double	Cosine (X in radians)	Same as argument
SIN(X)	Real, complex, double	Sine (X in radians)	Same as argument
TAN(X)	Real, double	Tangent (X in radians)	Same as argument
ASIN(X)	Real, double	Arc sine	Same as argument
ACOS(X)	Real, double	Arc cosine	Same as argument
ATAN(X)	Real, double	Arc tangent	Same as argument
ATAN2(X,Y)	Real, double	Arc tangent of X/Y	Same as arguments
COSH(X)	Real, double	Hyperbolic cosine	Same as argument
SINH(X)	Real, double	Hyperbolic sine	Same as argument
TANH(X)	Real, double	Hyperbolic tangent	Same as argument
AIMAG(X)	Complex, e.g. X = *(a,b)*	Imaginary part of X, i.e. *b*	Real
CONJG(X)	Complex, e.g. X = *(a,b)*	Conjugate of X, i.e. *(a,-b)*	Complex
DPROD(X,Y)	Real	Double precision product	Double

[19] Only the FORTRAN 77 library functions are presented in this chapter. Fortran 90 introduced a significant number of new library functions that give additional power to the language.

The symbols X and Y in the above table represent the **arguments** of the corresponding functions. In general, the argument of a function may be a single constant, a variable, or an expression. The argument may contain a function reference. For example, ABS(SIN(X)) returns the absolute value of the sine of the variable named X.

Many library functions, like COS and ABS, allow the use of more than one type of argument. Certain functions, like AIMAG and FLOAT (to be discussed later), allow only one type of argument.

The type of the value returned by a mathematical function is normally the same as that of its argument(s). For example, SQRT(X) returns a real value if X is real; it returns a double precision value if X is double precision. Two exceptions are the following: ABS(X) returns a real value when its argument is complex. AIMAG takes only a complex argument, and returns a real value.

The sign transfer function SIGN takes two arguments, say X and Y. The value of SIGN(X,Y) is determined as follows: SIGN(X,Y) = -|X| if Y < 0, and SIGN(X,Y) = |X| if Y ≥ 0[20].

The library function DIM returns the positive difference of its two arguments: DIM(X,Y) = 0 if X ≤ Y, while DIM(X,Y) = X - Y if X > Y. Thus, DIM(10,3) = 7, and DIM(2,4) = 0.

The maximum and minimum functions MAX and MIN take two or more arguments. MAX, for example, finds the maximum value in the argument list. Thus, MAX(10, -2) is 10, whereas MIN(-2.5,-7.0,3.,4.2) is -7.0.

The library function MOD computes the remainder of the division of its first argument by the second. For example, MOD(10,5)=0, MOD(7.,2.)=1.0. The formula used in the calculation is as follows: MOD(X,Y) = X - (INT(X/Y)*Y). The MOD function is often used to determine whether one integer is an exact divisor of another. If integer N is a divisor of M, the value of MOD(M,N) will be zero. The library function INT is described later in this section.

Example 1.7:

The following program illustrates the use of the function SQRT in finding the real roots of a quadratic equation. The coefficients of the equation are entered interactively.

```
      PROGRAM QUAD
C*************************************************************************
C  Program to solve a quadratic equation of the form
C              A*X**2 + B*X + C = 0
C  Only real roots are sought.
C*************************************************************************
```

[20] Sometimes in the text we shall use the equals sign (=) with its usual meaning in algebra, i.e. it will mean "is equal to." Remember, however, that (as the assignment operator) it has a different meaning within a FORTRAN program.

```
IMPLICIT NONE
REAL A, B, C, ROOT1, ROOT2, DISCR
PRINT*, 'Enter a, b, c: '
READ*, A, B, C
DISCR = B**2 - 4.0*A*C
ROOT1 = (-B + SQRT(DISCR))/(2.0*A)
ROOT2 = (-B - SQRT(DISCR))/(2.0*A)
PRINT*, 'First  root =', ROOT1
PRINT*, 'Second root =', ROOT2
END
```

The following shows how the program should execute:

```
Enter a, b, c: 1.  1.  -2.0

First  root =    1.00000
Second root =   -2.00000
```

It may be noted that we could write

```
ROOT1 = (-B + SQRT(B**2 - 4.0*A*C))/(2.0*A)
ROOT2 = (-B - SQRT(B**2 - 4.0*A*C))/(2.0*A)
```

instead of the three lines

```
DISCR = B**2 - 4.0*A*C
ROOT1 = (-B + SQRT(DISCR))/(2.0*A)
ROOT2 = (-B - SQRT(DISCR))/(2.0*A)
```

This latter approach reduces the chances of error because the resulting formulas for `ROOT1` and `ROOT2` are simpler. It may sometimes be convenient to break a complicated expression into subexpressions that are assigned to temporary variables (such as `DISCR`). For example, instead of writing

```
T = (1 + SQRT(A+B)+ABS(D))/(1+SQRT(A+B)+ABS(D)+2*C)
```

you can write the three statements

```
TEMP1 = 1 + SQRT(A+B)+ABS(D)
TEMP2 = TEMP1 + 2*C
T = TEMP1/TEMP2
```

which have the same effect. Note also that, we could write

```
DISCR**(1./2)
```

instead of

```
SQRT(DISCR)
```

However, it is more efficient and more accurate to use the `SQRT` function. Notice that we write `(1./2)`, and not `(1/2)`, because the value of the integer expression `1/2` is zero (not `0.5`) in FORTRAN.

Remark: The program given above (`QUAD`) actually has a few deficiencies: Firstly, it cannot handle a negative discriminant as it has no mechanism to check whether b^2-4ac is nonnegative or not. The execution stops with an error message if the discriminant is negative (try, for example, a = 1, b = 0 and c = 1). Secondly, consider what happens if a = 0: in the calculation of `ROOT1`, the denominator will be zero and (since division by zero is undefined)

this will generate another unpleasant looking error message. We shall learn how to handle these cases as we learn more about FORTRAN (see Example 2.7).

