Introduction to Computer Programming Using FORTRAN 77

- Al-Dhaher, K.
- Al-Muhtaseb, H.
- Yazdani, J.
- Garout, Y.
- Nazzal, A.
- Zeidan, Y
- Lafi, A.
- Saeed, M.

August 1995 Second Edition

Information and Computer Science Department
College of Computer Sciences and Engineering
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

CONTENTS

1 INTRODUCTION

ICS101 is an introductory course on computer programming. The goal of this course is to teach students the use of computers as tools to solve engineering and scientific problems.

We use many tools in our daily life, from simple things like pens and screwdrivers, to complicated things like watches, radios and TV remote controls, to more complicated things like calculators, television sets, video cameras and cars. More recently computers have been emerging as tools that are used in everyday life. Just as any other tool we should know how to use them properly. It would also be useful, though not necessary, to know how they work and what affects their behavior. If we know for example the structure of a compass, and that it uses a magnet as one of its components, and we know that magnetic fields affect each other, we can understand the behavior of the compass if another magnet is placed beside it.

Knowing how to use a tool or device involves knowing what it can do for us, how we should express what we want to do with the device (e.g. by pressing a key on a calculator, or turning the knob of a radio) and how to receive and interpret the result from this device (e.g. read the sum of the numbers from the calculator display, or listen to the sound from the radio speaker).

Computers vary in size, shape and function. There are small computers and big computers. Large computers are referred to as mainframes. Smaller computers are classified either as minicomputers or microcomputers. Some are used for a specific task, others are general purpose. This variation is similar to the variation in many other devices and tools. There are different screwdrivers, radios and cars. The proper tool for the task should be used. A truck should be used to carry heavy machinery, while a car would be used to carry people (and not the other way around).

A mainframe computer is a powerful machine that can serve hundreds of users that work on it through terminals scattered around and connected to the mainframe through a computer network. The terminals are used by computer users to enter data, write programs and see their results. All the computing is done by the mainframe.

There are other kinds of computers. Personal computers are getting more popular. These are computers that are mainly used by a single person at a time. They have attachments or devices for entering the data and programs, reading the results, as well as performing the actual computing. When you want to use a computer, big or small, you should at least know:

- what the computer can do for you (it might also be useful to know what it cannot do for you);

- the problem you want to solve, and understand it well;
- how to solve the problem;
- how to express the solution to the computer (what you want it to do for you); and
- how to receive and interpret the results.

Remember that the computer is a tool, just like a car for example. If you want to get somewhere, but you do not know where that place is, or how to get there, the car is useless. You have to know how to drive the car, in addition to knowing how to get to your destination from wherever you are. In the remainder of this introductory chapter, we will briefly describe the basic components of computers, and how to interact with them.

1.1 Computer System Components

We can think of computers as devices or machines that are capable of performing certain tasks. A very simple task, for example, is addition. Different computers might have different abilities, but in general, they have a similar internal structure. A typical computer should have input devices to receive input from the user, output devices to enable users to observe and interpret the results, a central processing unit to enable it to perform the needed operations and tasks, and memory to store all the data and programs it needs. An example of an input device is a keyboard or a mouse, an output device can be a video screen or a printer. The physical devices that make up the computer are called "*Hardware*".

1.2 Programs & Programming Languages

Arabic, English, French and other languages, are called natural languages. They are languages used by people to communicate with each other. To communicate correctly, people have to agree on a common language. If you go to Japan and start speaking in Arabic, even if you say simple things like "What time is it?", people will not understand what you are saying. A common language that is understood by both parties has to be used.

Even though there are grammar rules to control the language (what is linguistically correct and what is not), sometimes different interpretations of the same word or sentence are possible, which could be understood by the duration of breaks between words, tone of voice, facial expression, and so on. Some sentences are difficult to understand, even by humans. "*I saw Ahmed on a hill with a telescope*" could be interpreted in different ways. This problem is called the *ambiguity* of natural language. For these reasons - to avoid ambiguity and different interpretations - restricted special languages that have simpler grammars (structure) and restricted vocabulary, are used to communicate with machines (computers in particular). These are called computer programming languages.

Computers are electronic devices. They can only interpret electrical signals. They can be programmed based on their ability to interpret these electrical signals; by asking them to perform different tasks when they detect a signal or when they do not detect a signal. For example, if there are three wires that must have an electrical signal of [+5]

volts to indicate that there is a signal [interpreted as ON or 1], and [-5] volts to indicate that there is no signal [or simply as OFF or 0], the computer can be instructed to interpret the sequence [000] to be the number zero and [001] to be the number one, and [010] to be the number two, and [011] to be the number three, and so on. This is called the *binary system*. A program expressed in this form is usually said to be written in machine language. This language is also known as *low level language* because it is close to the machine hardware structure.

However, to perform any non-trivial task, thousands and thousands of these data values and instructions have to be written, and any mistake could lead to undesirable behavior. It is also extremely difficult to write these instructions and to correct them if there are errors. For this reason, it was suggested to assemble or group some of these binary digits into symbols, called *mnemonics*, and write a program (called an *assembler*) to read these symbols and convert them to machine code. These programs are known as assembly language programs. Assembly programs are at a higher level (in the programming language hierarchy) than machine code. Assembly programs are easier to write than machine code, but it is still difficult for humans to write and modify them. This is why *high level programming languages* were introduced. In a high level language, the programmer uses a *compiler*, which takes each statement in the programs, and translates it to machine code for the computer to understand.

1.2.1 Programs

In section 1.1, we mentioned that computers are machines that perform certain tasks, such as addition. We have to express what tasks we want it to perform, and in what order. If we tell the computer that we want it to add two numbers, it would know how to do that. For example in FORTRAN we can say X = 3 + 5. This asks the computer (we will see how later) to add 3 and 5, and store the value in X. This is a simple command, or program statement, that uses the computer's ability to perform the addition task or operation. A sequence of such statements is called a *program*.

A program is a sequence of statements that fully and clearly describes how a problem should be solved. The programs that tell the computer what to do, are usually called "*Software*".

A program should be written in a language that the computer understands. There are different kinds of languages used for different purposes. Some of the most widely used programming languages include FORTRAN, PASCAL, C, LISP, COBOL and PROLOG. All of these languages are high level programming languages.

1.3 Software Life Cycle

The production of software is similar to the production of artifacts in other engineering fields. A building, for example, might be constructed by laying bricks here and there, without an overall plan or a blue-print. However except for the simplest of buildings, the results would not be satisfactory, unsafe to say the least. The correct engineering method of constructing a building requires that the architect or civil engineers understand the requirements for constructing the building (e.g. residential), produce a preliminary design, verify it with the customer and modify the design accordingly, before the actual building is constructed. The process of software design is similar. The

programmer, or software engineer, should understand and analyze the problem to be solved well before any program is written. After the problem is analyzed, the approach for solving the problem should be identified. A solution is then designed and developed. After a solution is identified, the programmer can start writing the program code. After the code is written it has to be verified and checked for any mistakes or inconsistencies with the requirements, and the process is then repeated until the program behaves as required.

1.4 Modular Software Design

One approach for software development that has been shown to be effective for the production of large software systems is *stepwise refinement* or *top-down design*. Stepwise refinement is a form of *divide and conquer* strategy of problem solving. The basic idea is to divide the problem being solved into a number of steps, each of which can be described by an algorithm which is simpler and more manageable than an algorithm that describes the complete problem as a whole. Using this approach, problems that might seem difficult at the beginning are reduced to smaller problems that can be handled individually. In large software projects, different software engineers work on different sub-problems or modules. When they are done, the process of combining the modules to construct the solution of the original problem is conducted, and is usually straight-forward.

In this course, we apply the concepts of top-down design to solve simple scientific and engineering problems. The knowledge that you gain while you develop skills in top-down design will be valuable for you in other areas of problem solving in your field of study, not only in programming and software development.

1.5 Software Systems and Tools

To develop software, programmers need to use certain systems and tools. In this section we introduce some of the tools we will be using in this course. These include an editor and a compiler. All these tools are programs used by the computer system to assist the programmer in developing, running and maintaining programs.

1.5.1 Editors

To write programs and enter data in the computer, the programmer or user needs to use a tool called an *editor*. The editor allows the user to create and modify *files*. You can think of a file as a reserved area to write programs and data, just as you can write it on a piece of paper. However to enable the computer to read your program, it has to be written in a file, in a form that the computer can interpret. We will see in section1.5.3 the form of a FORTRAN program.

Editors allow their users to add, modify and delete things from a file. These things include characters, words, lines, pages and so on. There are some editors that offer other features and facilities. These include checking spelling mistakes, repeating words, lines and other things. In some systems, you can edit more than one file at the same time. You can copy from file to file. The features of editors are many and we will not attempt to enumerate them here. It suffices to know the purpose of using an editor, and that there are several kinds of editors available for use.

1.5.2 Compilers

In section 1.5.1, we mentioned that an editor enables the programmer to create files of programs and data according to specific forms. Some programming languages require that the program be written in a specific form so that it is easy to interpret. The computer uses a program called a *compiler* to read the program from a file that the programmer writes in, and converts the program into machine language. The FORTRAN compiler requires that the program be written in a specific form so that the compiler can perform the conversion to machine language.

1.5.3 FORTRAN Programs

FORTRAN (FORmula TRANslation) was developed in the fifties as a programming language for scientific and engineering applications. In 1977, standards for FORTRAN were revised, which resulted in a version of FORTRAN that came to be known as FORTRAN77. This is the version of FORTRAN that we will be using in this course. Using any editor, a programmer writes his/ here program in a file. A file consists of a collection of lines (which could also be called statements or records). The FORTRAN compiler requires that all program statements or lines, have a specific structure. A line can hold a maximum of 80 characters. Thus you can think of the program file as having 80 columns. The first position on the line is column one, the second position is column two and so on. Each program statement must begin in a new line and must be typed between columns 7 to 72 of the file. The compiler ignores any characters in columns 73 to 80. Columns 1 to 5 are used to include a label or a statement number, which is used to identify a specific line or statement of the program. Column 6 is used for continuation, which might be needed if the program statement or line is too long to fit in columns 7 to 72. Any character, except a zero, placed in column 6, indicates that this line is a continuation of the previous line.

A "*" or the character "C" in column one indicates that the line is a comment line. The compiler ignores what is typed on a comment line and does not execute it. This is useful for programmers to write descriptions of the different parts of their programs.

Each program should end with the "**END**" statement. This signifies the physical end of the program. The **STOP** statement signals the logical end of the program. While the **END** statement appears at the end of the program, the **STOP** statement may appear anywhere in the program, possibly, to stop execution of the program under certain conditions. The compiler sequentially executes each statement in the program. Exceptions to this sequential execution is possible using special FORTRAN statements such as **GOTO**, **IF** and **DO**. These are used to perform selection and repetition, as we shall see in later chapters.

1.5.4 Conclusion

In this course, you will be introduced to the basic concepts of computing and computer programming. The skills you gain in this course will enable you to start using computers as tools to solve the engineering and scientific problems you will encounter during your study. You should keep in mind that what you encounter in this course is but a drop in the ocean. The field of computer science is growing rapidly. As scientists and engineers, it is important to educate ourselves in different areas of technology. Without this new

technology, we will not be able to succeed and excel in our studies. It is also important to continue educating ourselves by identifying new developments in these areas. This course is the starting point. You should continue this process in order to remain competitive. Accordingly, when you study the material in this course, you should attempt to relate it to your field of study, and consider how the use of such tools can facilitate and enhance your productivity, and aid in the understanding of the material that you have already taken as well as the material that you will study in the future.

1.6 Exercises

1. Indicate the following statements as either TRUE or FALSE:
    1. Syntax errors are detected during compilation.
    2. A compiler is a hardware component that translates programs written in a high level language to a machine language.
    3. The input unit is the part of the computer that controls all the other parts.
    4. The last statement in a FORTRAN program should be the **END** statement.
    5. FORTRAN is a high level language.
    6. A comment statement is used for documentation purposes.
    7. Dividing by zero will cause a compilation error.
    8. If a FORTRAN statement exceeds column 72, then '+' at column # 6 in the next line can be used to continue the statement on that line.
    9. A computer is a machine used to solve problems only.
    10. A compiler checks the syntax of the program and converts the program into machine language.
    11. A program is a set of computer instructions.
    12. One can use as many '**STOP**' and '**END**' statements as he/she wishes in a single program.

2. Which of the following statement(s) is/are correct according to FORTRAN:
    A. Only column 1 is used for the statement label.
    B. Column 6 is used for comment.
    C. Column 1-5 is used for the statement label.
    D. Column 7 is used for the continuation line.
    E. Characters C or * in Column 1 is used to comment a line.

3. For each item of list (A), choose the correct definition from list (B) :

List A	List B
Assembler	1. A machine that converts an assembly language program into machine language.
Compiler	2. The physical components of a computer.
Software	3. A machine that converts a high level language program into machine language.
Hardware	4. A fundamental computer component that controls the operations of the other parts of the computer.
	5. Programs used to specify the operations in a computer.
	6. A fundamental computer component that performs all arithmetic and logic operations.
	7. A program that converts an assembly language program into machine language.
	8. A program that converts a high level language program into machine language.

4. For each term in list (A) choose the correct definition from list (B)

List A	List B
A program	1. is a FORTRAN statement that indicates the logical end of the program.
A computer	2. is a machine that can solve all problems.
END	3. translates programs written in an assembly language to a machine language.
STOP	4. is a machine that uses instructions given by the user to solve a problem.
	5. is a sequence of instructions which, when performed, will do a certain task.
	6. is a FORTRAN statement that indicates the physical end of a program.

1.7 Solutions to Exercises

Ans 1.

1. T	2. F	3. F
4. T	5. T	6. T
7. F	8. T	9. F
10. T	11. F	12. F

Ans 2.

III and V

Ans 3.

Assembler	7
Compiler	8
Software	5

Hardware	2

Ans 4.

A program	5
A computer	4
END	6
STOP	1

2 DATA TYPES AND OPERATIONS

We use computers to manipulate information that consists of letters, digits, and other special symbols. Such information is the interpretation of *data*. Although the word **data** is the plural of **datum**, many computer specialists use data as a mass noun such as *water* and *sand*. Data can be of different types. The basic data types in FORTRAN 77 are: **integer**, **real**, **character**, and **logical**. In this chapter we present these types in detail.

2.1 Constants

A constant is a fixed value of a data type that cannot be changed.

2.1.1 Integer Constants

Integer constants are whole numbers. An integer constant does not have a decimal point. Examples of integer constants are:

32 0 -6201 27 -83 1992

2.1.2 Real Constants

A real constant is a constant number that has a decimal point. Examples of real constants are 1.23, -0.0007, 3257.263, 5.0, 0.00002, 18., 774.00000, -64.9899 and 94000000000000000.0. The last number in the previous example leads us to the scientific notation for real numbers. 94000000000000000.0 can be written as 9.4×10^{16} or as 0.94×10^{17}. In FORTRAN, this number can be written in two possible ways: as 94000000000000000.0, or in scientific notation as 9.4E16 or 0.94E+17. Usually, such numbers are written in a way that the value of the first part is less than 1.0 and is greater than or equal to 0.1. The following table shows some examples of real numbers and their presentation in FORTRAN:

Real Number	Decimal Notation	FORTRAN Representation
6.3×10^{-5}	0.000063	0.63E-04
4.932×10^{7}	49320000.0	0.4932E+08
-5.7×10^{-6}	-0.0000057	-0.57E-05
5.7×10^{-6}	0.0000057	0.57E-05
5.7×10^{6}	5700000.0	0.57E+07

2.1.3 Logical Constants

There are two logical constants; *true* and *false*. In FORTRAN, the logical constant *true* is written as **.TRUE.** and the logical constant *false* is written as **.FALSE.**.

2.1.4 Character Constants

FORTRAN allows character usage and manipulation. Character constants must be placed between two consecutive single quotes. A character constant is also referred to as a character string. The following table shows some character constants and their representation in FORTRAN:

Character Constant	FORTRAN Representation
THIS IS CHAPTER TWO	'THIS IS CHAPTER TWO'
MORE THAN ONE BLANK	'MORE THAN ONE BLANK'
ISN'T IT?	'ISN''T IT?'
1234 AS CHARACTERS	'1234 AS CHARACTERS'

Note that if a single quote needs to be included in a character constant, it should be written as two single quotes.

2.2 Variables

A variable is an object of a certain data type that takes a value of that type. A variable, as the name suggests, can change its value through certain FORTRAN statements such as the assignment statement (section 2.5) and the **READ** statement (section 2.6). When a variable is defined, the compiler allocates specific memory location to that variable. This location must be given a name to be referenced later. The name of such a location is called a *variable name*. We shall use the term *variable* to mean *variable name*. Before using a variable we may define it. The definition of a variable means that we are allocating a memory location for that variable. However, it does *not* mean that the compiler assigns a value to the variable. There are some rules for choosing variable names in FORTRAN. These rules are as follows:

- The variable should start with an alphabetic character (A, B, C,...,Z)
- The length of the variable should not exceed 6 characters.
- A variable may contain digits (0, 1, 2, ..., 9).
- A variable should not contain special characters ($, ;, ,, :, !, ~, ^,(,{, [,), },], <, >, ?, “, ‘, \, | , @, %, &, #, +, -,/,*, .., etc.).
- A variable should not contain blanks.

Examples of valid and invalid variable names are given below:

Variable	Comment
TRY	Valid.
NAME21	Valid.
NAME211	Invalid. Length is more than 6 characters.
A+B	Invalid. Special character '+' can not be used.
5TEST	Invalid. Name does not start with a letter.
FIVE7	Valid.

The following subsections present different variable types and how to define them.

2.2.1 Integer Variables

Integer variables can hold only integer values. There are two ways to define an integer variable in FORTRAN: *explicitly* and *implicitly*. The *explicit* definition allows us to define variable types, irrespective of the first letter of the variable name. In such a case, we must use the **INTEGER** statement. The general form of this statement is as follows:

INTEGER *list of integer variables*

where *list of integer variables* is a list that has the names of variables separated by commas. The **INTEGER** statement is a FORTRAN declaration statement. This statement must be typed starting in either column 7 or after and must appear at the beginning of the program before any other executable statement. In fact, all declaration statements must appear at the beginning of the program. The following examples demonstrate the use of the **INTEGER** statement:

Example	Comments
`INTEGER BOOKS, NUM, X`	Three integer variables: BOOKS, NUM, X
`INTEGER Y1, AB3W`	Two integer variables: Y1, AB3W
`INTEGER CLASS, ID, TOTAL`	Three integer variables: CLASS, ID, TOTAL
`INTEGER SUM`	One integer variable: SUM

It is a good programming habit to use *explicit* definition in writing their programs. This minimizes logical errors that may arise while running such programs.

In *implicit* definition, we choose a variable name that starts with one of the following letters: **I, J, K, L, M, N.** Hence, any variable that starts with one of these letters is considered implicitly as an integer variable unless it is otherwise explicitly stated. Examples of integer variables are:

NUMB, N1, LAB, ISUM, JX, KILO, MEMO.

Implicit definition is assumed when a programmer forgets to use explicit definition.

2.2.2 Real Variables

Real variables can hold only real values. As was the case in integer variable definition, there are two ways to define a real variable: *explicitly* and *implicitly*. The explicit definition allows us to define variable types irrespective of the first letter of the variable name, using the **REAL** statement. The general form of this statement is as follows:

REAL *list of real variables*

where *list of real variable* is a list that has the names of variables separated by commas. The **REAL** statement is a FORTRAN declaration statement. It must be typed starting in either column 7 or after and must appear in the beginning of the program before any other executable statement. The following examples demonstrate the use of the **REAL** statement:

Example	Comments
`REAL NOTES, NUM2, IX`	Three real variables: NOTES, NUM2, IX
`REAL M1, AB3`	Two real variables: M1, AB3
`REAL INSIDE, KD2, SBTOT`	Three real variables: INSIDE, KD2, SBTOT
`REAL J1SUM`	One real variable: J1SUM

We should try our best to declare our variables explicitly. If we forget to use explicit definition, then FORTRAN compilers assume implicit definition.

In *implicit* definition, any variable that does not start with one of the letters **I, J, K, L, M, N** is considered, implicitly, as a real variable unless the type of the variable is explicitly stated. Examples of real variables are:

YNUMB, X1, PERC, SUM, RJX, TOTAL, STID, A5, EPSLON, PI.