Example 1.8:

The following program reads in the side lengths A, B, and C of a triangle, and computes and prints the area and the perimeter of the triangle. Here S denotes the semi-perimeter, i.e. half the perimeter. The area is equal to the square root of the product S(S-A)(S-B)(S-C).

```
      PROGRAM TRIAN
C*************************************************************************
C   This program computes the perimeter and area of a triangle.
C   A, B, C    :  Lengths of the three sides of the triangle
C   PER        :  Perimeter of the triangle
C   S          :  Semi-perimeter of the triangle
C*************************************************************************
      IMPLICIT NONE
      REAL A, B, C, AREA, PER, S
      PRINT*, 'Enter the side lengths of your triangle: '
      READ*, A, B, C
      PER = A + B + C
      S = PER/2.0
      AREA = SQRT(S*(S-A)*(S-B)*(S-C))
      PRINT*, 'Perimeter =', PER
      PRINT*, 'Area      =', AREA
      END
```

Example 1.9:

The area of a triangle can also be computed using the formula

$$\text{Area} = (1/2)\, a\, b \sin\theta$$

where θ is the angle between the sides whose lengths are a and b. The program presented below reads in the side lengths a, b, and the angle θ (represented by the variable name THETA), and computes and prints the side length c, the area, and the perimeter of the triangle. Note that the program reads θ in degrees and converts it to radians. (Remember that the argument of a FORTRAN trigonometric function has to be in radian measure.) The Law of Cosines is employed to calculate c, i.e. $c^2 = a^2 + b^2 - 2ab\cos\theta$.

```
      PROGRAM TRIAN2
C*************************************************************************
C   Given the lengths of two of its sides and the angle between these sides,
C   this program computes the length of the third side, the perimeter and
C   the area of a triangle.
C   A, B, C    :  Lengths of the three sides of the triangle
C   PER        :  Perimeter of the triangle
C   THETA      :  Angle between A and B
C*************************************************************************
      IMPLICIT NONE
      REAL A, B, C, AREA, PER, THETA
      PRINT*, 'Enter two side lengths of your triangle: '
      READ*, A, B
      PRINT*, 'Enter the angle (in degrees) between these sides: '
      READ*, THETA
C   Convert to radians
      THETA = 3.14159*THETA/180.0
```

```
      AREA = A*B*SIN(THETA)/2.0
C     Use Law of Cosines to calculate side C
      C = SQRT(A**2 + B**2 - 2*A*B*COS(THETA))
      PER = A + B + C
      PRINT*, 'Length of the third side =', C
      PRINT*, 'Perimeter                =', PER
      PRINT*, 'Area                     =', AREA
      END
```

The specific constant 3.14159 is used for π in the above program. This is acceptable, but may not be as accurate as possible. A alternative may be to introduce a variable `PI`, and to set `PI = ACOS(-1.)` at the top of the program. This variable can then be used to represent π in the rest of the program (see Exercise 13).

Exercises:

11. Write a program that prompts the user for the Cartesian coordinates of two points (x_1,y_1,z_1) and (x_2,y_2,z_2) and displays the distance between them:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

12. You are given the coordinates of three points (X_1,Y_1), (X_2,Y_2), (X_3,Y_3) on the X-Y plane. Write a program to read the coordinates. Next calculate and print the distances $DIST_{12}$, $DIST_{13}$, and $DIST_{23}$ between the points. Also calculate and print the area of the triangle formed by the three points:

$$\text{Area} = 0.5\,|X_1Y_2 - X_2Y_1 + X_2Y_3 - X_3Y_2 + X_3Y_1 - X_1Y_3|$$

13. The standard FORTRAN functions like `SQRT` and `ACOS` usually provide results that agree with exact values very well because they have been written carefully. The constant π, for example, can be calculated accurately using the trigonometric functions of FORTRAN. To demonstrate this, type and run the following program:

```
      PROGRAM CALCPI
C*****************************************************************************
C     Program demonstrates calculation of pi = 3.14159....
C*****************************************************************************
      IMPLICIT NONE
      REAL PI
      DOUBLE PRECISION DPI
C     Set PI and DPI equal to the "exact" value of pi
      PI  = 3.14159265358979323846264 3
      DPI = 3.14159265358979323846264 3D0
C     See how PI and DPI are stored in the computer
      PRINT 1, 'PI  =', PI
      PRINT 1, 'DPI =', DPI
C     See how PI can be calculated using library functions
      PRINT 1, 'ACOS(-1.)     =', ACOS(-1.)
      PRINT 1, 'ACOS(-1.D0)   =', ACOS(-1.D0)
      PRINT 1, '4*ATAN(1.)    =', 4*ATAN(1.)
      PRINT 1, '4*ATAN(1.D0)  =', 4*ATAN(1.D0)
    1 FORMAT(1X, A, F35.30)
      END
```

The output should look like the following:

```
PI  =    3.141593000000000000000000000000
```

```
DPI =   3.14159265358979300000000000000000
ACOS(-1.)     =   3.14159300000000000000000000000000
ACOS(-1.D0)   =   3.14159265358979300000000000000000
4*ATAN(1.)    =   3.14159300000000000000000000000000
4*ATAN(1.D0)  =   3.14159265358979300000000000000000
```

The "exact" value[21] of π was taken from *Mathematical Handbook* cited in References. This program uses user-formatted output (via the `FORMAT` statement) to print as many digits as desired after the decimal point. Do not worry about the use of the `FORMAT` statement at this stage (although you can read Section 1.9 to learn about this statement). Notice, however, that the values stored in the variables `PI` and `DPI` are not equal to the "exact" value assigned to them. Consequently, we might as well use the value of `ACOS(-1.)` or `ACOS(-1.D0)` for π in our programs instead of typing a long constant such as `3.14159265358979323846264`3.

In addition to the mathematical functions discussed so far, FORTRAN has a number of other library functions which augment the strength of the language. These are (i) type conversion functions, (ii) functions for truncating and rounding, and (iii) character processing functions. The following is a list of the names and the properties of the type conversion functions.

Name	Allowed Argument Types	Description	Function Type
`INT(X)`	Integer, real, complex, double	Converts `X` to integer	Integer
`REAL(X)`	Integer, real, complex, double	Converts `X` to real	Real
`DBLE(X)`	Integer, real, complex, double	Converts `X` to double precision	Double
`DFLOAT(X)`	Integer, real, complex, double	Converts `X` to double precision	Double
`CMPLX(X)`	Integer, real, double	Result = `(REAL(X),0.)`	Complex
`CMPLX(X)`	Complex	Result = `X`	Complex
`CMPLX(X,Y)`	Integer, real, double	Result = `(REAL(X),REAL(Y))`	Complex
`ICHAR(X)`	A single character	Converts `X` to integer	Integer
`CHAR(X)`	Integer	Converts `X` to character	Character

The type conversion function `INT` takes a single argument and converts it to integer. `INT(X)=X` if `X` is an integer. If `X` is real or double precision, then `INT` truncates `X` to yield an integer. For example, `INT(2.1)=2`, `INT(-3.5D0)=-3`, etc. If `X` is complex, say `X = (a,b)`, then `INT(X)=INT(a)`.

The type conversion function `REAL` takes a single argument and converts it to real. `REAL(X)=X` if `X` is real. If `X` is an integer, then `REAL(X)` appends a decimal point to `X`. Thus, `REAL(2)=2.`, `REAL(-10)=-10.`, etc. If `X` is a double precision number, then `REAL` converts `X` to real by truncation (i.e. `REAL(X)` is approximately the first six significant digits of `X`). If `X` is complex, say `X=(a,b)`, then `REAL(X)=a`.

Example 1.10:

In this book, we shall mostly use integer and real data (and occasionally logical and character data) since these are the most frequently needed data types in programming. After you complete reading this text, however, you should have no difficulty in employing complex and double precision constants and variables if the solution of a problem requires their use. The use of complex variables is illustrated in this example.

Note how the type conversion function CMPLX is utilized in this example: Since A, B, and C are real variables, the value of B**2-4.0*A*C is also real. Inserting this real expression into the SQRT function would lead to an error when it is negative (try it). While the square root of a negative real number is not defined (because the square x^2 of a real number x is always non-negative), the square root of a complex number with a negative real part is well-defined and can be calculated using SQRT. (See Example 2.7 for an alternative way of calculating the complex roots.)

```
      PROGRAM QUAD2
C*************************************************************************
C  Program to solve a quadratic equation of the form
C             A*X**2 + B*X + C = 0
C  where A, B, and C are real. Complex roots are allowed.
C*************************************************************************
      IMPLICIT NONE
      REAL A, B, C
      COMPLEX ROOT1, ROOT2
      PRINT*, 'Enter a, b, c: '
      READ*, A, B, C
      ROOT1 = (-B + SQRT(CMPLX(B**2 - 4.0*A*C))) / (2.0*A)
      ROOT2 = (-B - SQRT(CMPLX(B**2 - 4.0*A*C))) / (2.0*A)
      PRINT*, 'First  root =', ROOT1
      PRINT*, 'Second root =', ROOT2
C   Check the results
      PRINT*, 'A*ROOT1**2 + B*ROOT1 + C =', A*ROOT1**2 + B*ROOT1 + C
      PRINT*, 'A*ROOT2**2 + B*ROOT2 + C =', A*ROOT2**2 + B*ROOT2 + C
      END
```

The roots of $x^2+1 = 0$ are calculated in the following sample run (note that there are no real roots in this case):

```
Enter a, b, c:  1  0  1
First  root = (0.000000,1.00000)
Second root = (0.000000,-1.00000)
A*ROOT1**2 + B*ROOT1 + C = (0.000000,0.000000)
A*ROOT2**2 + B*ROOT2 + C = (0.000000,0.000000)
```

Functions for truncating and rounding are similar to the type conversion functions. They are described in the following table.

[21] Remember that π cannot be represented by a finite number of digits.

Name	Allowed Argument Types	Description	Function Type
AINT(X)	Real, double	Truncate the decimal part	Same as argument
NINT(X)	Real, double	Round to nearest integer	Integer
ANINT(X)	Real, double	Round to nearest whole number	Same as argument

The truncation function AINT is different from the type conversion function INT in that the value AINT returns is real or double precision (i.e. the decimal point is preserved), whereas INT returns an integer value. For example, INT(4.2) is equivalent to 4 (an integer) whereas AINT(4.2) is equal to 4.0 (a real number).

Another function that is useful in some applications is NINT. It takes a real or double precision value as its argument and yields the nearest integer to the argument. For example, NINT(15.9) yields 16 (whereas INT(15.9) is equal to 15), NINT(15.3) yields 15, and NINT(-15.9) is -16, the integer nearest to -15.9.

The function ANINT is similar to NINT, but the type of the value it returns is "same as argument." When X is double precision, ANINT(X) is equal to DBLE(NINT(X)). If X is real, then ANINT(X) is equivalent to REAL(NINT(X)).

The library functions (like SQRT) that allow the use of more than one type of argument are termed **generic intrinsic functions**. Those functions (like FLOAT) that can be used with only one type of argument are named **specific intrinsic functions**.

We have seen that most of the mathematical functions are generic functions. For example, the generic square root function SQRT can take a real, a double precision, or a complex argument. In the older versions of FORTRAN, however, SQRT was a specific function that took only a real argument and returned a real value. To handle a double precision argument, another specific function, namely DSQRT had to be used. Yet another specific function, CSQRT, was employed to calculate the square root of a complex number. Since SQRT can handle all three types of arguments, there remains no need to use the functions DSQRT and CSQRT. Specific functions such as these have been retained in FORTRAN mostly to provide backward compatibility, i.e. to make sure that older programs using the mentioned functions can be compiled using a FORTRAN 77 (or Fortran 90) compiler without modification. A table containing these functions is provided below as a reference. You should employ the generic versions of these functions whenever possible.

Name	Allowed Argument Type	Function Type	Corresponding Generic Function Name
DSQRT(X)	Double	Double	SQRT
CSQRT(X)	Complex	Complex	
DEXP(X)	Double	Double	EXP
CEXP(X)	Complex	Complex	
ALOG(X)	Real	Real	LOG
DLOG(X)	Double	Double	
CLOG(X)	Complex	Complex	
ALOG10(X)	Real	Real	LOG10
DLOG10(X)	Double	Double	
IABS(X)	Integer	Integer	ABS
DABS(X)	Double	Double	
CABS(X)	Complex	Real	
ISIGN(X,Y)	Integer	Integer	SIGN
DSIGN(X,Y)	Double	Double	
AMOD(X,Y)	Real	Real	MOD
DMOD(X,Y)	Double	Double	
IDIM(X,Y)	Integer	Integer	DIM
DDIM(X,Y)	Double	Double	
MAX0(X,Y,...)	Integer	Integer	MAX
MAX1(X,Y,...)	Real	Integer	
AMAX0(X,Y,...)	Integer	Real	
AMAX1(X,Y,...)	Real	Real	
DMAX1(X,Y,...)	Double	Double	
MIN0(X,Y,...)	Integer	Integer	MIN
MIN1(X,Y,...)	Real	Integer	
AMIN0(X,Y,...)	Integer	Real	
AMIN1(X,Y,...)	Real	Real	
DMIN1(X,Y,...)	Double	Double	
DCOS(X)	Double	Double	COS
CCOS(X)	Complex	Complex	
DSIN(X)	Double	Double	SIN
CSIN(X)	Complex	Complex	
DTAN(X)	Double	Double	TAN
DASIN(X)	Double	Double	ASIN
DACOS(X)	Double	Double	ACOS