2.2.3 Logical Variables

Logical variables have either a **.TRUE.** or a **.FALSE.** value. There is only one way to define logical variables - they must be declared explicitly. The statement that is used to define logical variables is the declarative **LOGICAL** statement. This statement should be typed starting either in column 7 or after. It must appear at the beginning of the program before any executable statement. The general structure of the **LOGICAL** statement is:

LOGICAL *list of logical variables*

where *list of logical variables* is one or more variables separated by commas. Examples of **LOGICAL** statement usage are given below:

Example	Comments
`LOGICAL TEST, FLAG, Q, P`	Four logical variables: TEST, FLAG, Q, P
`LOGICAL M5`	One logical variable: M5
`LOGICAL SORTED, LINK`	Two logical variables: SORTED, LINK

2.2.4 Character Variables

Character variables must be given character constants as their values. Only explicit definition allows us to define character variables. The declaration statement that is used in character definition is the **CHARACTER** statement. As is the case in other types of declaration statements, the **CHARACTER** declaration statement must appear at the beginning of the program and should be typed before any executable statement. The general form of the **CHARACTER** statement is as follows:

CHARACTER *list of character variables with their lengths*

or

CHARACTER*n *list of character variables with their lengths*

where *list of character variables with their lengths* consists of one or more variables separated by commas. Each variable may be followed by ***k**, where **k** is a positive integer specifying the length of the string that particular variable can hold. If ***k** is not specified, the length of that variable is assumed to be **n**. If **n** is not specified, the length is assumed to be 1. The following table shows some examples of **CHARACTER** statements.

Example	Character variables and their lengths
`CHARACTER NAME*20`	NAME is a character variable of length 20
`CHARACTER*6 M, WS*3, IN2`	M and IN2 are of length 6; WS is of length 3
`CHARACTER T1, T2, T3`	T1, T2 and T3 are of length 1
`CHARACTER Z*8, TEST`	Z is of length 8 and TEST is of length 1
`CHARACTER*12 Z1, Z2`	Z1 and Z2 are of length 12

Detailed character manipulation and usage will be discussed in chapter 10. In the remainder of this chapter, we present arithmetic and logical operations, the assignment statement, and simple input/output statements.

2.3 Arithmetic Operations

Addition, subtraction, multiplication, division, and exponentiation (power) are called arithmetic operations. The following subsections present details about these operations.

2.3.1 Arithmetic Operators

In FORTRAN there are five basic operators. These operators are shown in the following table with the sequence in which they are evaluated (precedency):

FORTRAN Operator	Operation	FORTRAN Example	Math Notation	Precedency
**	Exponentiation	`X ** Y`	x^y	1
*	Multiplication	`X * Y`	$x \times y$	2
/	Division	`X / Y`	$x \div y$	2
+	Addition	`X + Y`	$x + y$	3
-	Subtraction	`X - Y`	x - y	3

An arithmetic expression consists of one or more arithmetic operations. Operations that are applied on two operands are called binary operations. Operations that are applied on one operand are called unary operations. The minus operator '-' may be used as a unary operator or as a binary one. An operand can be a constant value, a variable that has been given a value, or a correct expression.

In any arithmetic expression, parentheses have the highest priority (precedence) in evaluation. In the case of nested parentheses (parentheses inside parentheses), evaluation starts with the most-inner parentheses. The next higher priority operator is

the exponentiation (also called power) operator '**'. If there are two or more consecutive exponentiation operators in an arithmetic expression, evaluation of these exponentiation operations is done from right to left. For example, in the expression 2**2**3, we start evaluating 2**3 (which is 8) and after that we evaluate 2**8 (which is 256). Division and multiplication operators have the same priority, but they are lower in priority than the exponentiation operator. The addition and subtraction operators have the same priority which is lower than the priority of multiplication and division operators. Operators with the same priority are evaluated from left to right with the exception of the exponentiation operator as explained earlier.

There are two restrictions on the use of arithmetic operators. The first restriction is that no two operators must appear consecutively. For example, if the expression 2 * -3 is intended, in FORTRAN, it should be written as 2*(-3). The second restriction is on the use of the exponentiation operator. This operator must not be used to raise a negative number to a real exponent. For example, expressions such as (-2.0) ** 1.5 or (-3) ** 2.3 are not allowed in FORTRAN language. To compute x^y, when y is real, most FORTRAN Compilers use the mathematical formula $e^{y \ln x}$. When x is negative, the value of $\ln x$ is undefined.

2.3.2 Integer Operations

An operator between two integer operands is considered to be an integer operator and the operation is considered to be an integer operation. Integer operations always produce integer results. The fraction part is ignored. The following table shows some examples of integer operations:

Expression	Value	Comment
50 - 23	27	
3 ** 2	9	
5 * 7	35	
8 / 2	4	
8 / 3	2	Fraction part is truncated (not 2.6666667)
9 / 10	0	Fraction part is truncated (not 0.9)

Note that the expression I/J * J is not always equivalent to I. For example, if I and J are integer variables, and the value of I is 17 and the value of J is 6, the expression becomes 17 / 6 * 6. To evaluate this expression we consider operator precedence. Since operators '/' and '*' have the same priority, they are evaluated from left to right. We start with 17 / 6. The two operands are integers and therefore '/' here is an integer operator. The result must be an integer, which in this case evaluates to 2. Now, evaluation proceeds as 2 * 6 which results in 12 and not 17.

2.3.3 Real Operations

An operator between two real operands is considered to be a real operator and the operation is considered to be a real operation. Real operations produce real results. The following table shows some examples of real operations:

Expression	Value
50.0 - 23.0	27.0000000
3.0 ** 2.0	9.0000000
5.0 * 7.0	35.0000000
8.0 / 2.0	4.0000000
8. / 3.0	2.6666667
9. / 10.	0.9000000
9.3 / 3.2	2.9062500

2.3.4 Mixed-mode Operations

An operator between an integer operand and a real operand is considered to be a mixed-mode operator and the operation is considered to be a mixed-mode operation. Mixed-mode operations produce real results. The following table shows examples of mixed-mode operations:

Expression	Value	Comment
50 - 23.0	27.0000000	
3.0 ** 2	9.0000000	
3 ** 2.0	9.0000000	
4** 0.5	2.0000000	
5.0 * 7	35.0000000	
56.7 / 7	8.1000000	
8 / 2.0	4.0000000	
8.0 / 3	2.6666667	
9 / 10.	0.9000000	Decimal point can be placed without zero.
17 / 6 * 6.0	12.0000000	'/' is an integer operator and '*' is a mixed mode operator

The number of positions to the right of the decimal point in a real number depends on the computer used. In the examples above, we have assumed that the computer allows up to 7 positions.

2.3.5 Examples

Example 1: *Evaluate the following arithmetic expression*

20 - 14 / 5 * 2 ** 2 ** 3

Solution:

Expression: 20 - 14 / 5 * 2 ** 2 ** 3
Priority is for ** from right to left
Step 1: 2 ** 3 = 8 (integer operation)
Expression: 20 - 14 / 5 * 2 ** 8
Priority is for ** from right to left
Step 2: 2 ** 8 = 256 (integer operation)
Expression: 20 - 14 / 5 * 256

Priority is for / and * from left to right
Step 3: 14 / 5 = 2 (integer operation)
Expression: 20 - 2* 256
Priority is for *
Step 4: 2 * 256 = 512 (integer operation)
Expression: 20 - 512
Priority is for -
Result: -492

Example 2: *Evaluate the following arithmetic expression*

14.0 / 5 * (2 * (7 - 4) / 4) ** 2

Solution:

Expression: 14.0 / 5 * (2 * (7 - 4) / 4) ** 2
Priority is for expression inside the inner most parenthesis
Step 1: (7 - 4) = 3 (integer operation)
Expression: 14.0 / 5 * (2 * 3 / 4) ** 2
Priority is for expression inside the parenthesis
Step 2 & 3: (2 * 3 / 4) = (6 / 4) = 1 (2 integer operations)
Expression: 14.0 / 5 * 1 ** 2
Priority is for **
Step 4: 1 ** 2 = 1 (integer operation)
Expression: 14.0 / 5 * 1
Priority is for / and * from left to right
Step 5: 14.0 / 5 = 2.8000000 (Mixed mode operation)
Expression: 2.8000000 * 1
Priority is for *
Result: 2.8000000

Example 3: *Rewrite the following FORTRAN expression as a mathematical form*

X + Y / W - Z

Solution:

$$x + \frac{y}{w} - z$$

Example 4: *Rewrite the following FORTRAN expression as a mathematical form*

X ** (1.0 / 2.0) / Y ** Z

Solution:

$$\frac{\sqrt{x}}{y^z} \quad or \quad \frac{x^{\frac{1}{2}}}{y^z}$$

Example 5: *Convert the following mathematical expression into FORTRAN expression. Use minimum number of parenthesis*

$$\frac{\sqrt{a+b}}{a^2 - b^2}$$

Solution:

(A + B) ** 0.5 / (A ** 2.0 - B ** 2.0)

2.4 Logical Operations

Logical operations evaluate to either **.TRUE.** or **.FALSE.**. The following subsections discuss logical operators, relational operators and logical expressions:

2.4.1 Logical Operators

This section discusses the three logical operators: **.AND., .OR.** and **.NOT.**. The **.AND.** operator is a binary logical operator that produces .TRUE., if and only if, both its operands have a .TRUE. value. If any of the operands have a .FALSE. value, the result of the operation is .FALSE.. The **.OR.** operator is a binary logical operator that produces .FALSE. if and only if both operands have the value .FALSE.. otherwise, the result is .TRUE.. The unary logical operator **.NOT.** produces the opposite value of its operand. The following table shows the results of the three logical operations .AND., .OR. and .NOT. on different operand values, assuming P and Q are logical variables:

P	Q	P .AND. Q	P. OR. Q	.NOT. P
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.

The *.NOT. operator has the highest priority of the three logical* operators followed by the *.AND.* operator. The *.OR. operator has the lowest priority.* These operators are shown in the following table with the sequence in which they are evaluated (precedency):

Logical Operator	FORTRAN Example	Precedence
.NOT.	.NOT. P	1
.AND.	P .AND. Q	2
.OR.	P .OR. Q	3

Example 1: *Evaluate the following logical expression:*

.FALSE. .OR. .NOT. .TRUE. .AND. .TRUE.

Solution:

Expression: .FALSE. .OR. .NOT. .TRUE. .AND. .TRUE.

priority is for .NOT.

Step 1: .NOT. .TRUE. is .FALSE.

Expression: .FALSE. .OR. .FALSE. .AND. .TRUE.

priority is for .AND.

Step 2: .FALSE. .AND. .TRUE. is .FALSE.

Expression: .FALSE. .OR. .FALSE.

priority is for .OR.

Result: .FALSE.

Example 2: *Assume that the following declaration is given:*

LOGICAL FLAG

If it is known that the expression

.NOT. FLAG .OR. .FALSE.

has the value .TRUE., what is the value of FLAG?

Solution:

The final result must be .TRUE.. The last step is *somevalue* .OR. .FALSE. because the .NOT. operator has higher priority than the .OR. operator. *somevalue* .OR. .FALSE. will have the value .TRUE. if and only if the value of *somevalue* is .TRUE.. But *somevalue* is equivalent to .NOT. FLAG, therefore the value of FLAG is .FALSE..

2.4.2 Relational Operators

The values of arithmetic expressions can be compared using relational operators. The following table shows the different relational operators. Assume all variables have been initialized:

Operator	Math	Example	Description
.EQ.	=	`X .EQ. Y`	True if X and Y are equal
.NE.	≠	`N .NE. 8`	True if N is not equal to 8
.GT.	>	`P1 .GT. 7.3`	True if P1 is greater than 7.3
.GE.	≥	`SM .GE. TOT`	True if SM is greater than or equal to TOT
.LT.	<	`A+B.LT.A*2.0`	True if the sum of A and B is less than 2A
.LE.	≤	`NUM.LE.CLASS`	True if NUM is less than or equal to CLASS

A relational expression evaluates to either .TRUE. or .FALSE.. Relational operators have lower priority than arithmetic operators and higher priority than logical operators. They are evaluated from left to right. The next subsection presents the use of relational, logical, and arithmetic operators in logical expressions.

2.4.3 Logical Expressions

A logical expression evaluates to .TRUE. or .FALSE.. It may contain different types of variables and operators. It may contain arithmetic expressions, logical expressions, and relational expressions. Logical expressions are used in selection constructs which are discussed in chapter 3. The evaluation of a logical expression starts with the evaluation of arithmetic expressions first followed by the relational expressions, and finally the logical expressions. The following examples demonstrate the evaluation of logical expressions:

Example 1: *Given that X has a value of 3.0, Y has a value of 5.0, Z has a value of 10.0, and FLAG is a logical variable with .FALSE. value, evaluate the following FORTRAN expression:*

.NOT. FLAG .AND. X*Y .GT. Z .OR. X+Y .GT. Z

Solution:

Expression: .NOT. FLAG .AND. X*Y .GT. Z .OR. X+Y .GT. Z

Evaluate arithmetic expressions first.

Expression: .NOT. FLAG .AND. 15.0 .GT.10.0 .OR. 8.0 .GT.10.0
Evaluate relational expressions next.
Expression: .NOT. FLAG .AND. .TRUE. .OR. .FALSE.
Evaluate logical expressions. Start with .NOT..
Expression: .TRUE. .AND. .TRUE. .OR. .FALSE.
Evaluate logical .AND. next.
Expression: .TRUE. .OR. .FALSE.
Evaluate .OR. next
Result: .TRUE.

Example 2: *When is the value of the following expression .TRUE.? Assume K and L are integers.*

K / L * L .EQ. K

Solution:

If K is divisible by L, the value of the expression is .TRUE.. Otherwise, the value will be .FALSE..

Example 3: *Given that X has a value of 3.0, Y has a value of 5.0, Z has a value of 10.0, and FLAG is a logical variable with the value .FALSE., find the value of each of the following expressions:*

.NOT. FLAG .OR. FLAG
X .GT. Y - Z / 2.0
X*Z .EQ. 20.0 .OR. FLAG .AND. .NOT. Z .EQ. 5.0
X .GT. Y .AND. X .GT. Z .OR. X .LT. Y .AND. X .LT. Z
Z*10 .NE. Y*30 .AND. X .LE. Y .AND. FLAG
.NOT. FLAG .AND. FLAG
.NOT. .NOT. FLAG

Solution:

Expression	**Value**
`.NOT. FLAG .OR. FLAG`	.TRUE.
`X .GT. Y - Z / 2.0`	.TRUE.
`X*Z .EQ. 20.0 .OR. FLAG .AND. .NOT. Z .EQ. 5.0`	.FALSE.
`X .GT. Y .AND. X .GT. Z .OR. X .LT. Y .AND. X .LT. Z`	.TRUE.
`Z*10 .NE. Y*30 .AND. X .LE. Y .AND. FLAG`	.FALSE.
`.NOT. FLAG .AND. FLAG`	.FALSE.
`.NOT. .NOT. FLAG`	.FALSE.

2.5 Assignment Statement

The assignment statement in FORTRAN assigns a value to a variable. The general form of the FORTRAN assignment statement is:

variable = expression

where *expression* must have a value of the same type as the *variable* with one exception: integer values can be assigned to real variables and real values can be assigned to integer variables. In assigning a real value to an integer variable, the decimal part is truncated before the value is stored in the variable. In the case of an integer value

being assigned to a real variable, the integer value is converted to a real value before it is stored in the variable. The FORTRAN assignment statement is **not a mathematical equation**. Therefore, it is possible to write assignment statements such as:

```
      X = 1.0
      X = X + 1.0
```

where the first statement assigns the value 1.0 to the variable X. The second statement evaluates the expression X + 1.0 which will be 2.0 and then assigns the result to the variable X. It should be clear that the old value of X (i.e 1.0) is changed to the new value (i.e. 2.0).

Example 1: *Write FORTRAN assignment statements to store the real number 3.25 into the variable X1 and 7.0 into the variable Y1.*

Solution:

```
      X1 = 3.25
      Y1 = 7.0
```

Example 2: *Write a FORTRAN assignment statement to store in X1 the value stored in Y1.*

Solution:

```
      X1 = Y1
```

Example 3: *Write a FORTRAN assignment statement to increment X1 by 1.*

Solution:

```
      X1 = X1 + 1.0
```

Example 4: *Write a FORTRAN assignment statement to add to X1 the value of Y1.*

Solution:

```
      X1 = X1 + Y1
```

Example 5: *Write a FORTRAN assignment statement to store in X1 the contents of X1 times the contents of Y1.*

Solution:

```
      X1 = X1 * Y1
```

Example 6: *Assume that the coefficients of a quadratic equation are given as A, B, and C. Write FORTRAN assignment statements to find the two roots, ROOT1 and ROOT2, of the quadratic equation.*

Solution:

```
      ROOT1 = (-B + (B ** 2.0 - 4.0 * A * C) ** 0.5) / (2.0 * A)
      ROOT2 = (-B - (B ** 2.0 - 4.0 * A * C) ** 0.5) / (2.0 * A)
```

Example 7: *Given SUM as the sum of student grades in an exam and COUNT as the number of students, write an assignment statement to find the average AVER.*

Solution:

```
      AVER = SUM / COUNT
```

Example 8: *Write FORTRAN assignment statements to exchange the values of the variables X and Y. (Hint: Use a temporary variable T)*

Solution:

```
T = X
X = Y
Y = T
```

Example 9: *If the variable **NAME** is declared as follows:*

```
CHARACTER NAME * 8
```

*what will the value of **NAME** be after the following assignment statement is executed?*
NAME = 'ICS101 FORTRAN'

Solution:

Since the length of the variable **NAME** is declared as 8, the assignment statement will assign the first 8 characters of the string constant to **NAME**. Hence, the value of **NAME** is going to be:

ICS101 F

Example 10: *Given the following declaration and assignment statements:*

```
CHARACTER MAJOR * 15
MAJOR = 'FINAL'
```

what is the value of the variable **MAJOR** ?

Solution:

Since the length of the variable **NAME** is declared as 15, the assignment statement will assign the string constant FINAL to the first 5 positions of **MAJOR** and fill the remaining 10 positions with blanks.

2.6 Simple Input Statement

We may assign a value to a variable by using either the assignment statement or by reading an input value into the variable. To read an input value from the terminal into a variable, we must use an input statement. There are two types of input statements: the formatted **READ** and the unformatted **READ**. This section presents the unformatted **READ** statement. The general form of the unformatted **READ** is

READ*, *list of variables separated by commas*

The following points must be noted while using the unformatted **READ** statement:

- Each read statement starts reading from a new line.
- If the input data is not enough in the current line, reading continues in the next line.
- The data values can be separated by blanks or comma.
- The data values must agree in type with the variables.
- Integer values can be read into real variables but real values must not be read into integer variables.
- Extra data on an input line is ignored.

2.6.1 Examples

Example 1: *Assume the following declaration:*

```
INTEGER NUM, M1, K, L1, L2, L3, K1, K2
REAL TOT, X1, YY, S, ST, A, X, Y, Z
```

*The following table gives examples of **READ** statements:*

Statement	Input Line	Effect
READ*, NUM, TOT	9 5.08	NUM = 9 TOT = 5.08
READ*, X1, YY	325 27	X1 = 325.0 YY = 27.0
READ*, M1	20.0	ERROR MESSAGE. DATA TYPE MISMATCH
READ*, K, S	18, 0.35E-2	K = 18 S = 0.35E-2
READ*, ST	-23.4	ST = -23.4
READ*, L1, L2, L3	7 6 5	L1 = 7 L2 = 6 L3 = 5
READ*, A, A	1.0, 2.0	A = 2.0
READ*, K1 **READ***, K2	5 8 20 9	K1 = 5 K2 = 20
READ*, X, Y, Z	5 8 20 9	X = 5.0 Y = 8.0 Z = 20.0

Example 2: *Assume the following declaration:*

```
CHARACTER NAME*9, STR1*5, STR2*3
LOGICAL P1, P2
```

*The following table gives examples of **READ** statements:*

Statement	Input Line	Effect
READ*, NAME	'AHMED ALI'	NAME = 'AHMED ALI'
READ*,STR1, STR2	'ALI' 'CLASS'	STR1 = 'ALI ' STR2 = 'CLA'
READ*, P1, P2	T F	P1 = .TRUE. P2 = .FALSE.