DATAN(X)	Double	Double	ATAN
DATAN2(X,Y)	Double	Double	ATAN2
DCOSH(X)	Double	Double	COSH
DSINH(X)	Double	Double	SINH
DTANH(X)	Double	Double	TANH

In addition to the generic type conversion, rounding, and truncating functions discussed earlier, there are a number specific functions that have been inherited from earlier versions of FORTRAN. These are shown in the following table.

Name	Allowed Argument Type	Function Type	Corresponding Generic Function Name
IFIX(X)	Real	Integer	INT
IDINT(X)	Double	Integer	
FLOAT(X)	Integer	Real	REAL
SNGL(X)	Double	Real	
DINT(X)	Double	Double	AINT
IDNINT(X)	Double	Integer	NINT
DNINT(X)	Double	Double	ANINT

IFIX and IDINT convert real and double precision arguments, respectively, to integers. Since INT can handle both of these data types, it is more general and can be used exclusively.

The library functions FLOAT and SNGL convert integer and double precision numbers, respectively, to reals. When X is double precision, the function references SNGL(X) and REAL(X) are interchangeable. Similarly, when X an integer, FLOAT(X) and REAL(X) are equivalent. The function REAL can be used exclusively to handle all four types of arguments.

The function DNINT is similar to ANINT, but it takes a double precision argument only, and returns a double precision value. Thus, DNINT(X) is equal to DBLE(NINT(X)). If X is double precision, then ANINT(X) is also equal to DBLE(NINT(X)). Recall that ANINT(X) is equivalent to REAL(NINT(X)) when X is real.

What has been said above should not be construed to imply that all specific intrinsic functions in FORTRAN are "redundant." Functions such as CHAR, ICHAR, DPROD, and the character functions are necessarily specific because of the tasks

they carry out, and therefore there are no generic functions that can be used instead of them.

Character functions are those library functions that operate on character constants and variables. Character functions will not be discussed in this chapter (see Chapter 5). For completeness and for future reference, however, a list and brief descriptions of these functions are given in the next table. Note that LGE, LGT, LLE, and LLT are functions of type logical, and as such the values they return are either .TRUE. or .FALSE. (see Chapter 2 for a description of the logical data type).

Name	Argument Type	Description	Function Type
LEN(ch)	Character	Length of string	Integer
INDEX(ch1,ch2)	Character	Position of substring ch2 within string ch1	Integer
LGE(ch1,ch2)	Character	ch1 ≥ ch2	Logical
LGT(ch1,ch2)	Character	ch1 > ch2	Logical
LLE(ch1,ch2)	Character	ch1 ≤ ch2	Logical
LLT(ch1,ch2)	Character	ch1 < ch2	Logical

Example 1.11:

The functions FLOAT and IFIX are *type conversion functions*. FLOAT takes an integer as its argument and converts it to type real. For example, FLOAT(4) yields the real number 4.0. Similarly, IFIX converts a real number to an integer. In this case, the fractional part of the real number is truncated, e.g. IFIX(5.6) yields 5, an integer value.

The functions REAL and INT can be used instead of FLOAT and IFIX, respectively. If the argument type is real, there is no difference (except in name) between the functions INT and IFIX. Similarly, the functions REAL and FLOAT are equivalent when the argument is an integer. The following program illustrates the use of these and other type conversion functions.

```
      PROGRAM CONV
C   This program demonstrates the FORTRAN type conversion functions.
      IMPLICIT NONE
      INTEGER NUM
      REAL A
      DOUBLE PRECISION DA, DPI
      COMPLEX C
C   Assign Values to A, NUM, C and DPI
      A = 3.0
      NUM = 3
      DPI = ACOS(-1.0D0)
      C = (1.0,-2.5)
C
      PRINT*, 'NUM           =', NUM
      PRINT*, 'REAL(NUM)     =', REAL(NUM)
      PRINT*, 'FLOAT(NUM)    =', FLOAT(NUM)
      PRINT*, 'DBLE(NUM)     =', DBLE(NUM)
```

```
      PRINT*, 'DFLOAT(NUM)   =', DFLOAT(NUM)
      PRINT*, 'CMPLX(NUM)    =', CMPLX(NUM)
      PRINT*, ' '
      PRINT*, 'A             =', A
      PRINT*, 'INT(A)        =', INT(A)
      PRINT*, 'IFIX(A)       =', IFIX(A)
      PRINT*, 'DBLE(A)       =', DBLE(A)
      PRINT*, 'DFLOAT(A)     =', DFLOAT(A)
      PRINT*, 'CMPLX(A)      =', CMPLX(A)
      PRINT*, ' '
      PRINT*, 'DPI           =', DPI
      PRINT*, 'INT(DPI)      =', INT(DPI)
      PRINT*, 'IDINT(DPI)    =', IDINT(DPI)
      PRINT*, 'REAL(DPI)     =', REAL(DPI)
      PRINT*, 'SNGL(DPI)     =', SNGL(DPI)
      PRINT*, 'CMPLX(DPI)    =', CMPLX(DPI)
      PRINT*, ' '
      PRINT*, 'NUM/2         =', NUM/2
      PRINT*, 'REAL(NUM)/2   =', REAL(NUM)/2
      PRINT*, 'FLOAT(NUM)/2  =', FLOAT(NUM)/2
      PRINT*, ' '
      PRINT*, 'A/2           =', A/2
      PRINT*, 'INT(A)/2      =', INT(A)/2
      PRINT*, 'IFIX(A)/2     =', IFIX(A)/2
      PRINT*, ' '
      PRINT*, 'C             =', C
      PRINT*, 'INT(C)        =', INT(C)
      PRINT*, 'REAL(C)       =', REAL(C)
      PRINT*, 'DBLE(C)       =', DBLE(C)
      PRINT*, 'DFLOAT(C)     =', DFLOAT(C)
      END
```

The output of the program should look like the following:

```
NUM           =           3
REAL(NUM)     =    3.00000
FLOAT(NUM)    =    3.00000
DBLE(NUM)     =    3.00000000000000
DFLOAT(NUM)   =    3.00000000000000
CMPLX(NUM)    = (3.00000,0.000000)

A             =    3.00000
INT(A)        =           3
IFIX(A)       =           3
DBLE(A)       =    3.00000000000000
DFLOAT(A)     =    3.00000000000000
CMPLX(A)      = (3.00000,0.000000)

DPI           =    3.14159265358979
INT(DPI)      =           3
IDINT(DPI)    =           3
REAL(DPI)     =    3.14159
SNGL(DPI)     =    3.14159
CMPLX(DPI)    = (3.14159,0.000000)

NUM/2         =           1
REAL(NUM)/2   =    1.50000
FLOAT(NUM)/2  =    1.50000
```

```
A/2             =    1.50000
INT(A)/2        =           1
IFIX(A)/2       =           1

C               = (1.00000,-2.50000)
INT(C)          =           1
REAL(C)         =    1.00000
DBLE(C)         =    1.00000000000000
DFLOAT(C)       =    1.00000000000000
```

Exercises:

15. Write a program to demonstrate the rounding and truncating functions. The output should be similar to the following:

```
AINT(3.7)       =    3.00000
AINT(3.7D0)     =    3.00000000000000
DINT(3.7D0)     =    3.00000000000000
ANINT(3.7)      =    4.00000
ANINT(3.7D0)    =    4.00000000000000
DNINT(3.7D0)    =    4.00000000000000
NINT(3.7)       =           4
NINT(3.7D0)     =           4
IDNINT(3.7D0)   =           4
NINT(-3.7)      =          -4
NINT(0.0)       =           0
```

16. Develop and run several test programs to better understand the FORTRAN library functions discussed in this chapter. Also consult the reference manual or the on-line help text of the particular compiler you are using to learn about the other library functions (i.e. functions not part of standard FORTRAN) provided by your compiler.

17. Write a program that will read a real value `X` and round it to the nearest two decimal places. Hint: Use the `NINT` function.

1.9 Output Editing

So far in our programs we have been using the list-directed `PRINT` statement

`PRINT*,` *output list*

where the *output list* is a list of constants, variables, and expressions. This method of writing output is referred to as **list-directed output**, which means that the type of an output list item determines the form of the value printed.

You may have noticed that, with this printing method, the programmer has limited control over the appearance of the output. For example, the displayed number of digits after the decimal point of a real number and the horizontal spaces between data values printed on the same line are automatically determined by the compiler. Actually, it is possible for the programmer to indicate the exact appearance of the displayed/printed data. This is accomplished through the use of **user-formatted output** or **output editing**.

The exact form of a line of output (or, several lines of output) is described by using a **format specification**. A format specification consists of a list of **edit**

descriptors enclosed in parentheses. The most frequently used edit descriptors are listed and described in the following table.

Edit Descriptor	Description
Iw	For printing an integer value in a field of width w (which includes a sign if it is negative). The printed number is right justified.
Fw.d	For printing a real or double precision value in a field of width w. The number is rounded to d digits after the decimal point. The field width w includes a sign if it is negative, the decimal point, and d. You must set w≥d+2 for negative numbers and w≥d+1 for positive numbers. The printed number is right justified.
Ew.d	For printing a real or double precision value in the exponential form. The number is rounded to d digits after the decimal point. The field width w includes a sign if it is negative, (possibly) a zero to the left of the decimal point, the decimal point, the letter E, the width of the exponent, and d. Use w≥d+6 for positive values and w≥d+7 for negative values. The printed number is right justified.
A	For printing a character value. The field width is automatically determined by using the length of the character value.
Aw	For printing a character value in a field of width w. The printed string is right justified, i.e. the string is padded with blanks on the left if it contains fewer than w characters. If the string consists of more than w characters, the extra characters on the right are not printed.
nX	For skipping n horizontal spaces (i.e. n blanks are printed).
/ (slash)	For passing to the next line. Use n(/) to have n-1 blank lines between two output lines, or to place n blank lines at the beginning or the end of output.
Tn	To advance to position n of the output line before printing the next item.
TLn TRn	To print the next output item starting n positions before (TL) or after (TR) the current position.
Lw	To print w-1 blanks, followed by T or F to represent a logical value.
:	To terminate format if there are no more output items.

In this table, n, w, and d represent integer constants; parameters (introduced in Chapter 3) may not be used. A format specification has the following general form:

$$(desc_1\ ,\ desc_2\ ,\ldots).$$

where $desc_1$, $desc_2$,... are edit descriptors. A format specification is used in conjunction with a FORMAT statement as follows:

```
n FORMAT(desc1, desc2,...).
```

where *n* is a unique statement label. This label is used (instead of *) in any PRINT statement that will employ the output format specified in the FORMAT statement:

PRINT *n, output list*

A FORMAT statement can be placed anywhere in the program between the PROGRAM and END statements, and more than one PRINT statement can reference the same FORMAT statement.

An alternative approach is to write the format specification as a character string (enclosed in apostrophes) and place it directly in the PRINT statement:

PRINT '(*desc*$_1$, *desc*$_2$,...)', *output list*

Note that the format specification is separated by a comma from *output list.*

The quickest way to understanding how edit descriptors and format specifications are used is to look at specific examples. Consider two integer variables N and M, and a real variable A, which have the values 200, 35, and 3.6789, respectively. Then, the output of the program segment

```
      PRINT 5, N, M, A
    5 FORMAT(I5, 2X, I5, 4X, F5.2)
```

will be

□200□□□□□35□□□□□3.68

where each □ stands for one blank character. Note that each number is printed right-justified within its specified field. Note also that the first column of the output line is not printed: although the field width for the first data item is 5 (the edit descriptor used for printing N is I5), the effective field width is 4 (one blank plus the number 200 appear within this field). As a general rule, you should avoid using column 1, since anything printed there will not be displayed. The edit descriptor 2X causes two blank characters to be printed after the value of N. The value of M is printed using the next edit descriptor, I5. Four blank spaces are displayed by 4X. Because of the edit descriptor F5.2 used for printing A, its value 3.6789 is rounded (not truncated) to two digits after the decimal point. The value of A in the memory, however, is not in any way affected (and remains equal to 3.6789).