2.7 Simple Output Statement

The **PRINT** output statement is used to print the values of variables, expressions or constants. There are two types of **PRINT** output statements: the formatted **PRINT** statement and the unformatted **PRINT** statement. The formatted **PRINT** statement will be discussed in chapter 8. The general form of the unformatted **PRINT** statement in FORTRAN is

PRINT*, list of variables, expressions, or constants separated by commas

The following subsection presents some examples on **PRINT** statement.

2.7.1 Examples

Example 1: *In the table below, examples of the **PRINT** statement are given assuming the following initializations:*

```
      LOGICAL FLAG
      INTEGER K, L
      REAL S1, S2
      FLAG = .TRUE.
      K = 3
      L = 20
      S1 = 35.0
      S2 = S1 - K - L
```

Statement	Output	Comments
`PRINT*, K, S1`	`3 35.0000000`	Blanks depends the type of computer
`PRINT*, L+S2, W`	`32.0000000 ???????`	??????? for undefined
`PRINT*, L, FLAG`	`20 T`	
`PRINT*, L / K * K`	`18`	
`PRINT*, L / K * K * 1.0`	`18.0000000`	
`PRINT*, L * 1.0 / K * K`	`20.0000000`	May be 19.9999994 (accuracy)
`PRINT*,5,6+7, L, 2, K+3`	`5 13 20 2 6`	Constants and expressions
`PRINT*, 'K= ',K,' L IS ',L`	`K= 3 L IS 20`	Characters may be printed
`PRINT*, 'THIS TESTS'`	`THIS TESTS`	
`PRINT*, FLAG, .FALSE.`	`T F`	Logical values either T or F
`PRINT*`		Prints an empty line

Example 2: *In the table below, more examples of the* ***PRINT*** *statement are given assuming the following initializations:*

```
      CHARACTER*10 LSTNAM
      CHARACTER CLASS*5, MAJOR*4
      LSTNAM = 'AL-FORTRAN'
      CLASS = 'BATAL'
      MAJOR = 'ANY1'
```

Statement	Output	Comments
`PRINT*, CLASS, MAJOR`	`BATALANY1`	No blanks in between
`PRINT*,LSTNAM, ' ',MAJOR`	`AL-FORTRAN ANY1`	Explicit blank as it is

The following points must be noted while using the **PRINT** statement:

- Each **PRINT** statement starts printing on a new line.
- If the spaces in the line are not enough to hold the whole output, printing continues on the next line.
- A variable that does not have a value will produce question marks if it is printed.

2.8 A Complete Program

The following program reads three real numbers, prints them, computes their average and prints it:

```
C      THIS PROGRAM READS 3 REAL NUMBERS
C      AND COMPUTES AND PRINTS THE AVERAGE
C
       REAL NUM1, NUM2, NUM3, COUNT, AVER
       COUNT = 3.0
       READ*, NUM1, NUM2, NUM3
       PRINT*, 'THE NUMBERS ARE ', NUM1, NUM2, NUM3
       AVER = (NUM1 + NUM2 + NUM3) / COUNT
       PRINT*, 'THE AVERAGE IS ', AVER
       END
```

The first three lines are comment lines. We can insert comment lines anywhere in the program. Each comment line must start with 'C' or '*' in column one. The fourth statement of the program is the **REAL** declaration statement. It declares five real variables that are going to be used in the program. The next statement is an assignment statement that assigns 3.0 to the variable COUNT. The **READ** statement will read 3 values from the input line and assign them to the variables NUM1, NUM2, and NUM3, respectively. The first **PRINT** statement is used to print the values that were read. The next statement is an assignment statement that computes the average. The result is stored in the variable AVER. The second **PRINT** statement prints the average with a proper message. The last statement is the **END** statement. The **END** statement signals the physical end of the program.

If the input line of this program is

```
9.0   8.0   10.0
```

the output is as follows:

```
THE NUMBERS ARE 9.0000000 8.0000000 10.0000000
THE AVERAGE IS 9.0000000
```

In FORTRAN programs, execution starts from the beginning of the program and proceeds statement by statement, in sequence, unless there is an indication for changing the sequence. Statements that may change the sequence of execution are selection and repetition statements. Selection is discussed in chapter 3 and repetition in chapter 5.

2.9 Exercises

1. Evaluate the following arithmetic expressions:
   1. 4 ** 2 / 3
   2. ((2 + 6) / 2 + 3.0 /6.0 *4) * (2 / 4)
   3. 10 ** 2 ** 3
   4. 10 / 4 /4 + (2 - 10 / 2.0)
2. Indicate if the statements below are valid FORTRAN statements or not:
   1. Y + X = K
   2. AB = A * B
   3. **PRINT***, 1.0, '+', 2.0, '=', 1.0 + 2.0
   4. X = Y ** -3
   5. X12345 = 8.0
   6. X = Y = 5.0
   7. P = (Q + R) * (- (-8))

8. X3X = 8.0
9. **READ***, R+A
10. **READ***, NUM,NUM

3. What will be printed by the following FORTRAN 77 programs ?

```
1.      INTEGER I, J, K
        I = 300
        J = 500
        K = J/I
        PRINT*, K
        END
```

```
2.      INTEGER ONE,TWO,THREE,FOUR,FIVE
        ONE = 1
        TWO = 2
        THREE = 3
        FOUR = 4
        FIVE = THREE + FOUR ** ( ONE / TWO )
        PRINT*, FIVE
        END
```

```
3.      INTEGER  M, N
        READ*, M
        READ*, N
        PRINT*, M, N
        END
```

Assume the input for the program is:

```
7   9
```

```
4.      INTEGER I, J, K, L
        READ*, I, J
        READ*, K, I
        PRINT*, I, J, K, L
        END
```

Assume the input for the program is:

```
4   5   6
7   8   9
```

```
5.      REAL X
        X = 1.2
        X = X + 1.0
        X = X + 1.0
        X = X + 1.0
        PRINT*, X , X, X, X
        END
```

```
6.      REAL A, X
        A = 8 ** 1/3
        X = 25 ** 1/2
        PRINT*, X, A
        END
```

```
7.      INTEGER XLM, NUM1, NUM2
        REAL PNM
        READ*, NUM1, NUM2
        PNM = NUM1 / NUM2
        XLM = 3 / PNM * 3.00 ** NUM2
        PRINT*, PNM, NUM1, NUM2, XLM
        END
```

Assume the input for the program is:

```
3,2
```

4. What is the value of each of the following expressions? Use the following values if needed:

```
        REAL A, B
        INTEGER K, J
        A = 2.0
        K = -2
        B = 3.5
        J = 1
```

1. 6 * J / K * 4
2. 9 + K / 5 * A / 2
3. A / (B + K) / J
4. 3 ** J ** A ** 1 + K / J
5. -2 / 4 * 4 ** 2
6. -2 / 4.0 * 2 ** 2 + 2 * 4.0 ** 2
7. 3 ** 2.0 * (3.0 - 1) + 2.0 * 1 * 3.0
8. 5 ** 3 / 2 ** 5 / 2
9. (5 / 2) ** 1.0 ** 2
10. (1 + (3.2 * 2 - (5 - 4)))
11. ((2 + 6) / 2 + 3.0 / 6.0 * 4) * (2 / 4)
12. 99999 / 100000 - 1
13. 2 ** 2 ** 3
14. 9 / 4 * 2 ** 1 / 2
15. 900 / 3.0E2

5. Convert the following FORTRAN assignment statements into an algebraic form :
   1. W = (X / Y / Z * T) ** 3 + 1 + 1.674E-24 * C
   2. Q = 1012.0 * P ** 0.5 * (1.0 - P / 100.0)
   3. K = A * B / C - 2
6. Which of the following are valid FORTRAN variable names?
   1. CS101GRADE
   2. AH/Q
   3. PRICE
   4. +RATE
   5. 2THIRD
   6. NUMB12
   7. IDNUMB

8. WHOLE-SALE-PRICE
9. $FORT
10. Y8X
11. ALL*

7. Indicate the following statements as either TRUE or FALSE:
   1. A **REAL** statement is an executable statement.
   2. Compiling the statement Y = 2 ** 4 ** 3.5E50 will cause syntax error.
   3. The statement **INTEGER** X,Y,Z implies that XYZ is an integer variable.
   4. If J, K, and L are integers, then the FORTRAN expressions (J + K) / L and (J / L) +(K / L) are equivalent.
   5. The **INTEGER** statement can appear any where in the program.
   6. If K and L are integers, then the FORTRAN expressions K * L**2 / K**2 and K * (L**2 / K**2) are equivalent.
   7. **PRINT***,X=5 is a valid FORTRAN 77 statement.
8. Add the minimum number of parentheses to the FORTRAN expression

   A ** B ** 2 + B - C / D + A * B / C * D

   to be equivalent to the mathematical expression :

$$\frac{a^{(b)^{2+b-c}}}{d+a} \times \frac{b}{cd}$$

9. In the following FORTRAN expression the operators have been numbered :

   1 2 3 4 5 6 7 8 9

   A ** B ** 2 + B - C / D + A * B / C * D

   Give the order in which the operators are evaluated a cording to FORTRAN 77 rules. (only write the operator numbers in order)

10. Write a FORTRAN program to read a 3 digit number, then prints the hundredth, the tenth, and the ones digits. If the input is:

```
728
```

    The output should be:

```
THE HUNDREDS DIGIT =  7
THE TENTH DIGIT    =  2
THE ONES DIGIT     =  8
```

11. Write a FORTRAN program which reads the radius of a sphere and calculates the surface area and the volume of the sphere. Your program should print the radius, surface area and the volume:

    Surface area = $4\pi r^2$

    Volume = $\frac{4}{3}\pi r^3$

12. Convert the following mathematical expressions / assignments to FORTRAN expressions / assignments. (do not use extra parentheses)

    1. $2x + \frac{y}{2}$

2. $\sqrt{\frac{a+b}{a-b}}$
3. $\frac{r^2}{3} - \frac{ac^{\frac{3}{4}}}{2b}$
4. $\frac{1}{\frac{1}{r1}+\frac{1}{r2}+\frac{1}{r3}}$
5. $a = b + \frac{xy}{c+d} + 2$
6. $2a + c^{-6}$
7. $\frac{a+\sqrt[4]{b}}{\frac{2}{a^2+5}} - 1$

13. For each of the following FORTRAN expressions, write an equivalent expression by deleting all "REDUNDANT" parentheses (i.e. parentheses whose deletion does not change the result of the expression).
    1. (A*B) * C / ((X*Y) **2)
    2. ((A+B) ** 2 + (3*C) ** 3) ** (A/B)
    3. ((A-B) +C) +(D*E)
    4. (C*X) ** ((2-A) * B)
    5. -B + ((B**2 - (4 * (A*C)))) ** 0.05

14. Write a program that converts a quantity expressed in seconds to a correspondence quantity expressed in hours, minutes and seconds. If the input is:

```
8125
```

The output should be:

```
2 HOURS, 15 MINUTES, 25 SECONDS.
```

15. The input data to a certain program is more than what is required. The data is as follows:

```
4   5   12   10
6   1   8   13   19
3   2   9   0   7   18   20
```

Write a FORTRAN program to read enough data (i.e. using the minimum number of variables in the **READ** statement) to print the following output:

```
4 5
1 8
9 0
```

(your program should have **READ** and **PRINT** statements only)

16. i) The output of the program below is as follows:

```
8
```

Fill in the spaces to get the output shown above

```
      INTEGER K, M, N
      K = -----
      M = 2
      N = 3
      PRINT*, M**N**M**K
      END
```

ii) The output of the program below is as follows:

```
1 4
7 8 10
```

Fill in the spaces to get the output shown above

```
      INTEGER K1, K2, K3, K4, K5
      READ*, -------
      READ*, -------
      READ*, -------
      PRINT*, K1, K2
      PRINT*, K3,K4,K5
      END
```

Assume the input for the program is:

```
1 2 3
4 5 6
7 8
10 11 12
```

17. Determine whether the following conditions are TRUE or FALSE. Assume A = 3.5, B = -4.1, I = -4, J = 9, FLAG = .TRUE. when needed:
    1. (3.0/2.LT.1.5).AND.(4/2.GT.1)
    2. .FALSE..AND..TRUE..OR..NOT.(.FALSE..AND..TRUE.)
    3. .NOT..FALSE..AND..TRUE.
    4. .NOT..FALSE..OR..TRUE..AND.3/2.EQ.1.0
    5. .NOT.5**2.EQ.5*2.AND.0.GT.5.OR.5*2+2.GT.0
    6. A.GT.B.OR.I.EQ.J.AND.FLAG
    7. A+I-4.GT.B-3+2*J.OR.A*B.GT.2.0*I
    8. FLAG.OR.(A-I)/(B-J).GT.1.021
    9. .NOT.(A.GT.B).OR.(I.GT.J)
    10. (A+B)/(I+J).LT.-5.0.AND..NOT.A*I.LE-.14.0
    11. .NOT.(.NOT.FALSE.).AND..TRUE..OR..FALSE.

2.10 Solutions to Exercises

Ans 1

1. 5 2. 0.0 3. 100000000 4. -3.0

Ans 2.

1. Invalid 2. Valid 3. Valid 4. Invalid 5. Valid
6. Invalid 7. Valid 8. Valid 9. Invalid 10. Valid

Ans 3.

1
4

Error Message
8 5 7 ???????
4.2 4.2 4.2 4.2
12.0 2.0
1.0 3 2 27

Ans 4.

1. -12	2. 9.0	3. 1.3333333	4. 1.0	5. 0	6. 30.0
7. 24.0	8. 1	9. 2.0	10. 6.4	11. 0.0	
12. -1	13. 256	14. 2	15. 3.0		

Ans 5.

1.. $w = \left(\frac{\frac{x}{y}}{z} t \right)^3 + 1 + 1.674 \times 10^{-24} c$

2. $q = 1012 p^{\frac{1}{2}} \left(1 - \frac{p}{100} \right)$

3 $k = \frac{ab}{c} - 2$

Ans 6.

1. Invalid	2. Invalid	3. Valid	4. Invalid	5. Invalid
6. Valid	7. Valid	8. Invalid	9. Invalid	
10. Valid	11. Invalid			

Ans 7.

1. FALSE	2. FALSE	3. FALSE	4. FALSE	5. FALSE
6. FALSE	7. FALSE			

Ans 8.

A ** B ** (2 + B - C) / (D + A) * B / (C * D)

Ans 9.

2 1 5 7 8 9 3 4 6

Ans 10.

```
INTEGER N, M, J, K
READ*, N
M = N / 100
N = N - M * 100
J = N / 10
K = N - J * 10
PRINT*, 'THE HUNDREDS DIGIT = ', M
PRINT*, 'THE TENTH DIGIT    = ', J
PRINT*, 'THE ONES DIGIT  = ', K
END
```

Ans 11.

```
      REAL R, PI, SAREA, VOLUME
      READ*, R
      PI = 3.14159
      SAREA = 4 * PI * R ** 2
      VOLUME = 4.0 / 3.0 * PI * R ** 3
      PRINT*,'RADIUS = ', R
      PRINT*, 'AREA = ', SAREA
      PRINT*, 'VOLUME = ', VOLUME
      END
```

Ans 12.

2 * X + Y / 2
((A + B) / (A - B)) ** 0.5
R ** 3 / 3.0 - A * C ** (3.0 / 4.0) / (2 * B)
1 / (1 / R1 + 1 / R2 + 1 / R3)
B + X * Y / (C + D) + 2
2 * A + C ** (-6)
(A + B ** (1.0 / 4.0)) / (2 / (A **2 + 5)) - 1

Ans 13.

A * B * C / (X * Y) ** 2
((A + B) ** 2 + (3 * C)** 3) ** (A / B)
(A - B + C) + D * E
(C * X) ** ((2 - A) * B)
-B + (B ** 2 - 4 * A * C) ** 0.05

Ans 14.

```
      INTEGER  SECNDS , MINTS , HOURS , QUAN
      READ*, QUAN
      HOURS  = QUAN / 3600
      QUAN   = QUAN - HOURS * 3600
      MINTS  = QUAN / 60
      SECNDS = QUAN - MINTS * 60
      PRINT*, HOURS,'HOURS',MINTS,'MINUTES',SECNDS,'SECONDS'
      END
```

Ans 15.

```
      INTEGER K1, K2
      READ*, K1 , K2
      PRINT*, K1 , K2
      READ*, K1 , K1 , K2
      PRINT*, K1 , K2
      READ*, K1 , K1 , K1 , K2
      PRINT*, K1 , K2
      END
```

Ans 16.

i) 0
ii)

```
      READ*, K1
      READ*, K2
      READ*, K3 , K4 , K5
```

Ans 17.

1. F2. T 3. T 4. T 5. T 6. T
7. F8. T 9. F 10. F 11. F

3 SELECTION CONSTRUCTS

Selection constructs are used to select between blocks of statements depending on certain *conditions*. Each condition is a logical expression (section 2.4.3) . In FORTRAN, the **IF** statement is used to represent selection constructs. This chapter introduces four types of **IF** constructs: **IF-ELSE**, **IF**, **IF-ELSEIF**, and the **simple IF** constructs.

3.1 IF-ELSE Construct

3.1.1 Definition

The general form of the **IF-ELSE** construct is as follows:

```
IF (condition) THEN
    BLOCK1
ELSE
    BLOCK2
ENDIF
```

where *condition* is a logical expression that evaluates either to .TRUE. or .FALSE.. *BLOCK1* and *BLOCK2* consist of one or more FORTRAN statements. If a block contains more than one statement, each statement must be in a separate line. Statements of *BLOCK1* and *BLOCK2* may be any FORTRAN statements including **IF** statements, assignment statements, input/output statements, repetition statements, transfer (**GOTO**) statements and others. In the above construct, *BLOCK1* will be executed if *condition* has the value .TRUE.. If the value of *condition* is .FALSE., *BLOCK2* will be executed. In either case, only one block is executed. After executing one of the two blocks, control transfers to the first statement after the **ENDIF**.

The keywords **IF** and **THEN** should appear in the same line along with the condition. The condition should be between parentheses. The keyword **ELSE** should appear in a separate line and the construct must end with the keyword **ENDIF** in a separate line. BLOCK1 and BLOCK2 begin, in a new line, after the column in which ***IF***, ***ELSE*** and ***ENDIF*** appear. This is known as *indentation*. Indentation is not a must but it increases program readability.

3.1.2 Examples on the IF-ELSE Construct

The following examples illustrate the **IF-ELSE** construct.

Example 1: *Write a FORTRAN program that reads two integer numbers and prints the maximum.*

Solution:

```
      INTEGER NUM1, NUM2
      READ*, NUM1, NUM2
      PRINT*, 'INPUT: ', NUM1, NUM2
      IF (NUM1 .GT. NUM2) THEN
          PRINT*, 'MAXIMUM IS ', NUM1
      ELSE
          PRINT*, 'MAXIMUM IS ', NUM2
      ENDIF
      END
```

Example 2: *What will be the output of the previous program if the input line is as follows*:

```
    347        -670
```

Solution:

The output will be as follows:

```
INPUT: 347 -670
MAXIMUM IS 347
```

Example 3: *Write a FORTRAN program that reads an integer number and finds out if the number is even or odd. The program should print a proper message.*

Solution:

```
      INTEGER K
      READ*, K
      PRINT*, 'INPUT: ', K
      IF(K / 2 * 2 .EQ. K) THEN
          PRINT*, 'EVEN'
      ELSE
          PRINT*, 'ODD'
      ENDIF
      END
```

Example 4: *What will be the output of the previous program if the input is as follows:*

```
     79
```

Solution: The output will be as follows:

```
INPUT: 79
ODD
```

3.2 IF Construct

3.2.1 Definition

We sometimes require a block of statements to be executed, if a *condition* is .TRUE.. Otherwise, if the condition is .FALSE., no statements must be executed. In this case we use the **IF** construct. The **IF** construct has the following general form:

```
      IF (condition) THEN
         BLOCK
      ENDIF
```

where *condition* is a logical expression that evaluates to either .TRUE. or .FALSE.. *BLOCK* consists of one or more FORTRAN statements. A statement in the *BLOCK* may be any FORTRAN statement including the **IF** statement. *BLOCK* will be executed if the *condition* evaluates to .TRUE. . The control then transfers to the first statement after the **ENDIF**. If the *condition* evaluates to .FALSE., control transfers to the first

statement after **ENDIF**, without executing any statement inside the **IF** construct. The keywords **IF** and **THEN** should appear in the same line along with the condition. The *condition* must be between parentheses. As was the case in the previous **IF** construct, indentation is not a must but it increases readability.

3.2.2 Examples on the IF Construct

The following examples illustrate the **IF** construct.

Example 1: *Write a FORTRAN program that reads a grade. If the grade is not zero, the program must add 2 points to the grade. Then, the new grade should be printed.*

Solution:

```
      REAL GRADE
      READ*, GRADE
      PRINT*, 'ORIGINAL GRADE IS', GRADE
      IF (GRADE .GT. 0) THEN
          GRADE = GRADE + 2.0
          PRINT*, 'SCALED GRADE IS ', GRADE
      ENDIF
      END
```

Example 2: *What will be the output of the previous program if the input line is as follows:*

```
           7.5
```

Solution: The output is as follows:

```
ORIGINAL GRADE IS 7.5000000
SCALED GRADE IS 9.5000000
```

Example 3: *What will be the output of the program of the previous example if the input line is as follows:*

```
           0.0
```

Solution: The output is as follows:

```
ORIGINAL GRADE IS 0.0000000
```

Example 4: *Write a FORTRAN program that reads a student ID and his GPA. If the GPA is greater than or equal to 3.0, the program should print the message 'HONOR'.*

Solution:

```
      REAL GPA
      INTEGER ID
      READ*, ID, GPA
      PRINT*, 'INPUT: ', ID, GPA
      IF (GPA .GE. 3.0) THEN
          PRINT*, 'HONOR'
      ENDIF
      END
```

Example 5: *What will be the output of the previous program if the input line is as follows:*

```
           918962        2.90
```

Solution: The output is as follows: (Note: Since the condition in the **IF** statement is not satisfied, the message HONOR is not printed.)

```
INPUT: 918962 2.9000000
```

3.3 IF-ELSEIF Construct

3.3.1 Definition

Assume you are given a numeric grade. A letter grade is to be printed based on the standard criteria i.e. if the grade is greater than or equal to 90, letter A is to be printed; if the grade is greater than or equal to 80, letter B is to be printed and so on . In such a case, we must use several **IF** statements. Instead FORTRAN provides a construct that can select a single block of statements from several blocks based on different conditions. This construct is the **IF-ELSEIF** construct and it is used when a single block is to be executed from a choice of several blocks. The general form of this construct is as follows:

```
IF (condition-1) THEN
    BLOCK1
ELSEIF (condition-2) THEN
    BLOCK2
ELSEIF (condition-3) THEN
    BLOCK3
..
..
ELSEIF (condition-n) THEN
    BLOCKn
ELSE
    BLOCKn+1
ENDIF
```

where *condition-i* for i = 1, 2, 3, ..., n is a logical expression that evaluates to either .TRUE. or .FALSE.. *BLOCKi* consists of one or more FORTRAN statements. The statements in each BLOCK are FORTRAN statements including any type of **IF** constructs. In the **IF-ELSEIF** construct *BLOCK1* will be executed if *condition-1* evaluates to .TRUE.. The control then transfers to the first statement after the **ENDIF**. If *condition-1* evaluates to .FALSE., condition-2 is examined. If *condition-2* evaluates to .TRUE., *BLOCK2* will be executed and control transfers to the first statement after the **ENDIF**. Otherwise, *condition-3* is examined and if it evaluates to .TRUE., *BLOCK3* will be executed and control transfers to the first statement after the **ENDIF**. The same action is applied to the rest of the **ELSEIF** clauses until a *condition* evaluates to .TRUE.. If all *conditions* evaluate to .FALSE., the **ELSE** part, i.e. *BLOCKn+1*, is executed and control passes to the first statement after the **ENDIF**. The **ELSE** part is optional. If all *conditions* are .FALSE and there is no **ELSE** part, control passes to the first statement after the **ENDIF**, without executing any of the blocks. In summary, the block corresponding to first condition that evaluates to .TRUE. is the only block that is executed. In case, no condition evaluates to .TRUE., the block corresponding to the **ELSE** part, if present, is executed. Indentation is not a must but it increases readability.

3.3.2 Examples on the IF-ELSEIF Construct

The following examples illustrate the **IF-ELSEIF** construct

Example 1: *Write a FORTRAN program that reads a student ID and his GPA out of 4.0. The program should print a message according to the following:*

Condition	Message

GPA ≥ 3.5	EXCELLENT
3.5 > GPA ≥ 3.0	VERY GOOD
3.0 > GPA ≥ 2.5	GOOD
2.5 > GPA ≥ 2.0	FAIR
GPA < 2.0	POOR

Solution:

```
      REAL GPA
      INTEGER ID
      CHARACTER*10 STATE
      READ*, ID, GPA
      PRINT*, 'INPUT: ', ID, GPA
      IF (GPA .GE. 3.5) THEN
          STATE = 'EXCELLENT'
      ELSEIF (GPA .GE. 3.0) THEN
          STATE = 'VERY GOOD'
      ELSEIF (GPA .GE. 2.5) THEN
          STATE = 'GOOD'
      ELSEIF (GPA .GE. 2.0) THEN
          STATE = 'FAIR'
      ELSE
          STATE = 'POOR'
      ENDIF
      PRINT*, ID,' ', STATE
      END
```

Another Solution:

```
      REAL GPA
      INTEGER ID
      CHARACTER*10 STATE
      READ*, ID, GPA
      PRINT*, 'INPUT: ', ID, GPA
      IF (GPA .LT. 2.0) THEN
          STATE = 'POOR'
      ELSEIF (GPA .LT. 2.5) THEN
          STATE = 'FAIR'
      ELSEIF (GPA .LT. 3.0) THEN
          STATE = 'GOOD'
      ELSEIF (GPA .LT. 3.5) THEN
          STATE = 'VERY GOOD'
      ELSE
          STATE = 'EXCELLENT'
      ENDIF
      PRINT*, ID,' ', STATE
      END
```

Example 2: *The following table has two columns, the first column gives the sample input to the previous program and the second column shows the expected output.*

Solution:

Sample Input	Expected Output
927322 2.3	INPUT: 927322 2.3000000 927322 FAIR
922822 3.4	INPUT: 922822 3.4000000 922822 VERY GOOD
848000 1.8	INPUT: 848000 1.8000000 848000 POOR

899999 3.7	INPUT: 899999 3.7000000 899999 EXCELLENT
912877 2.0	INPUT: 912877 2.0000000 912877 FAIR
943245 -2.0	INPUT: 943245 -2.0000000 943245 POOR
942221 7.0	INPUT: 942221 7.0000000 942221 EXCELLENT

Example 3: *Use **IF-ELSE** constructs to write a FORTRAN program that reads a student ID and his GPA out of 4.0. The program should print a message according to the following:*

Condition	Message
GPA ≥ 3.5	EXCELLENT
3.5 > GPA ≥ 3.0	VERY GOOD
3.0 > GPA ≥ 2.5	GOOD
2.5 > GPA ≥ 2.0	FAIR
GPA < 2.0	POOR

Solution:

```
      INTEGER ID
      REAL GPA
      CHARACTER*10 STATE
      READ*, ID, GPA
      PRINT*, 'INPUT: ', ID, GPA
      IF (GPA .GE. 3.5) THEN
          STATE = 'EXCELLENT'
      ELSE
          IF (GPA .GE. 3.0) THEN
              STATE = 'VERY GOOD'
          ELSE
              IF (GPA .GE. 2.5) THEN
                  STATE = 'GOOD'
              ELSE
                  IF (GPA .GE. 2.0) THEN
                      STATE = 'FAIR'
                  ELSE
                      STATE = 'POOR'
                  ENDIF
              ENDIF
          ENDIF
      ENDIF
      PRINT*, ID,' ', STATE
      END
```

Example 4: *Rewrite the above program using **IF** constructs.*

Solution:

```
      INTEGER ID
      REAL GPA
      CHARACTER*10 STATE
      READ*, ID, GPA
      PRINT*, 'INPUT: ', ID, GPA
      IF (GPA .GE. 3.5) THEN
          STATE = 'EXCELLENT'
      ENDIF
      IF (GPA .GE. 3.0 .AND. GPA .LT. 3.5) THEN
          STATE = 'VERY GOOD'
      ENDIF
      IF (GPA .GE. 2.5 .AND. GPA .LT. 3.0) THEN
          STATE = 'GOOD'
      ENDIF
      IF (GPA .GE. 2.0 .AND. GPA .LT. 2.5) THEN
          STATE = 'FAIR'
      ENDIF
      IF (GPA .LT. 2.0) THEN
          STATE = 'POOR'
      ENDIF
      PRINT*, ID,' ', STATE
      END
```

Example 5: *Write a FORTRAN program that reads three integer numbers and finds and prints the maximum. Use* ***IF-ELSEIF*** *construct.*

Solution:

```
      INTEGER X1, X2, X3, MAXIM
      READ*, X1, X2, X3
      IF (X1 .GE. X2 .AND. X1 .GE. X3) THEN
          MAXIM = X1
      ELSEIF (X2 .GE. X3) THEN
          MAXIM = X2
      ELSE
          MAXIM = X3
      ENDIF
      PRINT*, 'THE NUMBERS ARE ', X1, X2, X3
      PRINT*, 'THE MAXIMUM OF THE THREE NUMBERS = ',MAXIM
      END
```

3.4 Simple IF Construct

3.4.1 Definition

Sometimes a single FORTRAN statement must be executed if a *condition* is .TRUE.. In such cases, we may use a simple form of the **IF** construct which is written in a single line. It has the following general form:

```
      IF (condition) STATEMENT
```

where *condition* evaluates to .TRUE. or .FALSE. and STATEMENT is a simple FORTRAN statement such as an assignment statement, a **READ** statement, a **PRINT** statement, a **GOTO** statement, or a **STOP** statement. If *condition* evaluates to .TRUE., *STATEMENT* is executed and the control passes to the next statement. If *condition* is .FALSE., *STATEMENT* is not executed and the control transfers to the next statement.

3.4.2 Examples on the Simple IF Construct

The following examples illustrate the simple **IF** construct.

Example 1: *Use **simple IF** constructs to write a FORTRAN program that reads a student ID and his GPA out of 4.0. The program should print a message according to the following:*

Condition	Message
GPA ≥ 3.5	EXCELLENT
3.5 > GPA ≥ 3.0	VERY GOOD
3.0 > GPA ≥ 2.5	GOOD
2.5 > GPA ≥ 2.0	FAIR
GPA < 2.0	POOR

Solution:

```
      INTEGER ID
      REAL GPA
      CHARACTER*10 STATE
      READ*, ID, GPA
      PRINT*, 'INPUT: ', ID, GPA
      IF (GPA .GE. 3.5) STATE = 'EXCELLENT'
      IF (GPA .GE. 3.0 .AND. GPA .LT. 3.5) STATE = 'VERY GOOD'
      IF (GPA .GE. 2.5 .AND. GPA .LT. 3.0) STATE = 'GOOD'
      IF (GPA .GE. 2.0 .AND. GPA .LT. 2.5) STATE = 'FAIR'
      IF (GPA .LT. 2.0) STATE = 'POOR'
      PRINT*, ID,' ', STATE
      END
```

Example 2: *Write a FORTRAN program that reads three integer numbers and finds and prints the maximum. Use **simple IF** constructs.*

Solution:

```
      INTEGER X1, X2, X3, MAXIM
      READ*, X1, X2, X3
      PRINT*, 'THE NUMBERS ARE ', X1, X2, X3
      MAXIM = X1
      IF (X2 .GT. MAXIM) MAXIM = X2
      IF (X3 .GT. MAXIM) MAXIM = X3
      PRINT*, 'THE MAXIMUM OF THE THREE NUMBERS IS ', MAXIM
      END
```

Another Solution:

```
      INTEGER X1, X2, X3
      READ*, X1, X2, X3
      PRINT*, 'THE NUMBERS ARE ', X1, X2, X3
      IF (X1 .GE. X2 .AND. X1 .GE. X3) PRINT*, 'MAXIMUM IS ', X1
      IF (X2 .GE. X1 .AND. X2 .GE. X3) PRINT*, 'MAXIMUM IS ', X2
      IF (X3 .GE. X1 .AND. X3 .GE. X2) PRINT*, 'MAXIMUM IS ', X3
      END
```

3.5 Exercises

1. What will be printed by the following programs? If an error message is generated, which statement causes the error?

```
1.      INTEGER N, M
        N = 15
        M = 10
        IF (M.GE.N) THEN
            M = M + 1
            IF (N.EQ.M) THEN
                N = N + 5
            ELSEIF (N.GT.0) THEN
                N = N + 10
            ENDIF
            M = M - 1
        ENDIF
        M = M - 1
        PRINT*, M, N
        END
```

```
2.      LOGICAL A, B
        INTEGER EX1, EX2, EX3
        READ*, EX1, EX2, EX3
        A = EX1.LE.EX2.OR.EX2.LE.EX3
        B = EX2+2.GT.EX3*2
        IF (B) THEN
            A = .NOT. A
        ELSE
            B = .NOT. B
        ENDIF
        PRINT*, A, B
        END
```

Assume the input for the program is:

```
40 35 20
```

```
3.      REAL A, B, C
        A = -3
        B = -4.0
        IF (.NOT. A.LT.B) THEN
            C = A - B
        ELSE
            C = A * B
        ENDIF
        PRINT*, C
        END
```

```
4.      REAL A,B
        INTEGER I
        READ*, A, I, B
        IF (A.LT.3.0) THEN
            PRINT*, A+I
            IF (B.LT.2.5) THEN
                PRINT*, B**I
            ENDIF
        ELSE
            PRINT*, A*B*I
        ENDIF
        END
```

Assume the input for the program is:

```
2.5 2 2.5
```

```
5.      INTEGER A, B, C
        READ*, A, B, C
        IF (A.GT.B) THEN
            IF (B.LT.C) THEN
                PRINT*, B
            ELSE
                PRINT*, C
            ENDIF
        ELSE
            PRINT*, A
        ENDIF
        PRINT*, A, B, C
        END
```

Assume the input for the program is:

```
-2 -4 -3
```

```
6.      LOGICAL A,B
        INTEGER K1, K2
        K1 = 10
        K2 = 12
        A = K1.LT.K2
        B = .TRUE.
        IF (A) B = .FALSE.
        PRINT*, A, B
        END
```

```
7.      EEAL A, B
        INTEGER K, L
        READ*, A, B, L, K
        IF (A .GT. B) THEN
            IF (A .LT. L/2) THEN
                PRINT*, 'THURSDAY'
            ELSE
                PRINT*, 'SUNDAY'
            ENDIF
        ELSE
            IF (K/4.GE.B-2) THEN
                PRINT*, 'MONDAY'
            ELSE
                PRINT*, 'TUESDAY'
            ENDIF
        ENDIF
        END
```

Assume the input for the program is:

```
3.0 3.0 4 6
```

```
8.      INTEGER RANKX, RANKY
        REAL X, Y
        READ*, X, Y
        IF (X.GT.Y) THEN
            RANKX = 1
            RANKY = 2
        ELSE
            RANKX = 2
            RANKY = 1
        ENDIF
        PRINT*, RANKX, RANKY
        END
```

Assume the input for the program is:

```
4.0 4.0
```

```
9.      INTEGER SALARY, BONUS, TOTAL
        INTEGER AGE, EXP
        READ*, IDNO, AGE, EXP, SALARY
        IF (AGE.GE.40 .OR. EXP.GT.10) THEN
            BONUS = SALARY/8 + 450.0
        ELSE
            BONUS = SALARY/10 + 350.0
        ENDIF
        TOTAL = SALARY + BONUS
        PRINT*, IDNO, BONUS, TOTAL
        END
```

Assume the input for the program is:

```
834567 38 12 40000
```

2. Write a FORTRAN program that reads the value of a real number (DELTA) . If the value of (DELTA) is negative, then the program prints the message (NUMBER IS OUT OF RANGE) . Otherwise, the program computes the square root of (DELTA) and prints the result.
3. Write a complete FORTRAN program that reads the variables A, B and C, then computes the value of X where:

$$x = \frac{\sqrt{a - b + 2a^2}}{c}$$

   The program should take care of the problem of dividing by zero or getting a negative number under the square root. The program should print the appropriate messages accordingly (i.e. "DIVIDING BY ZERO", or, "NEGATIVE NUMBER UNDER SQUARE ROOT"). If both errors occur, the program should print both messages. If no error occurs, the program should print the value of X.
4. Consider the following structure where A is a real variable :

```
        IF (A.LE.10) THEN
            IF (A.LT.5) THEN
                PRINT*, 'AAA'
            ELSEIF (A.LT.4) THEN
                PRINT*, 'BBB'
            ELSEIF (A.GT.6) THEN
                PRINT*, 'CCC'
            ELSE
                PRINT*, 'DDD'
            ENDIF
        ENDIF
```

The condition that causes AAA to be printed is (A < 5) .

   1. What is the condition that will cause BBB to be printed?
   2. What is the condition that will cause CCC to be printed?
   3. What is the condition that will cause DDD to be printed?

5. Assume that V1 and V2 are **LOGICAL** variables and STATEMENT1, STATEMENT2 and STATEMENT3 are any valid FORTRAN statements. Given the following **IF**-structure:

```
      IF (V1) THEN
          STATEMENT1
      ELSEIF (.NOT. V2) THEN
          STATEMENT2
      ELSE
          STATEMENT3
      ENDIF
```

choose the equivalent structure(s) from the following:

```
I.    IF (.NOT. V1) THEN
          IF (.NOT. V2) THEN
              STATEMENT2
          ELSE
              STATEMENT3
          ENDIF
      ELSE
          STATEMENT1
      ENDIF
```

```
II.   IF (.NOT.V2) THEN
          STATEMENT2
      ELSEIF (V1) THEN
          STATEMENT1
      ELSE
          STATEMENT3
      ENDIF
```

```
III.  IF (V1) THEN
          STATEMENT1
      ELSE
          IF (.NOT. V2) THEN
              STATEMENT2
          ELSE
              STATEMENT3
          ENDIF
      ENDIF
```

6. Consider the following FORTRAN 77 program segment :

```
      IF (A.GT.B .OR. A.EQ.B) PRINT*, A
```

Which one(s) of the following segments is(are) equivalent to the above?

```
I.    IF (A.GE.B) THEN
          PRINT*, A
      ENDIF
```

```
II.   IF (A.GT.B .AND. A.EQ.B) THEN
          PRINT*, A
      ENDIF
```

```
III.  IF (.NOT. (A.LT.B) ) THEN
          PRINT*, A
      ENDIF
```

7. What values of X cause the value of A to be changed in the following statement?

```
       IF (X.LT.3.0 .AND. 7.0.LT.X) A = A + 1
```

8. Write a complete FORTRAN program that reads a real number into a real variable NUM. If NUM is non-zero prints the value of its reciprocal (1/NUM) . Otherwise, prints the message "RECIPROCAL NOT DEFINED".

9. Give the FORTRAN statements that perform the steps indicated below :
    1. If y is not positive, and 3.5>x>1.5 then print the value of y.
    2. If time is greater than 15.0, increment time by 1.0.
    3. If dist is less than 50.0 and time is greater than 10.0, increment time by 2.0. Otherwise, increment time by 2.5.
    4. Interchange the value of a and b (i.e. a gets the value of b and b gets the old value of a, if both a and b are positive.
    5. If grade is greater than or equal to 4.0 then increment a by 1.0. If grade is greater than or equal to 3.0 but less than 4.0 then increment b by 1.0. If grade is greater than or equal to 2.0 but less than 3.0 then increment c by 1.0, otherwise increment d by 1.0.
10. Assume COND1, COND2, COND3, and COND4 are FORTRAN logical expressions. Consider the following program segment.

```
      IF (COND1) THEN
          IF (COND2) THEN
              PRINT*,'RIYADH'
          ELSE
              IF (COND3) THEN
                  PRINT*, 'JEDDAH'
              ELSE
                  PRINT*, 'KHOBAR'
              ENDIF
          ENDIF
      ELSEIF (COND4) THEN
          PRINT*, 'TAIF'
      ELSE
          PRINT*, 'DHAHRAN'
      ENDIF
```

If the output of the above segment is

```
KHOBAR
```

What are the logical values of COND1, COND2, COND3 and COND4?