As noted above, an integer value is printed right-justified in its field. If there are insufficient columns to display the integer and its sign (for example, if I3 is used to print 1000), the value displayed depends on the compiler. Usually, a string of asterisks will be displayed (try it). Similarly, if an insufficient number of columns is specified when printing a real value using the edit descriptor F, many compilers display a string of asterisks. If insufficient columns are specified (when using the edit

descriptor `Aw`) to print a character string, then only part of the string that fits the output field will be printed (see Exercise 21).

Consider next printing a character string, say `'TEXT'`. To that end, the edit descriptor `A` can be used:

```
      PRINT 30, 'TEXT'
   30 FORMAT(1X,A)
```

The descriptor `1X` is needed to skip the first column. An alternative is to use the following approach (try both approaches and observe the output):

```
      PRINT 30, ' TEXT'
   30 FORMAT(A)
```

Note the leading blank in `' TEXT'` above. Yet another valid approach is as follows:

```
      PRINT 30, 'TEXT'
   30 FORMAT(A5)
```

Instead of using a `FORMAT` statement, you may write the format specification as a character string and place it in the `PRINT` statement:

```
      PRINT '(A5)', 'TEXT'
```

This approach may be particularly convenient when the format specification is short.

A character string can also be placed within a format specification. This is termed **apostrophe editing**. The following is a simple example (try it):

```
      PRINT 20
      PRINT '(1X, ''Second.'')'
   20 FORMAT(1X, 'First.')
      END
```

Note the two consecutive apostrophes placed on each side of the string `'Second.'`. These are needed because the format specification itself is a string enclosed within apostrophes. (Recall that two consecutive apostrophes must be used to represent a single apostrophe within a character string.) An additional level of nested apostrophes would require twice as many apostrophes as the previous level to make the apostrophes' meaning clear. For example:

```
      PRINT '(1X, ''Elif''''s Cooking House.'')'
```

The output of the program segment

```
      PRINT 31, 'TEXT', 'TEXT', 'TEXT'
   31 FORMAT(A5, 2X, A10, 2X, A2)
```

will be

`TEXT□□□□□□□□TEXT□□TE`

Assume that five real numbers are to be printed on a line using the same edit descriptor, say `F8.3`. This can be done by repeating the edit descriptor as many times as required:

```
      PRINT 7, A, B, C, D, E
    7 FORMAT(1X, F8.3, 2X, F8.3, 2X, F8.3, 2X, F8.3, 2X, F8.3)
```

An alternative and more efficient approach is indicated in the following:

```
      PRINT 7, A, B, C, D, E
    7 FORMAT(1X, 4(F8.3, 2X), F8.3)
```

The `E` and `F` edit descriptors can be used to print double precision values. Alternatively, the edit descriptor `Dw.d` (which is similar to `Ew.d`) can be employed.

Each edit descriptor must be associated with an output list item of the correct data type. For example, the edit descriptor `Iw` should be used only to print integer data values.

When a slash, /, is encountered in the middle of a format specification, the current line is terminated and a new line is started. To terminate the current line, and then to skip a line (i.e. to display a blank line) before the next line of output, two consecutive slashes, //, can be used. Multiple slashes can be indicated using the form `n(/)`, where n is a positive integer constant. If the slashes are in the middle of a format specification, the number of blank lines displayed will be one less than the number of consecutive slashes. If the slashes are placed at the beginning or at the end of a format specification, then the number of blank lines printed will be equal to the number of consecutive slashes. Commas are not necessary to separate consecutive slashes or to separate slashes from other edit descriptors.

The last edit descriptor shown in the table is the colon (`:`) edit descriptor. It terminates format control if there are no more items in the *output list*. This feature is used to suppress output when some of the edit descriptors in the format specification do not have corresponding data in the *output list*. See Exercise 26.

Exercises:

18. Write a program to print the integer constant `325` via user-formatted output. Try the following edit descriptors: `I2, I3, I4, I5, I6`. Next, employ the same edit descriptors to print `-325`. (Try both tasks with and without using the descriptor `1X` to skip the first column.)

19. Write a program to print the real constant `3.14159` via user-formatted output. Try the following edit descriptors: `F5.2, F5.1, F6.3, F4.0, F3.2`. (Do not forget to use the descriptor `1X` to skip the first column.) Examine the output carefully.

20. Write a program to print the real constant -4.576 via user-formatted output. Try the following edit descriptors: F4.1, F6.2, F4.0, F9.5, F7.5. Make sure that you understand the output.

21. Write a program to print the character constant 'DESK' using the following edit descriptors: A, A1, A2, A3, A4, A5. (What happens if you do not use the descriptor 1X to skip the first column? Try it.)

22. A string value is printed right-justified in its output field. When the string is stored in a character variable first and then the variable is displayed, however, the situation is a little different. For example, if STATUS is of type CHARACTER*5, the assignment STATUS = 'OK' stores the string OK followed by three blanks in variable STATUS. Consequently, when the value of STATUS is printed using format specification A or A5, FORTRAN displays the string OK followed by three blanks. If STATUS is printed using format specification A10, the value of STATUS is displayed right-justified, so that five blanks before the string OK and three blanks after OK are seen on the screen. To observe these points, type and run the following program:

```
      PROGRAM CHARV
      IMPLICIT NONE
      CHARACTER*5 STATUS
      PRINT 1, 'OK'
      STATUS = 'OK'
      PRINT 1, STATUS
      PRINT 2, STATUS
    1 FORMAT(1X, A5)
    2 FORMAT(1X, A10)
      END
```

23. To understand the use of the slash descriptor better, type and run the following program:

```
      PROGRAM SLASH
      PRINT 1, 'Demonstration of the Slash Descriptor:',
     1     'You can use two consecutive slashes two skip a line:',
     2     'Use two slashes at the end of format to skip two lines:'
    1 FORMAT(1X, A, /, 2(1X, A, //))
      PRINT 2, 'This line printed using FORMAT statement with label 2.'
    2 FORMAT(1X, A, /)
      PRINT 3,
     1  'One slash at end of FORMAT st.2 resulted in one blank line.',
     2  'You don''t have to use commas around slashes for separation.',
     3  'To observe this you can inspect FORMAT statement labeled 3.'
    3 FORMAT(1X, A / 1X, A / 1X, A)
      END
```

Note also that the characters 1, 2, and 3 used as continuation marks (typed in column 6) are not in any way confused with the statement labels 1, 2, and 3 (typed in columns 1 to 5). Notice the use of the edit descriptor 1X to skip the first column of each new output line. What happens if you remove the descriptor 1X? (Try it.)