11. Write a program that reads an integer number N and prints YES if the following expression is satisfied.

$$0 < N < 100 \quad \text{and} \quad N > 50$$

12. Write a FORTRAN program which reads an integer number between 10 and 99 and prints the number reversed. For example, if the number read is 87, then the program output must be 78.
13. Consider the following **IF** statements carefully. Each of Blocks A, B, C, D, E, F, G, H represents a block of FORTRAN statements.

```
I.    IF (CONDITION) THEN
          A
      ELSE
          B
      ENDIF
      C
      END
```

```
II.    IF (CONDITION) D
          E
       END
```

```
III.   IF (CONDITION) THEN
          F
       ELSEIF (CONDITION) THEN
          G
       ELSE
          H
       ENDIF
       END
```

Assuming that X has a value 0.0, which block(s) are executed in program segments (i) , (ii) and (iii) , if CONDITION is the expression listed below?

i) X.GE.0

ii) X.LE.0

iii) X.GT.0

iv) X.LT.0

14. Write a FORTRAN program that reads three integers A, B, and C. The program checks if A, B, and C are in increasing order or in decreasing order and prints an appropriate message. If the integers are not in order, then the program prints UNORDERED. For example, if the input is

```
3 4 5
```

The program prints

```
INCREASING ORDER
```

15. A year between 1900 and 1999 is a LEAP year if it is divisible by 4 and not by 100 or if it is divisible by 400. Write a FORTRAN program which will read a year and determine whether the year is a LEAP or NOT. The program should print one of the following messages accordingly:

```
THE YEAR IS OUT OF RANGE
```

or

```
THE YEAR IS A LEAP YEAR
```

or

```
THE YEAR IS NOT A LEAP YEAR
```

16. Consider the following **IF** statement:

```
       IF (X.GE.Y) THEN
          PRINT*, X
       ELSE
          PRINT*, Y
       ENDIF
```

In each of the following program segments, fill the spaces by relational or logical operators (.EQ., .NE., .LT., LE., .GT., .GE., .AND., .OR., .NOT.) such that each of the program segments below gives the same output as the program segment above.

```
I.     IF (X ------ Y) PRINT*, X
       IF (X ------ Y) PRINT*, Y
```

```
II.    IF (X.GT.Y) THEN
           PRINT*, X
       ELSEIF (X ----- Y) THEN
           PRINT*, X
       ELSE
           PRINT*, Y
       ENDIF
```

```
III.   IF (X ----- Y ----- X.EQ.Y) THEN
           PRINT*, X
       ELSE
           PRINT*, Y
       ENDIF
```

17. Write a program that reads any two positive integer numbers and finds the larger of the two numbers. The program then checks if the larger number is divisible by the smaller one. If it is divisible the program should print the word DIVISIBLE. If the larger number is not divisible by the smaller number, the program checks if both numbers are odd and prints BOTH ODD.

3.6 Solutions to Exercises

Ans 1.

```
 9  15
 F   T
 1.0
  4.5
          -4
          -2  -4  -3
 T  F
 MONDAY
 2   1
 834567   5450   45450
```

Ans 2.

```
      READ*, DELTA
      IF (DELTA .LT. 0.0) THEN
          PRINT*, 'NUMBER IS OUT OF RANGE'
      ELSE
          PRINT*, DELTA ** 0.5
      ENDIF
      END
```

Ans 3.

```
      READ*, A , B , C
      D = A - B + 2 * A ** 3
      IF (C .EQ. 0 .OR. D .LT. 0) THEN
          IF (C .EQ. 0) PRINT*, 'DIVISION BY ZERO'
          IF (D .LT. 0) PRINT*, 'NEGATIVE UNDER SQUARE ROOT'
      ELSE
          X =  D ** 0.5/ C
          PRINT*, X
      ENDIF
      END
```

Ans 4.

1. Never 2. $10 \geq A > 6$ 3. $6 \geq A \geq 5$

Ans 5.

I and III

Ans 6.

I and III

Ans 7.

No values for X,

A can't be changed according to this condition

Ans 8.

```
      REAL NUM
      READ* , NUM
      IF (NUM .NE. 0) THEN
          PRINT*, 1 / NUM
      ELSE
          PRINT*, 'RECIPROCAL NOT DEFINED'
      ENDIF
      END
```

Ans 9.

1.

```
IF( Y .LT. 0 .AND. (X .GT. 1.5 .AND. X .LT. 3.5))PRINT*,Y
```

2.

```
IF( TIME .GT. 15.0 ) TIME = TIME + 1
```

3.

```
      IF( DIST .LT. 50.0 .AND. TIME .GT. 10.0 ) THEN
          TIME = TIME + 2.0
      ELSE
          TIME = TIME + 2.5
      ENDIF
```

4.

```
      IF( A .GT. 0 .AND. B .GT. 0 ) THEN
          T = A
          A = B
          B = T
      ENDIF
```

5.

```
      IF( GRADE .GE. 4.0 ) THEN
          A = A + 1.0
      ELSEIF( GRADE .GE. 3.0 ) THEN
          B = B + 1.0
      ELSEIF( GRADE .GE. 2.0 ) THEN
          C = C + 1.0
      ELSE
          D = D + 1.0
      ENDIF
```

Ans 10.

COND1 : T
COND2 : F
COND3 : F
COND4 : Can be T or F

Ans 11.

```
      READ*, N
      IF (N .GT. 50 .AND. N .LT. 100) THEN
          PRINT*,'YES'
      ENDIF
      END
```

Ans 12.

```
      INTEGER REV
      READ*, K
      IF (K .GT. 10 .AND. K .LE. 99) THEN
          REV = (K - K / 10 * 10) * 10 + K / 10
          PRINT*, REV
      ELSE
          PRINT*, 'NUMBER IS OUT OF RANGE'
      ENDIF
      END
```

Ans 13.

X .GE. 0	i) A , C	ii) D , E	iii) F
X .LE. 0	i) A , C	ii) D , E	iii) F
X .GT. 0	i) B , C	ii) E	iii) H
X .LT. 0	i) B , C	ii) E	iii) H

Ans 14.

```
      READ*, A , B , C
      IF (A .GE. B .AND. B .GE. C) THEN
          PRINT*, 'DECREASING ORDER'
      ELSEIF(A .LE. B .AND. B .LE. C) THEN
          PRINT*, 'INCREASING ORDER'
      ELSE
          PRINT*, 'UNORDERD'
      ENDIF
      END
```

Ans 15.

```
      INTEGER Y
      READ*, Y
      IF(Y .GE. 1900 .AND. Y .LE. 1999) THEN
        IF(Y/4*4.EQ.Y.AND.Y/100*100.NE.Y.OR.Y/400*400.EQ.Y) THEN
          PRINT*, 'THE YEAR IS A LEAP YEAR'
        ELSE
          PRINT*, 'THE YEAR IS NOT A LEAP YEAR'
        ENDIF
      ELSE
          PRINT*, 'THE YEAR IS OUT OF RANGE'
      ENDIF
      END
```

Ans 16.

i) X .GE. Y ii) X .EQ. Y iii) X .GT. Y .OR.X .LT. Y

Ans 17.

```
      READ*, M , N
      IF(M .GE. N) THEN
          MAX = M
          MIN = N
      ELSE
          MAX = N
          MIN = M
      ENDIF
      IF(MAX / MIN * MIN .EQ. MAX) THEN
          PRINT*, 'DIVISABLE'
      ELSE
          IF(MAX/2*2 .NE. MAX .AND. MIN/2*2 .NE. MIN) THEN
              PRINT*,'BOTH ODD'
          ENDIF
      ENDIF
      END
```

4 TOP DOWN DESIGN

Many problems consist of a number of tasks. One good technique in solving such problems is to identify the tasks, decompose each task into sub-tasks and solve these sub-tasks by smaller and simpler solutions. Ultimately, the main tasks and the sub-tasks are converted to program code. In this chapter, we introduce the top down design technique based on problem decomposition and the means to implement such a technique.

4.1 Basic Concepts of Top Down Design

Top down design is a technique that reduces the complexity of large problems. The technique is based on the divide-and-conquer strategy, wherein the problem tasks are divided into sub-tasks repetitively. The division of tasks stops when the sub-tasks are relatively easy to program. The terms *successive refinement or step-wise refinement* also refer to the top-down design technique.

In FORTRAN, each sub-task can be implemented by a separate module. FORTRAN uses two types of program modules, *subroutines* and *functions*. These modules are also called *subprograms*. A typical FORTRAN program consists of a main program with several subprograms. Each subprogram represents a sub-task in the top down design solution.

The top down design process has many advantages:

1. The subprograms can be independently implemented and tested.
2. Subprograms developed by others can be used. For example, a huge library of FORTRAN subprograms known as IMSL (International Mathematical and Statistical Library) is available. The IMSL library has efficient, well tested subprograms for common problems in matrix manipulation, algebraic equations, statistical computations, .. etc.
3. The size of the program is reduced, since identical code segments in the main program are replaced by a single subprogram.

4.2 Subprogram Terminology

There are several new terms with which we should be familiar with while using subprograms. The program file usually consists of a program called the *main program* and all the associated subprograms. These subprograms may appear before or after the main program. A subprogram is *called* or *invoked* by another subprogram or the main

program. The calling program passes information to the subprogram through *arguments* or *parameters*. The subprogram returns information to the calling program. In the case of a function, the information which is a single value, is returned as the value of the function name. In the case of a subroutine, the information is returned through some or all the arguments. The arguments that appear in the description of the subprogram are called *dummy* arguments and those that appear in the calling statement are called *actual* arguments. Every subprogram consists of a *header* followed by a *body*. The subprogram body has a statement called the ***RETURN*** statement to return execution control to the calling program. There may be more than one **RETURN** statements in a subprogram. A subprogram ends with an **END** statement.

4.3 Function Subprograms

A function subprogram is the description of a function consisting of several statements. The subprogram computes a single value and stores that value in the function name. A function subprogram consists of a function header and a function body.

4.3.1 Function Header

The *function header* is the first statement of the function and has the following format:

type **FUNCTION** fname (***a list of arguments***)

where

type is the type for the function name (**REAL**, **INTEGER** ..);

fname is the name of the function;

a list of arguments is the optional list of dummy arguments.

If the *type* of the function is not specified, the function type is assumed as either **INTEGER** or **REAL**, as in the case of variables. The rules that apply in naming a variable also apply to function names. If there are no arguments to a function, then the empty parentheses () appear with the function name.

4.3.2 Function Body

The *function body* is similar to a FORTRAN program. It consists of declaration statements, if any, in the beginning, followed by executable statements. Each function body must end with an **END** statement. The **RETURN** statement must appear in the function body at least once. This statement is used to transfer control from the function back to the calling program. The function name should be assigned a value in the function body. A *typical* layout of a function is as follows:

```
      TYPE FUNCTION FNAME (A LIST OF DUMMY ARGUMENTS)
      DECLARATION OF DUMMY ARGUMENTS AND VARIABLES TO BE USED IN THE
      FUNCTION

          EXECUTABLE STATEMENTS
      ..
      ..
      FNAME = EXPRESSION
      ..
      ..
      RETURN
      END
```

4.3.3 *Examples on function subprograms*

Example 1: *Write a real function VOLUME that computes the volume of a sphere* $(4/3\pi r^3)$ *given its radius.*

Solution:

```
      REAL FUNCTION VOLUME(RADIUS)
      REAL RADIUS, PI
      PI = 3.14159
      VOLUME = 4.0 / 3.0 * PI * RADIUS ** 3
      RETURN
      END
```

Example 2: *Write a logical function ORDER that checks whether three different integer numbers are ordered in increasing or decreasing order.*

Solution:

```
      LOGICAL FUNCTION ORDER(X, Y, Z)
      INTEGER X, Y, Z
      LOGICAL INC, DEC
      DEC = X .GT. Y .AND. Y .GT. Z
      INC = X .LT. Y .AND. Y .LT. Z
      ORDER = INC .OR. DEC
      RETURN
      END
```

Example 3: *Write a function subprogram to evaluate the function f(x) defined below.*

$f(x) = 2x^2 + 4x + 2$ if $x < 5$

$f(x) = 0$ if $x = 5$

$f(x) = 3x + 1$ if $x > 5$

Solution:

```
      FUNCTION F(X)
      REAL F, X
      IF (X .LT. 5) THEN
           F = 2 * X ** 2 + 4 * X + 2
      ELSEIF (X .EQ. 5) THEN
           F = 0
      ELSE
           F = 3 * X + 1
      ENDIF
      RETURN
      END
```

4.3.4 Function Call

Let us consider a program consisting of a main program and a function subprogram. The execution of the program begins with the main program. For each call to a function, control is transferred to the function. After the function is executed, the **RETURN** statement ensures that control is transferred back to the calling program. The execution of the main program then resumes at the location the function is called.

Example: *In the following two tables, correct and incorrect function calls to the functions defined in Examples 1, 2 and 3 are given. We assume that in the calling*

program the function names VOLUME, F are declared as **REAL**, *and ORDER as* **LOGICAL**. *We also assume A = 5.0, B = 21.0, where A and B are real numbers:*

Examples of correct function calls:

Function Call	Function Value
ORDER(3, 2, 4)	.FALSE.
ORDER(3, 4 * 3, 99)	.TRUE.
F(A)	0.0
F(3 + F(2.0))	64.0
VOLUME(B)	38808.0
F(A + B)	79.0

Examples of incorrect function calls:

Incorrect Function Call	Error Message
ORDER(3.0, 2, 4)	Argument 1 referenced as real but defined to be integer
F(3.2, 3.4)	More than one argument to function F
VOLUME(5)	Argument 1 referenced as integer but defined to be real

4.3.5 Function Rules

The following rules must be observed in writing programs with function subprograms:

- Actual and dummy arguments must match in type, order and number. The names of these arguments may or may not be the same.
- Actual arguments may be expressions, constants or variable names. Dummy arguments must be variable names and should never be expressions or constants.
- The type of the function name must be the same in both the calling program and the function description.
- The result from the function subprogram, to be returned to the calling program, should be stored in the function name.
- A return statement transfers control back to the calling program. Every function should have at least one return statement.
- The function may be placed either before or after the main program.
- A function is called or invoked as part of an expression.
- A FORTRAN function cannot call itself.

4.3.6 Complete *Examples on function subprograms*

Example 1: *The sum of three integer numbers: Write an integer function SUM to sum three integer numbers. Also write a main program to test the function SUM.*

Solution:

```
C MAIN PROGRAM
      INTEGER X, Y, Z, SUM
      READ*, X, Y, Z
      PRINT*, SUM (X, Y, Z)
      END

C FUNCTION SUBPROGRAM
      INTEGER FUNCTION SUM(A, B, C)
      INTEGER A, B, C
      SUM = A + B + C
      RETURN
      END
```

The execution starts with the reading of variables X, Y and Z in the main program. The execution of the expression SUM(X, Y, Z) transfers control to the function SUM. The value of the actual arguments X, Y and Z is passed to the dummy arguments A, B and C respectively. In the function SUM, execution begins with the first executable statement which computes the value of SUM. The return statement returns control to the main program. The print statement in the main program prints the value of SUM(X, Y, Z) and the execution ends. Assume that the input to the above program is as follows:

```
7 3 9
```

then the output of the program is

```
19
```

Example 2: *Reverse a Two Digit Number: A two digit integer number is to be reversed. A two digit number ranges between 10 and 99. Write a function that first checks if the number is a two digit number and then returns the number with the digits reversed. The function should return an error code -1 if the argument is not a two digit number. Write a main program to test the function.*

Solution:

The main program invokes function RVSNUM after reading a number. If the value returned from the function is -1, an error message is printed. Otherwise, the number and its reversed value are printed. Notice the use of two **RETURN** statements in the function.

```
      INTEGER FUNCTION RVSNUM(NUMBER)
      INTEGER NUMBER, RDIGIT, LDIGIT
      IF (NUMBER .LT. 10 .OR. NUMBER .GT.99) THEN
           RVSNUM = -1
           RETURN
       ENDIF
      LDIGIT = NUMBER / 10
      RDIGIT = NUMBER - LDIGIT / 10 * 10
      RVSNUM = RDIGIT * 10 + LDIGIT
      RETURN
      END

C     MAIN PROGRAM
      INTEGER NUMBER, RVSNUM, RNUM
      READ*, NUMBER
      RNUM = RVSNUM(NUMBER)
      IF (RNUM .EQ. -1) THEN
           PRINT*, 'INPUT ERROR : ', NUMBER
      ELSE
           PRINT*, 'ORIGINAL NUMBER IS ', NUMBER
           PRINT*, 'REVERSED NUMBER IS ', RNUM
      ENDIF
      END
```

If the input to this program is

```
78
```

then the output is:

```
ORIGINAL NUMBER IS 78
REVERSED NUMBER IS 87
```

If the input to this program is

```
123
```

then the output is:

```
INPUT ERROR : 123
```

Note that the actual arguments can be expressions. If the function is invoked with the statement **PRINT***, RVSNUM(4 * 6), the value 42 is printed.

4.4 Special Cases of Functions

There are special cases of functions that do not require subprogram description. These cases may be classified into two groups:

1. Intrinsic (built-in) Functions
2. Statement Functions

4.4.1 Intrinsic Functions

These are predefined functions that are available from the FORTRAN language. Certain functions, such as the trigonometric functions, are frequently encountered in programming. Instead of developing them repeatedly in each program, the language provides these functions. For example, MOD(M,N) is an intrinsic function that requires two integer arguments M and N. The result of the function MOD is an integer value representing the remainder when M is divided by N. A list of commonly used intrinsic functions is given below.

Function	Function Value	Comment
SQRT(X)	Square Root of X	X is a real argument
ABS(X)	Absolute Value of X	
SIN(X)	Sine of angle X	Angle is in radians
COS(X)	Cosine of angle X	Angle is in radians
TAN(X)	Tangent of angle X	Angle is in radians
EXP(X)	e raised to the power X	
LOG(X)	Natural Logarithm of X	X is real
LOG10(X)	Logarithm of X to base 10	X is real
INT(X)	Integer value of X	Converts a real to an integer
REAL(K)	Real value of K	Converts an integer to real
MOD(M, N)	Remainder of M/N	Modulo function

Common Intrinsic Functions

4.4.2 Statement Functions

In engineering and science applications, we frequently encounter functions that can be written in a single statement. For example, $f(x) = x + 2$ is a simple function. In such cases, FORTRAN allows us to write a statement function instead of writing a function subprogram. A *statement function* is defined in the beginning of a program after declaration statements. As a non-executable statement, it should appear before any executable statement. The general form of this statement is as follows:

fname (*a list of arguments*) = expression

where

fname is the name of the function;

a list of arguments is the optional list of dummy arguments; and

expression computes the function value.

The type of the statement function may be declared in the declaration statements. If the type of the function is not declared, it is implicitly defined.

4.4.2.1 Examples of statement functions:

Example 1: *Write a statement function to compute the area of a triangle, given its two sides and an angle.*

```
REAL AREA
AREA(SIDE1,SIDE2,ANGLE) = 0.5 * SIDE1 * SIDE2 * SIN (ANGLE)
```

Example 2: *Write a statement function to compute the total number of seconds, given the time in hours, minutes and seconds.*

Solution:

```
REAL TOTSEC
TOTSEC(HOUR,MINUTE,SECOND) = 3600 * HOUR +60 * MINUTE + SECOND
```

Example 3: *Write a statement function to compute the function* $f(x,y) = 3x^2 + 5xy$

Solution:

```
REAL F
F(X, Y) = 3 * X ** 2 + 5 * X * Y
```

Example 4: *Write a logical statement function to check if three different integer numbers are in increasing or decreasing order.*

Solution:

```
LOGICAL ORDER
ORDER(X,Y,Z) = X.GT.Y .AND. Y .GT. Z .OR. X.LT.Y .AND. Y.LT.Z
```

Example 5: *Temperature Conversion: Convert temperatures from one unit into another using statement functions. Write a main program to test the functions based on a code. If the code is 1, convert from centigrade to Fahrenheit. If code is 2, convert from Fahrenheit to centigrade. Otherwise, print an error message.*

Solution:

```
      REAL FTEMP, CTEMP, TEMP, VALUE
      INTEGER CODE
C FUNCTION FTEMP CONVERTS FROM CENTIGRADE TO FAHRENHEIT
      FTEMP(TEMP) = TEMP * 9 / 5 + 32
C FUNCTION CTEMP CONVERTS FROM FAHRENHEIT TO CENTIGRADE
      CTEMP(TEMP) = (TEMP - 32) * 5 / 9
      READ*, CODE, VALUE
      IF (CODE .EQ. 1) THEN
          PRINT*, VALUE , ' C = ' , FTEMP(VALUE), ' F'
      ELSEIF (CODE .EQ. 2) THEN
          PRINT*, VALUE , ' F = ' , CTEMP(VALUE), ' C'
      ELSE
          PRINT*, 'INPUT ERROR'
      ENDIF
      END
```

The statement functions FTEMP and CTEMP convert the argument value to Fahrenheit and centigrade respectively. The statement functions are placed immediately after the declaration statements. The variables CODE and VALUE are read. Based on the value of CODE, the appropriate statement function is invoked and the converted value is printed.

4.5 Subroutine Subprograms

A function produces a single result. In many instances, we would like a subprogram to produce more than one result. Subroutines are designed to produce *zero*, *one* or *many* results. A *subroutine* consists of a subroutine header and a body.

Subroutines differ from functions in the following ways:

- A subroutine may return a single value, many values, or no value.
- To return results, the subroutine uses the argument list; thus, the subroutine argument list consists of *input* arguments and *output* arguments.
- Since the results are returned through arguments, a subroutine name is used for documentation purposes only and does not specify a value.
- The general form of the subroutine header is as follows:

```
      SUBROUTINE SNAME (a list of dummy arguments)
```

where

SNAME is the name of the subroutine; and

a list of dummy arguments is optional.

- A subroutine is called or invoked by an executable statement, the **CALL** statement. The general form of the statement is as follows:

```
CALL SNAME (a list of actual arguments)
```

A subroutine is similar to a function in several ways. The subroutine actual and dummy arguments must match in type, number and order. At least one **RETURN** statement must be present to ensure transfer of control from a subroutine to the calling program.

Consider a program that consists of a subroutine and a main program. With each **CALL** statement in the main program, control is transferred to the subroutine. After the subroutine is executed, the **RETURN** statement ensures that control is transferred back to the calling program, to the statement immediately following the **CALL** statement.

4.5.1 *Examples on Subroutine Subprograms:*

Example 1: *Write a subroutine that exchanges the value of its two real arguments.*

Solution:

```
SUBROUTINE EXCHNG(NUM1, NUM2)
REAL NUM1, NUM2, TEMP
TEMP = NUM1
NUM1 = NUM2
NUM2 = TEMP
RETURN
END
```

The subroutine EXCHNG can be invoked using the **CALL** statement. An example illustrating a call to the subroutine EXCHNG is given below:

Assume the variables X, Y are declared as real in the calling program and have the values 3.0 and 8.0 respectively. The **CALL** statement

CALL EXCHNG(X, Y)

after execution will exchange the value of X and Y. During the execution of the **CALL** statement, the value of actual argument X is passed to the dummy argument NUM1 and the value of actual argument Y is passed to the dummy argument NUM2. At this point, the execution control is transferred to the subroutine EXCHNG. The subroutine exchanges the values of variables NUM1 and NUM2. When the **RETURN** statement of the subroutine is executed, the control returns to the calling program and the new values of variables NUM1 and NUM2 are passed back to the actual arguments X and Y respectively. Therefore, the new value of variable X would be 8.0 and the value of variable Y would be 3.0.

Example 2: *Write a subroutine that takes three different integer arguments X, Y and Z and returns the maximum and the minimum.*

Solution:

```
      SUBROUTINE MINMAX(X, Y, Z, MAX, MIN)
      INTEGER X, Y, Z, MAX, MIN
      MIN = X
      MAX = X
      IF (Y .GT. MAX) MAX = Y
      IF (Y .LT. MIN) MIN = Y
      IF (Z .GT. MAX) MAX = Z
      IF (Z .LT. MIN) MIN = Z
      RETURN
      END
```

Examples illustrating calls to the subroutine MINMAX is given below:

Example 3: *Assume the variables A, B, C are declared as integer in the calling program and have the values 4, 6, 8 respectively. Also assume that MAX and MIN are integer variables. After the following* ***CALL*** *statement*

```
      CALL MINMAX(A, B, C, MAX, MIN)
```

is executed, the value of MAX will be 8 (the maximum of variables A, B, C) and the value of MIN will be 4 (the minimum of variables A, B, C). Note that the names of the actual arguments may be similar or different from the corresponding dummy arguments but the type must be the same.

Example 4: *If the following* ***CALL*** *statement*

```
      CALL MINMAX(C+4, -1, A+B, MAX, MIN)
```

is executed, the value of MAX will be 12 and the value of MIN will be -1, since the first three actual arguments in the ***CALL*** *statement are evaluated to 12, -1 and 10 respectively. Note here that the actual arguments can be expressions.*

Example 5: *Sum and Average: Write a subroutine to sum three integers and compute their average. The subroutine should return the sum and average of the three numbers. Write a main program to test the subroutine.*

Solution:

```
C     MAIN PROGRAM
      INTEGER X, Y, Z, TOTAL
      REAL AVERAG
      READ*, X, Y, Z
      CALL SUBSUM (X, Y, Z, TOTAL, AVERAG)
      PRINT*, 'TOTAL IS ', TOTAL
      PRINT*, 'AVERAGE IS ' , AVERAG
      END
C      SUBROUTINE SUBPROGRAM

      SUBROUTINE SUBSUM(A, B, C, TOTAL, AVG)
      INTEGER A, B, C, TOTAL
      REAL AVG
      TOTAL = A + B + C
      AVG = TOTAL / 3.0
      RETURN
      END
```

The subroutine SUBSUM has three dummy arguments A, B, C and returns two results, the value of the fourth argument TOTAL and the fifth argument AVERAG. The **CALL** statement in the main program invokes the subroutine.

Arguments X, Y, Z, TOTAL and AVERAG in the main program are the actual arguments. Note that, before the subroutine is called, arguments X, Y and Z have values and arguments TOTAL and AVERAG do not have a value. Arguments A, B, C, TOTAL and AVERAG in the subprogram are the dummy arguments. X, Y and Z are input arguments, TOTAL and AVERAG are output arguments.

The execution starts with the reading of variables X, Y and Z in the main program. The execution of the **CALL** statement transfers control to the subroutine SUBSUM. The value of the actual arguments X, Y and Z is passed to the dummy arguments A, B and C respectively. Since TOTAL and AVERAG in the main program are not initialized, no value is passed to the corresponding arguments in the subprogram. In the subroutine SUBSUM, execution begins with the first executable statement which computes the value of argument TOTAL. The next statement computes the average of the three arguments. The return statement returns control to the main program.

The values of arguments A, B, C, TOTAL and AVERAG in the subroutine are passed back to the arguments X, Y, Z, TOTAL and AVERAG in the main program respectively. The print statement in the main program prints the value of TOTAL and AVERAG, and the execution ends.

If the input to this program is

```
20, 60, 40
```

then the output is:

```
TOTAL IS 120
AVERAGE IS 40.0000000
```

Example 6: *Integer and Real Parts of a Number. The integer and decimal parts of a real number are to be separated. For example, if the number is 3.14, the integer part is 3 and the decimal part is 0.14. Write a subroutine SEPNUM to separate the real and integer parts.*

Solution:

```
C   SUBROUTINE SUBPROGRAM
      SUBROUTINE SEPNUM(NUMBER, IPART, RPART)
      REAL NUMBER, RPART
      INTEGER IPART
      IPART = INT(NUMBER)
      RPART = NUMBER - IPART
      RETURN
      END
C MAIN PROGRAM
      REAL NUMBER, PART2
      INTEGER PART1
      READ*, NUMBER
      CALL SEPNUM(NUMBER, PART1, PART2)
      PRINT*, ' INTEGER PART OF ', NUMBER, ' IS ', PART1
      PRINT*, ' DECIMAL PART OF ', NUMBER, ' IS ', PART2
      END
```

The subroutine has three dummy arguments: argument NUMBER represents the real number to be separated, argument IPART is the integer part of NUMBER and argument RPART represents the real part of the number.

If the input to this program is

```
57.231
```

then the output is:

```
INTEGER PART OF 57.2310000 IS 57
DECIMAL PART OF 57.2310000 IS 0.2310000
```

If the subroutine SEPNUM is invoked with the statement

```
 CALL SEPNUM(3.14, PART1, PART2)
```

then the value of PART1 is 3 and value of PART2 is 0.14.

4.6 Common Errors in Subprograms

There are several common errors that occur in the use of subprograms. We illustrate such errors through an example. The following program computes the new salary, given the current salary and the number of years of service. If the number of years is more than five, the salary is to be incremented by 8%, otherwise, the increment is 4%. The program uses a function INCSAL to compute the new salary. There are several errors in the program.

When the program is executed, the following error messages appear:

- *Error #1: INCSAL is an unreferenced symbol.* A function should return a single result stored in the function name. But in function INCSAL, the function name INCSAL is not assigned any value.
- *Error #2: Function INCSAL referenced as an integer but defined to be real.* The type of the function name in the main program is, by default, integer but its type in the function definition is real.

```
C  FUNCTION SUBPROGRAM
      REAL FUNCTION INCSAL(SALARY, YEARS)
      REAL SALARY, NSAL
      INTEGER YEARS
      IF (YEARS .GT. 5) THEN
          NSAL = SALARY * 8 / 100 + SALARY
      ELSE
          NSAL = SALARY * 4 / 100 + SALARY
      ENDIF
      END
C  MAIN PROGRAM
      REAL SALARY, YEARS
      READ*, SALARY, YEARS
      PRINT*, INCSAL(SALARY, YEARS)
      END
```

- *Error #3: Argument number 2 in call to INCSAL - real argument was passed but integer argument expected.* The type of argument number 2 in the calling statement does not match with its type in function subprogram. Mismatch of arguments is a common error in calls to both subroutines and functions.
- *Error #4: **RETURN** statement is missing.* The **RETURN** statement is missing in function INCSAL. This error may not be reported by many compilers.

4.7 Exercises

1. (a) Which of the following statement(s) is (are) FALSE?
   1. A function may contain more than one **RETURN** statement.
   2. A subroutine may return one value, many values, or no value.

3. A subroutine cannot call itself in FORTRAN.
4. The statement function is a non-executable statement.
5. A function may return more than one value.
6. A program may contain more than one subprogram.
7. A subroutine cannot call another subroutine.
8. The order and type of arguments in a subroutine call and the corresponding subroutine statement must be the same.
9. Use of subroutines increases the complexity of programming.
10. A function transfers results back to the calling program in the argument lists only.

2. What is printed by the following programs ?

```
1.      INTEGER A, B, X, Y, Z, F
        A = 2
        B = 3
        X = F(4, A)
        Y = B * 3
        Z = F(Y, X)
        PRINT*, X, Y, B, Z
        END
        INTEGER FUNCTION F(X,Y)
        INTEGER X, Y, Z
        Z = 2*Y
        F = X+Z
        RETURN
        END
```

```
2.      INTEGER OP
        REAL X, Y, CALC
        READ*, X, OP, Y
        PRINT*, CALC(X, OP, Y)
        READ*, X, OP, Y
        PRINT*, CALC(X, OP, Y)
        END
        REAL FUNCTION CALC(ARG1,OP,ARG2)
        INTEGER OP
        REAL ARG1, ARG2
        IF (OP .EQ. 1) THEN
            CALC = ARG1 + ARG2
        ELSEIF (OP .EQ. 2) THEN
            CALC = ARG1 - ARG2
        ELSE
            CALC = 0
        ENDIF
        RETURN
        END
```

Assume the input is

```
1.0,5,7.0
5.0,2,4.0
```

```
3.    LOGICAL  DIV
      INTEGER   N, J
      READ*,  N, J
      IF (DIV(N, J))THEN
          PRINT*,'YES'
      ELSE
          PRINT*,'NO'
      ENDIF
      END
      LOGICAL  FUNCTION  DIV(N, J)
      INTEGER  N, J
      DIV = N - N / J * J .EQ. 0
      RETURN
      END
```

Assume the input is

```
18  4
```

```
4.    INTEGER K , EVL
      K = 1
      PRINT*, EVL (K), K
      END
      INTEGER FUNCTION EVL (M)
      INTEGER M, K
      K = 2
      EVL = M * K
      RETURN
      END
```

```
5.    INTEGER A, B
      REAL FUN
      READ*, A, B
      A = FUN(A, B)
      B = FUN(B, A)
      PRINT*, FUN(A, B)
      END
      REAL FUNCTION FUN(X, Y)
      INTEGER X, Y
      FUN = X ** 2 + 2 * Y
      RETURN
      END
```

Assume the input is

```
1, 2
```

```
6.    INTEGER A, B, C, G
      G(A,B,C) = A * B-4 * C
      READ*, A, B, C
      PRINT*, G(A + B, B + C, C + A)
      END
```

Assume the input is

```
4 5 3
```

```
7.    LOGICAL F
      INTEGER X, Y, Z
      F(X, Y, Z) = X .GT. Y .AND. X .GT. Z
      READ*, X, Y, Z
      IF (F(X, Y, Z)) PRINT*, X
      IF (F(Y, X, Z)) PRINT*, Y
      IF (F(Z, X, Y)) PRINT*, Z
      END
```

Assume the input is

```
10 30 5
```

```
8.    INTEGER A,B,P,Q, G
      G(A,B) = A*A + B
      READ*, P, Q
      A = 1
      B = 2
      PRINT*, G(P,Q), G(Q,P), G(P+2, Q+2)*G(B,A)
      END
```

Assume the input is

```
 2  3
```

```
9.    LOGICAL FUNC
      INTEGER K, L
      FUNC(K,L) = K .GE. L
      READ*, K, L
      IF (FUNC(K,L)) THEN
          PRINT*, K
      ELSE
          PRINT*,L
      ENDIF
      END
```

Assume the input is

```
80  90
```

```
10.   INTEGER K, L
      K = -9
      L = 10
      PRINT*, MOD(ABS(K),L)
      END
```

```
11.   REAL A, B, DIST, X, Y
      DIST(X,Y) = SQRT(X ** 2 + Y ** 2)
      READ*, A, B
      PRINT*, DIST(A - 3.0, DIST(A, B) - 6.0)
      END
```

```
12.   INTEGER FUNCTION FUN(J, K, M)
      REAL SUM
      SUM = J + K + M
      FUN = SUM /3.0
      RETURN
      END
      INTEGER FUN,FUS, J, K
      FUS(J, K) = J * K / 2
      PRINT*, FUS(FUN(2, 3, 4), FUN(5, 6, 7))
      PRINT*, FUN(FUS(2, 3), FUS(4, 5), FUS(6, 7))
      END
```

Assume the input is

```
    6.0     8.0
```

```
13.   REAL F, G, A, B, X, Y
      F(A , B) = A + B
      G(X) = X ** 2
      READ*, Y
      PRINT*, G(Y), G(F(Y, Y + 2))
      END
```

Assume the input is

```
3.0
```

```
14.   LOGICAL  COMP
      REAL  X, Y, Z, A, B, C
      COMP(A, B, C) = A. GE. B .AND. A .GE. C
      READ*, X, Y, Z
      IF (COMP(X, Y, Z)) PRINT*, X
      IF (COMP(Y, X, Z)) PRINT*, Y
      IF (COMP(Z, X, Y)) PRINT*, Z
      END
```

Assume the input is

```
35.0   90.0   65.0
```

```
15.   INTEGER A,B,C
      A = 1
      B = 2
      C = 3
      PRINT*, A, B, C
      CALL CHANGE(A,B)
      PRINT*, A, B, C
      END
      SUBROUTINE CHANGE(A,B)
      INTEGER A,B,C
      C = B
      B = A + B
      A = C
      RETURN
      END
```

```
16.   INTEGER TOT
      REAL A, B
      A = 5.5
      B = 4.5
      CALL ADD(A,B,TOT)
      PRINT*, TOT
      END
      SUBROUTINE ADD(X,Y,SUM)
      INTEGER SUM
      REAL X, Y
      IF (X.LT.Y) THEN
      SUM = X + Y
      ELSE
      SUM = X - Y
      ENDIF
      RETURN
      END
```

```
17.   INTEGER JJ
      JJ = 1
      CALL TRY1(JJ,3)
      CALL TRY1(JJ,4)
      CALL TRY1(JJ,5)
      PRINT*, JJ
      END
      SUBROUTINE TRY1(X,Y)
      INTEGER X,Y,TRY2, N
      TRY2(N) = N-3
      X = TRY2(Y)+2*X
      RETURN
      END
```

```
18.   INTEGER X, Y, H
      H = 2
      CALL K(X,Y)
      PRINT*, H, Y, X
      END
      SUBROUTINE K(H,Y)
      INTEGER H,Y
      REAL X
      READ*, H, Y
      H = H / (Y+H)
      Y = H+3
      X = Y+2/3
      PRINT*, H, Y, X
      RETURN
      END
```

Assume the input is

```
    5 3 2
```

```
19.   REAL X,Y
      X = 3.0
      Y = 1.0
      CALL F(X,Y)
      PRINT*, X, Y
      END
      SUBROUTINE F(A,B)
      REAL A, B
      CALL G(B,A)
      B = A + B
      A = A - B
      RETURN
      END
      SUBROUTINE G(C,D)
      REAL C, D
      C = C + D
      D = C - D
      RETURN
      END
```

```
20.   INTEGER JJ
      JJ = 1
      CALL TEST1
      PRINT*, JJ
      END
      SUBROUTINE TEST1
      INTEGER JJ
      JJ = 2
      CALL TEST2
      RETURN
      END
      SUBROUTINE TEST2
      INTEGER JJ
      JJ = 3
      RETURN
      END
```

```
21.   REAL A, C
      A = 5
      CALL SUBPRO(A,C)
      PRINT*, A, C
      END
      SUBROUTINE SUBPRO(A,B)
      REAL A, B, C, X
      C(X) = X*2-2
      B = C(A)
      RETURN
      END
```

```
22.   SUBROUTINE CHANGE (W,X,Y,Z)
      INTEGER W,X,Y,Z
      W = X
      X = Y
      Y = Z
      Z = W
      RETURN
      END
      INTEGER A,B
      READ*, A, B
      CALL CHANGE(A * 2, B * 3, A, B)
      PRINT*, A * 2, B * 3
      END
```

Assume the input is

```
3    4
```

```
23.   INTEGER X, Y
      X = 3
      Y = X*3
      PRINT*, X, Y
      CALL CHANGE(X,Y)
      PRINT*, X, Y
      END
      SUBROUTINE CHANGE(X,Y)
      INTEGER X, Y
      X = X + 1
      Y = X - 1
      PRINT*, X, Y
      RETURN
      END
```

```
24.   LOGICAL FLAG
      REAL X, Y
      FLAG = .TRUE.
      READ*, X, Y
      CALL LOGIC (X, Y, FLAG)
      PRINT*, X, Y, FLAG
      END
      SUBROUTINE LOGIC (FLAG, X, Y)
      LOGICAL Y
      REAL X, Y
      IF (.NOT. Y) THEN
          FLAG = X**2+FLAG**2
          Y = .NOT. Y
      ELSE
          FLAG = (FLAG + X)
      ENDIF
      RETURN
      END
```

Assume the input is

```
4   5
```

```
25.    REAL A, B, C
       READ*, A,B
       CALL FIRST(A,B,C)
       PRINT*, A,B,C
       END
       SUBROUTINE FIRST (X,Y,Z)
       REAL X, Y, Z
       X = X + Y
       Y = Y - X
       CALL SECOND(X,Y,Z)
       RETURN
       END
       SUBROUTINE SECOND(N,M,L)
       REAL N,M,L
       L = THIRD(N,M)
       RETURN
       END
       REAL FUNCTION THIRD(J,K)
       REAL J,K
       THIRD = J - K
       RETURN
       END
```