24. Care should be exercised when using the T edit descriptor. Specifically, position n of the output line is normally position (n-1) of the display screen. This is because position 1 of the output line is not displayed. (A **line-control character** is present at the first position. This will be explained in Chapter 6.) Thus, the descriptor T10 will cause printing at column 9 of the display, not column 10. Type and run the following program as an exercise:

```
      PROGRAM TAB1
      PRINT 10, 'Demonstration of the T descriptor:',
     1  'T2 causes a start at position 1 of the display screen.',
     2  'T3 advances to position 3 of output line (position 1',
     3  'is not printed, so) this is position 2 of the display.'
    10 FORMAT(1X, A, /, T2, A, /, T3, A, /, 2X, A)
      END
```

25. It was pointed out that a format specification can also be written as a character string and placed in a PRINT statement. Another alternative is to use a character variable of sufficient length to store the format specification. To observe this, type and run the following program. Note that the three PRINT statements have equivalent effects.

```
      PROGRAM FORMTS
      IMPLICIT NONE
      CHARACTER*20 FORM1
      INTEGER NUM1
      REAL X
      NUM1=5
      X=ACOS(-1.)
      FORM1='(1X,I3,2X,F8.5)'
      PRINT FORM1, NUM1, X
      PRINT '(1X,I3,2X,F8.5)', NUM1, X
      PRINT 1, NUM1, X
    1 FORMAT(1X,I3,2X,F8.5)
      END
```

26. Type and run the following program. Next, remove the colon from the format specification and compile/run the program again. What difference do you observe?

```
      PROGRAM COLON
      IMPLICIT NONE
      INTEGER M, N
      M = 10
      N = 20
      PRINT 1, M
      PRINT 1, M, N
    1 FORMAT(1X,'M=',I3,:,2X,'N=',I3)
      END
```

The use of a character variable or a FORMAT statement may be advantageous if (i) the format specification is so long that placing it in a PRINT statement will not be convenient, or (ii) the same format specification will be used by more than one PRINT statement. While a FORMAT statement can be placed anywhere in the program, a character variable containing a format specification must be defined (i.e. its value must be assigned) before it is used in a PRINT statement. It is common practice to place all FORMAT statements together at the bottom or top of a program. Alternatively, each FORMAT statement can be placed immediately after the PRINT statement that references it.

1.10 Programmer-Defined Functions

All of the programs we have studied so far consisted of a single part, i.e. a **main program**. A main program starts with a PROGRAM statement and ends with the END statement. There is another type of **program unit** in FORTRAN, namely a **subprogram**. The two main types of subprogram in FORTRAN are **subroutines** and **functions**. These two types of subprogram are also referred to as **procedures**. While we shall study both of these subprogram types in great detail later (see Chapter 4), an introduction to function subprograms is presented in this section.

We have already used several function subprograms, i.e. intrinsic functions such as SQRT, MOD, COS, etc. These special subprograms are written by compiler manufacturers because they are needed very frequently by FORTRAN programmers. An advantage of library functions is that they are very well written, i.e. they are efficient, accurate, and robust.

Often, however, the functions available in the function library are not sufficient for the solution of a particular programming problem. FORTRAN makes it possible for programmers to develop and use their own functions to solve such problems. Programmer-defined functions are called **external functions** (as opposed to the *intrinsic functions* of the language). The following is the most typical form of an external function definition:

```
functype FUNCTION funcname(dummy argument list)
type declarations for dummy arguments
type declarations for local variables
   function body
   funcname = expression
RETURN
END
```

The *functype* specifies the type (INTEGER, REAL, LOGICAL, etc.) of the result returned by the function. The function name *funcname* must obey the usual rules for variable names (i.e., it must consist of one to six alphanumeric characters with the first character being a letter). At least one statement assigning a value to *funcname* must be included in the body of the function. The *dummy argument list* is a list of symbolic names: constants or expressions cannot appear in this list. The items that constitute the *dummy argument list* are separated by commas and enclosed in parentheses. The RETURN statement transfers control back to the program unit that has invoked the function. The function may be invoked in the main program and/or in another subprogram (a subroutine or a function).

In a subprogram the END statement has the same effect as a RETURN statement; therefore the RETURN statement can actually be omitted if it is immediately followed by the END statement. A RETURN statement is necessary to exit the function (and return to the program that invoked the function) before reaching the END statement. We need, however, to learn more about FORTRAN (specifically, the decision structures discussed in Chapter 2 must be studied) before this can be exemplified.

A complete program may consist of a number of different program units. Exactly one of these program units must be a main program unit. On the other hand, there may be several subprogram units. Execution of the complete program will start at the beginning of the main program unit.

The subprograms can be placed within the same file that contains the main program unit. Alternatively, a subprogram may exist in a separate source file. In the latter case, a separate object file corresponding to each source file is generated first. These object files are then **linked** to form a single executable file.

Several example function subprograms are presented in this and the next chapter. It is recommended that you read the above paragraphs again after studying these examples.

Example 1.12:

By the definition of a radian, 2π radians = 360°, i.e. π radians = 180°. Therefore, given the value of an angle in radians, we have

$$\text{(angle in degrees)} = 180\text{(angle in radians)}/\pi$$

The function DEGREE given below employs this formula to carry out the desired conversion.

```
      PROGRAM CONV
C   Driver routine to test the function DEGREE
      IMPLICIT NONE
      REAL DEGREE
      REAL ANGLE
      PRINT*,'Enter angle in radians: '
      READ*, ANGLE
      PRINT*, ANGLE, ' radians = ', DEGREE(ANGLE), ' degrees.'
      END
C
      REAL FUNCTION DEGREE(THETA)
C*********************************************************************
C   Function to convert radians to degrees
C   THETA: Angle in radians (input to the function)
C*********************************************************************
      IMPLICIT NONE
      REAL THETA
      REAL PI
      PI = ACOS(-1.)
      DEGREE = 180.*THETA/PI
      RETURN
      END
```

The main program CONV reads in the angle in radians, stores this value in the real variable ANGLE, and then invokes DEGREE to compute the angle in degrees.