Assume the input is

```
1   1
```

```
26.    INTEGER A, B
       LOGICAL FLAG
       READ*, A, B
       FLAG =  A .GT. B
       CALL SUB(A, B)
       PRINT*, A, B, FLAG
       END
       SUBROUTINE  SUB(A, B)
       INTEGER  A, B, T
       LOGICAL FLAG
       T = A
       A = B
       B = T
       FLAG = A .GT. B
       RETURN
       END
```

Assume the input is

```
6   3
```

```
27.    SUBROUTINE COMP (M , N)
       INTEGER M, N
       M = M + N
       N = M + N
       RETURN
       END
       INTEGER M, N
       READ*, M, N
       CALL COMP (M , N)
       PRINT*, M, N
       END
```

Assume the input is

```
1   2
```

```
28.   SUBROUTINE   MIDTERM (A, B)
      INTEGER  A, B, C
      IF (A . LT. B) THEN
          C = A
          A = B
          B = C
      ENDIF
      RETURN
      END
      INTEGER  A, B, C
      READ*, A, B, C
      PRINT*, A, B, C
      CALL  MIDTERM (B, A)
      PRINT*, A, B, C
      END
```

Assume the input is

```
17  23  31
```

```
29.   INTEGER B, C
      REAL A
      READ*, A,  C
      CALL BEST (A, REAL(C), B)
      PRINT*, A, B, C
      CALL BEST (A, B + 2.0 , C)
      PRINT*, A, B, C
      END
      SUBROUTINE BEST (ONE, TWO, THREE)
      REAL ONE, TWO
      INTEGER THREE
      THREE = ONE + TWO
      RETURN
      END
```

Assume the input is

```
9.5, 4
```

```
30.   REAL X, Y, A, B
      F(A, B) = A / B * 2
      CALL MYSUB(F(4.0, 1.0), X, Y)
      PRINT*, X, Y, F(X, X)
      END
      SUBROUTINE MYSUB (X, Y, Z)
      REAL X, Y, Z
      IF (X .LT. 0.0) THEN
          Z = X
      ELSEIF (X .EQ. 0.0) THEN
          Z = X + 2.0
      ELSE
          Z = X / 2.0
      ENDIF
      Y = Z * X
      RETURN
      END
```

```
31.    INTEGER NUM1, NUM2
       READ*, NUM1, NUM2
       CALL EXCHNG (NUM1, NUM2)
       PRINT*, NUM1, NUM2
       END
       SUBROUTINE EXCHNG (NUM1, NUM2)
       INTEGER NUM1, NUM2, TEMP
       LOGICAL COND
       IF (.NOT. COND(NUM1, NUM2)) THEN
           TEMP = NUM1
           NUM1 = NUM2
           NUM2 = TEMP
       ENDIF
       RETURN
       END
       LOGICAL FUNCTION COND(X, Y)
       INTEGER X, Y
       COND =X .GE. 0 .AND. Y .GT. X
       RETURN
       END
```

Assume the input is

```
3, -2
```

3. Which of the following functions may be used to find the maximum of two integer numbers K and M?

```
A.     INTEGER FUNCTION MAXA(K,M)
       INTEGER K, M
       MAXA = K
       IF (K.GT.M) MAXA = K
       RETURN
       END
```

```
B.     INTEGER FUNCTION MAXC(K,M)
       INTEGER K, M
       IF (M.GE.K) THEN
           MAXC = M
       ELSE
           MAXC = K
       ENDIF
       RETURN
       END
```

```
C.     INTEGER FUNCTION MAXB(K,M)
       INTEGER K, M
       MAXB = K
       IF (M.GT.K) MAXB = M
       RETURN
       END
```

4. Write a logical function subprogram FACTOR that takes two arguments and checks if the first argument is a factor of the second argument. Write a main program to test the function.
5. Write a function subprogram to reverse a three digit number. For example, if the number is 243, the function returns 342. Write a main program to test the function.
6. Write a function subprogram called AREA to compute the area of a circle. The argument to the function is the diameter of the circle. Write a main program to test the function.

7. Write a logical function subprogram that checks whether all its three arguments are non-zero. Write a main program to test the function.
8. Write the functions in problems 4, 5, 6, and 7 as statement functions.
9. Consider the following statement function IXX (J,K) = J-J/K*K. Which one of the following intrinsic (built-in) functions is the same as the function IXX ?
   i) MOD
   ii) MAX
   iii) MIN
   iv) SQRT
10. Rewrite the following function as a STATEMENT **FUNCTION**.

```
A.      REAL FUNCTION AREA(CIRCUM)
        REAL CIRCUM, RADIUS, PI
        PI = 3.14159
        RADIUS = CIRCUM/(2.0*PI)
        AREA = RADIUS **2*PI
        RETURN
        END
```

```
B.      REAL FUNCTION  X (A, B, C, D)
        Y = A ** 2 - B ** 2
        Z = C ** 3 + 1 / D ** 2
        X = Y / Z
        RETURN
        END
```

```
C.      REAL FUNCTION  AREA (R)
        AREA = 2 * 3.14 * R ** 2
        RETURN
        END
```

11. Write a function subprogram COST that computes the cost of postage according to the following: SR 0.50 for weight of less than an ounce, SR 0.10 for each additional ounce, plus a SR 50 extra charge if the customer wants fast delivery. The arguments to the function are the weight of the package and a logical variable FAST indicating fast delivery. Write a main program to test the function.
12. Write an function subprogram that takes the three sides of a triangle and returns the type of the triangle. For a right triangle, then the function returns an integer value 1; for an isosceles triangle, the value returned is 2; for an equilateral triangle, the function returns a value 3; otherwise, a value 0 is returned.
13. Which of the following functions return the maximum of the integers K, L and M?

```
I.      INTEGER FUNCTION F1(K,L,M)
        INTEGER K, L, M
        F = K
        IF (F .LT. L) F = L
        IF (F .LT. M) F = M
        F1 = F
        RETURN
        END
```

```
II.    INTEGER FUNCTION F2(K,L,M)
       INTEGER K, L, M
       IF (K .GE. L .AND. K .GE. M) THEN
           F2 = K
       ELSEIF (L .GE. M) THEN
           F2 = L
       ELSE
           F2 = M
       ENDIF
       RETURN
       END
```

```
III.   INTEGER FUNCTION F3(K,L,M)
       LOGICAL F4
       INTEGER K, L, M
       F4(K,L,M) = K .GE. L .AND. K .GE. M
       IF (F4(K,L,M)) F3 = K
       IF (F4(L,K,M)) F3 = L
       IF (F4(M,L,K)) F3 = M
       RETURN
       END
```

14. Given the following program which has some errors.

```
       INTEGER FUNCTION TEST (A, B)
       X = (A + B) ** 2
       Y = B * 2
       RETURN
       END
       REAL TEST
       PRINT*, TEST(1, 2, 3)
       END
```

Which of the following statements is correct?

I. Function name TEST is of type integer in function description but is a real in the calling program.

II. Function name TEST is not assigned a value in the function description.

III. Argument types do not match.

IV. The number of actual arguments is more than the number of dummy arguments.

15. Rewrite the following subroutine as a function subprogram.

```
       SUBROUTINE  DIVIDE (M, N, FACTOR)
       LOGICAL  FACTOR
       INTEGER M, N
       IF (N / M * M .EQ. N) THEN
           FACTOR = .TRUE.
       ELSE
           FACTOR = .FALSE.
       ENDIF
       RETURN
       END
```

16. Rewrite the following function subprogram as a subroutine. (Hint: The statement function is part of the function subprogram).

```
      REAL FUNCTION  SO (A, B, C)
      REAL A, B, C, FUN
      FUN (A, B, C) = A / B + C
      SO = FUN (A, B, C) / FUN (C, B, A)
      RETURN
      END
```

17. Write a subroutine that takes three arguments A, B, C and returns the arguments in increasing order. Write a main program to test the subroutine.
18. Write a subroutine that takes a numeric grade of a student and prints the letter grade based on the following policy:

numeric grade	**letter grade**
above 90	A
above 80	B
above 70	C
above 60	D
below 61	F

19. Write a subroutine that computes and returns the diameter, area, and the circumference of a circle given its radius.
20. Write the functions in problems 4, 5, 6, and 7 as subroutines.
21. Write a subroutine subprogram that takes the three sides of a triangle and prints one of the following types of the triangle: right triangle, isosceles triangle, or equilateral triangle.

4.8 Solutions to Exercises

Ans 1.

Statements 5, 7, 9, and 10 are FALSE.

Ans 2.

```
   8   9   3   25
0.0
    1.0
    NO
    2    1
53.0000000
44
30
7   11    21    5
90
9
5.0000000
```

```
9
      11
9.0000000 64.0000000
90
1   2   3
     2   3   3
1
12
0   3   3.0
     2   3   0
-4.0  5.0
1
5.0  8.0
8   36
3    9
     4    3
     4        3
9.0   5.0   T
2.0  -1.0   3.0
3   6 T
3    5
17   23      31
     17      23      31
 9.5000000      13      4
     9.5000000      13      24
32.0000000     4.0000000     2.0000000
-2      3
```

Ans 3.

b and c

Ans 4.

```
      LOGICAL FUNCTION FACTOR(AR1, AR2)
      INTEGER AR1, AR2
      IF (AR2 / AR1 * AR1 .EQ. AR2)THEN
          FACTOR = .TRUE.
      ELSE
          FACTOR = .FALSE.
      ENDIF
      RETURN
      END
C     MAIN  PROGRAM
      LOGICAL FACTOR
      INTEGER AR1, AR2
      READ*, AR1, AR2
      PRINT*, FACTOR(AR1, AR2)
      END
```

Ans 5.

```
      INTEGER N, REV
      READ*, N
      IF (N .GE. 100 .AND. N .LT. 1000)THEN
          PRINT*, REV (N)
      ELSE
          PRINT*, 'OUT OF RANGE'
      ENDIF
      END

      INTEGER FUNCTION REV(N)
      INTEGER N, K, J, M
      K = N / 100
      N = N - K * 100
      J = N / 10
      M = N - J * 100
      REV = M * 100 + J * 10 + K
      RETURN
      END
```

Ans 6.

```
      REAL FUNCTION AREA (D)
      REAL D, R
      R = D / 2
      AREA = R ** 2 * 3.14
      RETURN
      END
      REAL D
      READ*, D
      PRINT*, AREA(D)
      END
```

Ans 7.

```
      LOGICAL FUNCTION TEST(A, B, C)
      REAL A, B, C
      TEST = A .NE.0 .AND. B .NE. 0 .AND. C .NE. 0
      RETURN
      END
C     MAIN  PROGRAM
      LOGICAL TEST
      REAL A, B, C
      READ*, A, B, C
      IF (TEST(A, B, C)) THEN
          PRINT*, 'ALL NUMBERS ARE NON-ZERO'
      ELSE
          PRINT*, 'NOT ALL NUMBERS ARE NON-ZERO'
      ENDIF
      END
```

Ans 8.

```
      INTEGER AR1, AR2, REV
      LOGICAL FACTOR
      REAL AREA
      FACTOR(AR1, AR2) = AR2 / AR1 * AR1 .EQ.AR2
      REV(N) = (N - N / 10 * 10) * 100 +
     *(N - N / 100 * 100) / 10 * 10 + N / 100
      AREA (D) = (D / 2) ** 2 * 3.14
      TEST (A, B, C) = A.NE.0 .AND. B.NE.0 .AND. C.NE.0
```

Ans 9.

i

Ans 10.

```
A.    REAL AREA
      AREA(CIRCUM) = 3.14159 * (CIRCUM/(2.0 * 3.14159)) ** 2
```

```
B.    REAL X
      X(A, B, C, D) = (A ** 2 - B ** 2) / (C ** 3 + 1 / D ** 2)
```

```
C.    REAL AREA
      AREA(R) =  2 * 3.14 * R ** 2
```

Ans 11.

```
      REAL FUNCTION COST (WEIGHT, FAST)
      LOGICAL FAST
      IF (WEIGHT .LT. 1) THEN
          COST = 0.5
      ELSE
          COST = 0.5 + (WEIGHT - 1) * 0.10
      ENDIF
      IF (FAST) COST = COST + 50
      RETURN
      END
      LOGICAL FAST
      READ* , WEIGHT, FAST
      PRINT*, COST(WEIGHT, FAST)
      END
```

Ans 12.

```
      INTEGER FUNCTION TTYPE(A, B, C)
      REAL A, B, C
C  ASSUMING C IS THE LARGEST SIDE
      IF(SQRT(C) .EQ. SQRT(A + B)) THEN
          TTYPE = 1
      ELSEIF(A .EQ. B .AND. A .EQ. C) THEN
          TTYPE = 3
      ELSEIF(A .EQ. B .OR. B .EQ. C .OR. C .EQ. A) THEN
          TTYPE = 2
      ELSE
          TTYPE = 0
      ENDIF
      RETURN
      END
```

Ans 13.

I, II and III.

Ans 14.

I, II, III and IV.

Ans 15.

```
      LOGICAL FUNCTION FACTOR (M, N)
      INTEGER M, N
      IF (N / M * M .EQ. N) THEN
          FACTOR = .TRUE.
      ELSE
          FACTOR = .FALSE.
      ENDIF
      RETURN
      END
```

Ans 16.

```
      SUBROUTINE ANS(A,B,C,SO)
      REAL A, B, C, SO, FUN
      FUN (A, B, C) = A / B + C
      SO = FUN (A, B, C) / FUN (C, B, A)
      RETURN
      END
```

Ans 17.

```
      SUBROUTINE ORDER (A, B, C)
      INTEGER A, B, C, T
      IF (A .GT. B) THEN
          T = A
          A = B
          B = T
      ENDIF
      IF (A .GT. C) THEN
          T = A
          A = C
          C = T
      ENDIF
      IF (B .GT. C) THEN
          T = B
          B = C
          C = T
      ENDIF
      RETURN
      END
      INTEGER A, B, C
      READ* , A, B, C
      CALL ORDER (A, B, C)
      PRINT*, A, B, C
      END
```

Ans 18.

```
      SUBROUTINE LGRADE(MARK)
      REAL MARK
      IF(MARK .GE. 0  .AND. MARK .LE. 100) THEN
          IF(MARK .GT. 90) THEN
              PRINT*, 'A'
          ELSEIF(MARK .GT. 80) THEN
              PRINT*, 'B'
          ELSEIF(MARK .GT. 70) THEN
              PRINT*, 'C'
          ELSEIF(MARK .GT. 60) THEN
              PRINT*, 'D'
          ELSE
              PRINT*, 'F'
          ENDIF
      ELSE
          PRINT*, 'MARK OUT OF RANGE'
      ENDIF
      RETURN
      END
```

Ans 19.

```
      SUBROUTINE CIRCLE(R, D, A, C)
      REAL R, D, A, C
      D = R / 2
      A = 22.0 / 7.0 * R ** 2
      C = 2 *  22.0 / 7.0 * R
      RETURN
      END
```

Ans 20.

of problem 4

```
      SUBROUTINE   FACTOR (AR1, AR2, FLAG)
      INTEGER AR1, AR2
      LOGICAL FLAG
      FLAG =  AR2 / AR1 * AR1 .EQ. AR2
      RETURN
      END
```

of problem 5.

```
      SUBROUTINE   FIND (N, REV)
      INTEGER N, REV
      M = N / 100
      N = N - M * 100
      J = N / 10
      K = N - J * 10
      REV = K * 100 + J * 10 + M
      RETURN
      END
```

of problem 6.

```
      SUBROUTINE  CIRCLE(D, AREA)
      R = D / 2
      AREA = 22.0 / 7.0 * R ** 2
      RETURN
      END
```

of problem 7.

```
      SUBROUTINE   CHECK (A, B, C, TEST)
      LOGICAL TEST
      TEST = A .NE. 0 .AND. B .NE. 0 .AND. C .NE. 0
      RETURN
      END
```

Ans 21.

```
      SUBROUTINE   TTYPE (A, B, C)
      REAL A, B, C
C  ASSUMING C IS THE LARGEST SIDE
      IF(SQRT(C) .EQ. SQRT(A + B)) THEN
          PRINT* , 'RIGHT TRIANGLE'
      ELSEIF(A .EQ. B .AND. A .EQ. C) THEN
          PRINT* , 'EQUILATERAL TRIANGLE'
      ELSEIF(A.EQ.B .OR. B.EQ.C .OR. C.EQ.A)THEN
          PRINT* , 'ISOSCELES TRIANGLE'
      ELSE
          PRINT* , 'NONE OF THE OTHER TYPES'
      ENDIF
      RETURN
      END
```

5 REPETITION

While writing a program, it may be necessary to execute a statement or a group of statements repeatedly. Repetition is supported in FORTRAN through two repetition constructs, namely, the **DO** and the **WHILE** constructs. A repetition construct is also known as a *loop*.

In a repetition construct, a group of statements, which are executed repeatedly, is called the *loop body*. A single execution of the loop is called an *iteration*. Every repetition construct must *terminate* after a *finite* number of iterations. The termination of the loop is decided through what is known as the *termination condition*. A decision is made whether to execute the loop for another iteration through the termination condition. In the case of a **DO** loop, the number of iterations is known before the loop is executed; the termination condition checks whether this number of iterations have been executed. In the case of a **WHILE** loop, such a decision is made in every iteration.

Repetition constructs are very useful and extensively used in solving a significant number of programming problems. Let us consider the following example as an illustration of such constructs.

Example : *Average Computation: Assume that we were asked to write a FORTRAN program that reads the grades of 8 students in an exam. The program is to compute and print the average of the grades. Without repetition, the following program may be considered as a solution.*

Solution:

```
      REAL X1, X2, X3, X4, X5, X6, X7, X8
      REAL SUM, AVG
      READ*, X1
      READ*, X2
      READ*, X3
      READ*, X4
      READ*, X5
      READ*, X6
      READ*, X7
      READ*, X8
      SUM = X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8
      AVG = SUM / 8.0
      PRINT*, AVG
      END
```

The variable SUM is a real variable in which we store the summation of the grades. The statements are considerably long for just 8 students. Imagine the size of such statements

when the number of students is 100. It is highly inefficient to use 100 different variable names.

From the example above, let us try to extract the statements where repetition occurs. The reading and assignment statements are clearly such statements. We can do the reading and addition in these statements, individually, for each grade. The following repetitive segment can be used instead of the long read and assignment statements :

```
      SUM = 0
      REPEAT THE FOLLOWING STATEMENTS 8 TIMES
          READ*, X
          SUM = SUM + X
```

In each iteration, one grade is read and then added to the previous grades. In the first iteration, however, there are no previous grades. Therefore, SUM is initialized to zero, meaning that the summation of the grades is zero, before any grade is read.

This repetitive solution is more efficient since it can be used for any number of students. By reading the number of students N, the repetition construct above, can be changed, to find the sum of the grades of N students, as follows :

```
      SUM = 0
      READ*, N
      REPEAT THE FOLLOWING STATEMENTS N TIMES
          READ*, X
          SUM = SUM + X
```

The repetition construct above is not written in the FORTRAN language. To implement this construct in FORTRAN, we can use two types of loops: the **DO** Loop and the **WHILE** loop.

5.1 The DO Loop

One very basic feature of the **DO** loop repetitive construct is that the number of iterations (the number of times the loop is executed) is known (computed) before the loop execution begins. The general form of the **DO** loop is:

```
      DO N index = initial, limit, increment
          BLOCK OF FORTRAN STATEMENTS
N     CONTINUE
```

The **CONTINUE** statement indicates the end of the **DO** loop.

The number of times (iterations) the loop is executed is computed as follows :

$$\text{Number of times a Do loop is Executed} = \left\lceil \frac{\text{limit} - \text{initial}}{\text{increment}} \right\rceil + 1$$

The detailed logic of the **DO** loop is as follows:

- If the *increment* is positive, the value of the *initial* must be less than or equal to the value of the *limit*. If the *increment* is negative, the value of the *initial* must be greater than or equal to the value of the *limit*. Otherwise, the loop will not be executed. If the values of the *initial* and the *limit* are equal, the loop executes only once.
- In the first iteration, the *index* of the loop has the value of *initial* .
- Once the **CONTINUE** statement is reached, the *index* is increased or decreased by the *increment* and the execution of the next iteration starts. Before each

iteration, the *index* is checked to see if it has reached the *limit*. If the *index* reaches the *limit*, the loop iterations stop. Otherwise, the next iteration begins.