When DEGREE is invoked by the main program, the value that ANGLE has at that instant is substituted for THETA (ANGLE is the actual argument, THETA is the corresponding dummy argument: see the relevant discussion later in this section), and control is transferred to the function.

Next, the instructions within the function are executed. Once all the steps in the body of the function are completed, the RETURN statement returns control back to the main program. Execution is then terminated in the main program by the END statement. (Execution would have continued in the main program if there had been other executable statements between the PRINT statement and the END statement.) Here is a sample run of CONV:

```
Enter angle in radians: 3.1415926
    3.14159      radians =      180.000      degrees.
```

The main program CONV was written with the sole purpose of testing the function DEGREE and demonstrating its use. Now that DEGREE has been tested, it can be used in other programs. The following are some examples of valid uses of DEGREE:

```
PRINT*,'Cosine of', DEGREE(3.14159), ' degrees is', COS(3.14159)
GAMMA = DEGREE(3.14159/2. + 0.1)
BETA = DEGREE(ASIN(1.))
ALFA = DEGREE(A + 2*B)
```

where ALFA, BETA, GAMMA, A, and B are real variables. In the first example, the actual argument is a constant. In the second and fourth cases, the actual arguments are expressions. In third case, we have a function reference as the actual argument. When an expression or a function reference appears as an actual argument, its value is first evaluated, and then that value is passed to the subprogram.

When a subprogram is invoked, the constants, variable names, function references, and the expressions that appear in the argument list are termed **actual arguments**. In Example 1.12, DEGREE is invoked in the main program within the PRINT statement

```
PRINT*, ANGLE, ' radians = ', DEGREE(ANGLE), ' degrees.'
```

Here ANGLE is referred to as an actual argument, implying that it is expected to have a certain value at this point in the program unit that is invoking the function. On the other hand, the names in the argument list of a subprogram definition are called **dummy arguments**. Thus, in the function definition

```
REAL FUNCTION DEGREE(THETA)
...
END
```

the symbolic name THETA is a dummy argument, meaning that it does not represent a location in memory. An actual numerical value is assigned to THETA only after the function is invoked in the main program.

Note that the name of the actual argument (ANGLE) is different from the name of the dummy argument (THETA) used in the function definition: this is completely

permissible. *Correspondence between a dummy argument and an actual argument is established via the positions of the arguments, and not the names of the arguments.* Thus, the first actual argument in a function reference corresponds to the first dummy argument in the function definition, the second actual argument corresponds to the second dummy argument, etc.

Furthermore, *there is name independence between programs.* That is, the variable *names* used in a subprogram have no relation to the *names* in the main program and other subprograms. For example, `PI` is declared as a real variable in `FUNCTION DEGREE`. This declaration is *local* to the function and is not visible outside the function definition. We could declare and use `PI` inside the main program, say, as an integer variable representing an entirely different quantity.

Notice also that the type of the function is declared in the main program with the statement

```
REAL DEGREE
```

This is necessary because of the `IMPLICIT NONE` statement used in the main program: When the `IMPLICIT NONE` statement is used in a program unit, the types of all functions used by that unit must be declared explicitly within that unit. Furthermore, the `IMPLICIT NONE` statement placed in a program unit does not have any effect outside the body of that unit. Therefore, a separate `IMPLICIT NONE` statement must be included within each subprogram and the main program.

Example 1.13:

While the square root of a number can be calculated using the `SQRT` function, a similar library function for the calculation of cubic root does not exist in standard FORTRAN. We can, however, define our own cubic root function as shown below.

```
      REAL FUNCTION CBRT(X)
C*********************************************************************
C   Calculates cube root of positive real number X
C*********************************************************************
      IMPLICIT NONE
      REAL X
      CBRT = EXP(LOG(X)/3.)
      RETURN
      END
```

The function makes use of the following mathematical properties of the exponential and logarithmic functions:

$$b \ln A = \ln A^b \quad \text{and} \quad e^{\ln C} = C$$

Note that `CBRT` takes a real number as input, i.e. it cannot be used to compute the cubic root of another type of number. The following main program illustrates how `CBRT` can be employed (type and run it):

```
      PROGRAM CBROOT
C   Driver routine to test CBRT
      IMPLICIT NONE
      REAL Y, CBRT
      PRINT*,'Enter a real number: '
      READ*, Y
      PRINT*, 'Cube root of', Y, ' is', CBRT(Y)
      PRINT*, 'Compare (number)**(1./3) =', Y**(1./3)
      END
```

What happens if you attempt to calculate the square root of an integer, a double precision, or a complex number using CBRT? (Try it.)

Remark: The main program CBROOT and the function subprogram CBRT can be placed within the same file. They can also be typed into different files. Each file (i.e. the source code in each file) should be separately compiled in the latter case. The object files generated in this manner must then be *linked* to build a single executable file. You should learn how to carry out these steps with the FORTRAN system you are using.

Example 1.14:

To calculate the logarithm of a number to base *b*, the following formula can be utilized:

$$\log_b X = \frac{\log_{10} X}{\log_{10} b}$$

FUNCTION LOGB given below implements this formula:

```
      REAL FUNCTION LOGB(B,X)
C*************************************************************************
C   Function calculates logarithm of X to base B
C*************************************************************************
      IMPLICIT NONE
      REAL B, X
      LOGB = LOG10(X)/LOG10(B)
      RETURN
      END
```

To test and demonstrate this function, type and run the following main program:

```
      PROGRAM LOGART
C   Driver routine to test and demonstrate LOGB
      IMPLICIT NONE
      REAL LOGB, B, X, RESULT
      PRINT*, 'Enter base B (a positive number): '
      READ*, B
      PRINT*, 'Enter X: '
      READ*, X
      RESULT = LOGB(B,X)
      PRINT*, 'log of X w.r.to base B is', RESULT
      END
```

What happens if you type a non-positive number (0 or a negative number) as input?

Exercise:

27. Write a function that can be used to convert angle in degrees to radians. Write a main program to test the function.