Consider the following example as an illustration of the **DO** loop :

```
      DO 15 K = 1, 5, 2
         PRINT*, K
15    CONTINUE
```

The loop above is executed $\left\lceil \frac{5-1}{2} \right\rceil + 1 = 3$ times. Thus, the values index K takes during the execution of the loop are 1, 3, and 5. Note that the value of K increments by 2 in each iteration. In the beginning, we make sure that the initial is less than the limit since the value of the increment is positive. The execution of the loop begins and the value of K, which is **1**, is printed. The **CONTINUE** statement returns the control to the **DO** statement and the execution of the loop takes place for the second time with the value of K as **3**. This continues for the third time with K as **5**. Once this iteration is over, the control goes back and the *index* K gets incremented again to **7**, which is more than the *limit*. The execution of the loop stops and control transfers to the statement following the **CONTINUE** statement. Note that the value of K outside the loop is 7.

The following **rules** apply to **DO** loops:

- The *index* of a **DO** loop must be a variable of either **INTEGER** or **REAL** types.
- The parameters of the loop, namely, *initial*, *limit*, and *increment* can be expressions of either **INTEGER** or **REAL** types. Although it depends on the nature of the problem being solved, it is recommended that the type of the parameters match the type of the *index*.
- The value of the **DO** loop *index* cannot be modified inside the loop. Any attempt to modify the *index* within the loop will cause an error.
- The *increment* must not be **zero**, otherwise an error occurs.
- If the *index* is an integer variable then the values of the parameters of the **DO** loop will be truncated to integer values before execution starts.
- The value of the index after the execution of the loop is either the value that has been incremented and found to exceed the limit (for a positive increment) or the value that has been decremented and found to be less than the limit (for a negative increment).
- It is not allowed to branch into a **DO** loop. Entering the **DO** loop has to be through its **DO** statement. It is possible to branch out of a **DO** loop before all the iterations are completed. This type of branching must not be used unless necessary.
- It is possible to have a **DO** loop without the **CONTINUE** statement. The *statement number*, which is given to the **CONTINUE** statement, can be given to the last FORTRAN statement in the loop, except in the case when the last statement is either an **IF**, **GOTO**, **RETURN**, **STOP** or another **DO** statement.
- In the **DO** loop construct, in the absence of the increment, the default increment is +1 or +1.0 depending on the type of the *index*.

- In the case when the *increment* is positive but the *initial* is greater than the *limit*, a **zero-trip DO** loop occurs. That is, the loop executes zero times. The same happens when the *increment* is negative and the *initial* is less than the *limit*. Note that a zero-trip **DO** loop is not an error.
- The same continue statement number can be used in both a subprogram and the main program invoking the subprogram. This is allowed because subprograms are considered separate programs.
- The parameters of the loop are evaluated before the loop execution begins. Once evaluated, changing their values will not affect the executing of the loop. For an example, consider the following segment. Changing **DO** loop parameters inside the loop should be avoided while writing application programs.

```
      REAL X, Y
      Y = 4.0
      DO 43 X = 0.0, Y, 1.5
          PRINT*, X
          Y = Y + 1.0
          PRINT*, Y
43    CONTINUE
```

In the above loop, the value of Y which corresponds to the limit in the **DO** loop, starts with **4**. Therefore, and according to the rule we defined earlier, this loop is executed $\left\lceil \frac{4.0-0.0}{1.5} \right\rceil + 1 = 3$ times. The values of the parameters (*initial, limit,* and *increment*) are set at the beginning of the loop and they never change for any iteration of the loop. Although the value of Y changes in each iteration within the loop, the value of the limit does not change. The following examples illustrate the ideas explained above:

5.1.1 Examples on DO loops

Example 1: *Consider the following program.*

```
      DO 124 M = 1, 100, 0.5
          PRINT*, M
124       CONTINUE
      PRINT*, M
      END
```

*In the above program, the value of the increment is **0.5**. When this value is added and assigned to the index M, which is an **integer**, the fraction part gets truncated. This means that the increment is **0** which causes an error.*

Example 2: The Factorial: *Write a FORTRAN program that reads an integer number M. The program then computes and prints the factorial of M.*

Solution:

```
      INTEGER M, TERM, FACT
      READ*, M
      IF (M.GE.0) THEN
          FACT = 1
          TERM = M
           DO 100 M = TERM, 2, -1
              IF (TERM.GT.1) THEN
                  FACT = FACT * TERM
100       CONTINUE
           PRINT*,'FACTORIAL OF ', M, ' IS ', FACT
      ELSE
          PRINT*, 'NO FACTORIAL FOR NEGATIVES'
      ENDIF
      END
```

To compute the factorial of 3, for example, we have to perform the following multiplication: 3 * 2 * 1. Notice that the terms decrease by 1 and stop when the value reaches 1. Therefore, the header of the **DO** loop forces the repetition to stop when TERM, which represents the number of terms, reaches the value 1.

5.2 Nested DO Loops

DO loops can be nested, that is you may have a **DO** loop inside another **DO** loop. However, one must start the inner loop after starting the outer loop and end the inner loop before ending the outer loop. It is allowed to have as many levels of nesting as one wishes. The constraint here is that inner loops must finish before outer ones and the indexes of the nested loops must be different. The following section presents some examples of nested **DO** loops.

5.2.1 Example on Nested DO loops

Example 1: *Nested **DO** Loops. Consider the following program.*

```
          DO 111 M = 1, 2
              DO 122 J = 1, 6 , 2
                  PRINT*, M, J
122           CONTINUE
111       CONTINUE
          END
```

The output of the above program is:

```
1   1
1   3
1   5
2   1
2   3
2   5
```

Example 2: *The above program can be rewritten using one **CONTINUE** statement as follows:.*

```
          DO 111 M = 1, 2
              DO 111 J = 1, 6 , 2
                  PRINT*, M, J
111       CONTINUE
          END
```

Notice that both do loops has the same label number and the same ***CONTINUE*** *statement.*

Example 3: *The above program can be rewritten without any* ***CONTINUE*** *statement as follows:*

```
          DO 111 M = 1, 2
             DO 111 J = 1, 6 , 2
111             PRINT*, M, J
          END
```

Notice that the label of the do loop will be attached to the last statement in the do loop.

5.3 The WHILE Loop

The *informal* representation of the WHILE loop is as follows :

```
      WHILE condition EXECUTE THE FOLLOWING
         block of statementS.
```

In this construct, the *condition* is checked before executing the *block of statements*. The *block of statements* is executed only if the *condition*, which is a logical expression, evaluates to a *true* value. At the end of each iteration, the control returns to the beginning of the loop where the *condition* is checked again. Depending on the value of the *condition*, the decision to continue for another iteration is made. This means that the number of iterations the **WHILE** loop makes depends on the *condition* of the loop and could not always be computed before the execution of the loop starts. This is the main difference between **WHILE** and **DO** repetition constructs.

Unlike other programming languages such as PASCAL and C, standard FORTRAN does not have an explicit **WHILE** statement for repetition. Instead, it is built from the **IF** and the **GOTO** statements.

In FORTRAN, the **IF-THEN** construct is used to perform the test at the beginning of the loop. Consider an **IF** statement, which has the following structure :

```
      IF (condition) THEN
         block of statements
      ENDIF
```

If the condition is TRUE, the *block of statements* is executed once. For the next iteration, since we need to go to the beginning of the **IF** statement, we require the **GOTO** statement. It has the following general form :

```
      GOTO statement number
```

A **GOTO** statement transfers control to the statement that has the given statement number. Using the **IF** and the **GOTO** statements, the general form of the **WHILE** loop is as follows :

```
n     IF (condition) THEN
         block of statements
         GOTO n
      ENDIF
```

n is a positive integer constant up to 5 digits and therefore, ranges from 1 to 99999. It is the label of the **IF** statement and must be placed in columns 1 through 5.

The execution of the loop starts if the *condition* evaluates to a *.TRUE.* value. Once the loop iterations begin, the *condition* must be ultimately changed to a .FALSE. value,

so that the loop stops after a finite number of iterations. Otherwise, the loop never stops resulting in what is known as the *infinite* loop. In the following section, we elaborate more on the **WHILE** loop.

5.3.1 Examples on WHILE Loops

Example 1: *Computation of the Average: Write a FORTRAN program that reads the grades of 100 students in a course. The program then computes and prints the average of the grades.*

Solution:

```
      REAL X, AVG, SUM
      INTEGER K
      K = 0
      SUM = 0.0
25    IF (K.LT.100) THEN
          READ*, X
          K = K + 1
          SUM = SUM + X
          GOTO   25
      ENDIF
      AVG = SUM / K
      PRINT*, AVG
      END
```

Note that the variable K starts at 0. The value of K is incremented after the reading of a grade. The **IF** condition presents the loop from reading any new grades once the 100th grade is read. Reading the 100th grade causes K to be incremented to the value of 100 as well. Therefore, when the condition is checked in the next iteration, it becomes .FALSE. and the loop stops.

In each iteration, the value of the variable GRADE is added to the variable SUM. After the loop, the average is computed by dividing the variable SUM by the variable K.

Example 2: *The Factorial: The problem is the same as the one discussed in Example 2 of Section 5.2. In this context, however, we will solve it using a* ***WHILE*** *loop.*

Solution:

```
      INTEGER M, TERM, FACT
      READ*, M
      IF (M.GE.0) THEN
          FACT = 1
          TERM = M
3     IF (TERM.GT.1) THEN
          FACT = FACT *TERM
          TERM =TERM - 1
          GOTO 3
      ENDIF
          PRINT*,'FACTORIAL OF ', M, ' IS ', FACT
      ELSE
          PRINT*, 'NO FACTORIAL FOR NEGATIVES'
      ENDIF
      END
```

Note the similarities between both solutions. The **WHILE** loop starts from **M** (the value we would like to compute the factorial of) and the condition of the loop makes sure that the loop will only stop when TERM reaches the value 1.

Example 3: *Classification of Boxers: Write a FORTRAN program that reads the weights of boxers. Each weight is given on a separate line of input. The boxer is classified according to the following criteria: if the weight is less than or equal to 65 kilograms, the boxer is light-weight; if the weight is between 65 and 85 kilograms, the boxer is middle-weight and if the weight is more than or equal to 85, the boxer is a heavy-weight. The program prints a proper message according to this classification for a number of boxers by reading their weights repeatedly from the input. This repetitive process of reading and classification stops when a weight of* ***-1.0*** *is read.*

Solution:

```
      REAL WEIGHT
      READ*, WEIGHT
11    IF (WEIGHT.NE.-1.0) THEN
          IF (WEIGHT.LT.0.OR.WEIGHT.GE.400) THEN
              PRINT*, ' WEIGHT IS OUT OF RANGE '
          ELSEIF (WEIGHT.LE.65) THEN
              PRINT*, ' LIGHT-WEIGHT '
          ELSEIF (WEIGHT.LT.85) THEN
              PRINT*, ' MIDDLE-WEIGHT '
          ELSE
              PRINT*, ' HEAVY-WEIGHT '
          ENDIF
          READ*, WEIGHT
          GOTO 11
      ENDIF
      END
```

Note that in this example, the condition that stops the iterations of the loop depends on the **READ** statement. The execution of the loop stops when a value of **-1.0** is read. This value is called the ***end marker*** or the ***sentinel***, since it marks the end of the input. A sentinel must be chosen from outside the range of the possible input values.

5.4 Nested WHILE Loops

WHILE loops may be nested, that is you can put a **WHILE** loop inside another **WHILE** loop. However, one must start the inner loop after starting the outer loop and end the inner loop before ending the outer loop for a logically correct nesting. (The following example is equivalent to the nested **DO** loop example given earlier.)

Example: Consider the following program.

```
      M = 1
22    IF( M .LE. 2) THEN
          J = 1
11        IF (J .LE. 6) THEN
              PRINT*, M, J
              J = J + 2
              GOTO 11
          ENDIF
          M = M + 1
          GOTO 22
      ENDIF
      END
```

The output of the above program is:

```
1   1
1   3
1   5
```

```
2   1
2   3
2   5
```

There are two nested **WHILE** loops in the above program. The outer loop is controlled by the variable M. The inner loop is controlled by the variable J. For each value of the variable M, the inner loop variable J takes the values 1, 3 and 5.

5.5 Examples on DO and WHILE Loops

Example 1: *Evaluation of Series: Write a FORTRAN program that evaluates the following series to the 7th term.*

$$\sum_{i=1}^{N} 3^{i}$$

(Summation of base 3 to the powers from 1 to N. Assume N has the value 7)

Solution:

```
      INTEGER SUM
      SUM = 0
      DO 11 K = 1, 7
          SUM = SUM + 3 ** K
11    CONTINUE
      PRINT*, SUM
      END
```

Example 2: *Alternating Sequences/ Series: Alternating sequences, or series, are those which have terms alternating their signs from positive to negative. In this example, we find the sum of an alternating series.*

Question: Write a FORTRAN program that evaluates the following series to the 100th term.

1 - 3 + 5 - 7 + 9 - 11 + 13 - 15 + 17 - 19 + ...

Solution:

It is obvious that the terms differ by 2 and start at the value of 1.

```
      INTEGER SUM, TERM,NTERM
      SUM = 0
      TERM = 1
      DO 10 NTERM = 1, 100
          SUM = SUM + (-1) ** (NTERM + 1) * TERM
          TERM = TERM + 2
10    CONTINUE
      PRINT*, SUM
      END
```

Notice the summation statement inside the loop. The expression (-1) ** (NTERM + 1) is positive when NTERM equals 1, that is for the first term. Then, it becomes negative for the second term since NTERM + 1 is 3 and so on.

Example 3: *Series Summation using a WHILE loop: Question: Write a FORTRAN program which calculates the sum of the following series :*

$$\frac{1}{2}+\frac{2}{3}+\frac{3}{4}+\frac{4}{5}+\mathrm{L}+\frac{99}{100}$$

Solution:

```
      REAL N, SUM
      N = 1
      SUM = 0
10    IF (N.LE.99) THEN
          SUM = SUM + N / (N + 1)
           N = N + 1
          GOTO 10
      ENDIF
      PRINT*, SUM
      END
```

In the above program, if N is not declared as **REAL**, the expression N/(N+1), in the summation inside the loop, will always compute to zero.

Example 4: *Conversion of a WHILE loop to a **DO** loop: Convert the following **WHILE** loop into a **DO** loop.*

```
      REAL X, AVG, SUM
      INTEGER K
      K = 0
      SUM = 0.0
25    IF (K.LT.100) THEN
          READ*, X
          K = K + 1
          SUM = SUM + X
          GOTO 25
      ENDIF
      AVG = SUM / K
      PRINT*, AVG
      END
```

In the **WHILE** loop, K starts with the value of 0, and within the loop it is incremented by 1 in each iteration. The *termination condition* is that the value of K must exceed 99. In the equivalent program using a **DO** loop, K starts at 0 and stops at 99 and gets incremented by 1 in each iteration.

Solution:

The equivalent program using a **DO** loop is as follows:

```
      REAL X, AVG, SUM
      INTEGER K
      SUM = 0.0
      DO 25 K = 0, 99, 1
          READ*, X
          SUM = SUM + X
 25   CONTINUE
      AVG = SUM / 100
      PRINT*, AVG
      END
```

An important point to note in this example is the way the average is computed. The statement that computes the average divides the summation of the grades SUM by 100. Note that the value of the K is 100 because the loop stops when the value of K exceeds 99. Keeping in mind that the increment is 1, the value of K after the loop terminates is 100. However, it is not recommended to use the value of the index outside the **DO** loop.

It is also important to note that any other parameters such as:

```
      DO 25 K = 200, 101, -1
```

would also have the same effect. Note that the variable K exits the loop with the value 100 in this case as well.

It is not always possible to convert a **WHILE** loop into a **DO** loop. As an example, consider the **WHILE** loop in the Classification of Boxers example. There, we cannot accomplish the conversion because the number of times the **WHILE** loop gets executed is not known. It depends on the number of data values before the *end marker*.

5.6 Implied Loops

Implied loops are only used in **READ** and **PRINT** statements. The implied loop is written in the following manner :

```
      READ*,(list of variables, index = initial, limit, increment)
      PRINT*,(list of expressions, index = initial, limit, increment)
```

As in the case of explicit **DO** loops, the index must be either an integer or real expression. The variables in the **READ** statement can be of any type including array elements. The expressions in the **PRINT** statement can be of any type as well. All the rules that apply to **DO** loop parameters also apply to implied loop parameters. Usage of implied loops is given in the following examples :

Example 1: *Printing values from 100 to 87: The following segment prints the integer values from 100 down to 87 in a single line.*

```
      PRINT*, (K, K = 100 , 87 , -1)
```

Output:

```
100   99    98    97    96    95    94    93    92    91    90    89    88    87
```

Notice that the increment is -1, which means that the value of K decreases from 100 to 87. In each iteration, the value of K is printed. The value of K is printed $\left\lceil \frac{87-100}{-1} \right\rceil + 1 = 14$ times. Since K is the index of the loop, the value printed here is the value of the index, which varies in each iteration. Consider the following explicit **DO** loop version of the implied loop :

```
      DO 60 K = 100, 87 , -1
         PRINT*, K
60    CONTINUE
```

Output:

```
100
 99
 98
...
...
...
 87
```

The two loops are equivalent except in terms of the shape of the output. In the implied loop version, the output will be printed on one line. In the explicit **DO** loop version, the output will be printed as one value on each line.

Example 2: *Printing more than one value in each iteration of an implied loop: The following segment prints a percentage sign followed by a + sign three times :*

```
      PRINT*, ('%' , '+' , M = 1 , 3)
```

This produces the following output:

```
%+%+%+
```

Notice that the parenthesis encloses both the % and the + signs, which means they both have to be printed in every iteration the loop makes.

Example 3: Nested Implied Loops: *An implied loop may be nested either in another implied loop or in an explicit **DO** loop. There is no restriction on the number of levels of nesting. The following segment shows nested implied loops.*

```
      PRINT*, ((K, K = 1 , 5 , 2), L = 1 , 2)
```

Nested implied loops work in a similar manner as the nested **DO** loops. One very important point to note here is the double parenthesis before the K in the implied version. It means that the inner loop with index variable K is enclosed within the outer one with index variable L. The L loop is executed $\left\lceil \frac{2-1}{1} \right\rceil + 1 = 2$ times. The K loop forces the value of K to be printed $\left\lceil \frac{5-1}{2} \right\rceil + 1 = 3$ iterations. However, since the K loop is nested inside the L loop, the K loop is executed 3 times in each iteration of the L loop. Thus, K is printed 6 times. Therefore, the output of the implied version is:

```
1   3   5   1   3   5
```

5.7 Repetition Constructs in Subprograms

Subprograms in FORTRAN are considered separate programs during compilation. Therefore, repetition constructs in subprograms are given the same treatment as in programs. The following is an example that shows how repetition is used in subprograms.

Example: *Count of Integers in some Range that are Divisible by a given Value: Write a function subprogram that receives three integers as input. The **first** and **second** input integers make the range of values in which the function will conduct the search. The function searches for the integers in that range that are divisible by the **third** input integer. The function returns the count of such integers to the main program. The main program reads five lines of input. Each line consists of three integers. After each read, the main program calls the function, passes the three integers to it and receives the output from it and prints that output with a proper message :*

Solution:

```
      INTEGER K, L, M, COUNT, J, N
      DO 10 J = 1 , 5
          READ*, K, L, M
          N = COUNT(K , L , M)
          PRINT*, 'COUNT OF INTEGERS BETWEEN',K,'AND', L
          PRINT*, 'THAT ARE DIVISIBLE BY', M, 'IS', N
          PRINT*
10    CONTINUE
      END
      INTEGER FUNCTION COUNT(K , L , M)
      INTEGER K, L, M, INCR, NUM, J
      INCR = 1
      NUM = 0
      IF (L .LT. K) INCR = -1
      DO 10 J = K, L, INCR
          IF (MOD(J , M) .EQ. 0) NUM = NUM + 1
10    CONTINUE
      COUNT = NUM
      RETURN
      END
```

If we use the following input:

```
2   34  2
-15 -30  5
70  32   7
0   20   4
-10 10  10
```

The typical output would be as follows:

```
COUNT OF INTEGERS BETWEEN 2 AND 34
THAT ARE DIVISIBLE BY 2 IS 12

COUNT OF INTEGERS BETWEEN -15 AND -30
THAT ARE DIVISIBLE BY 5 IS 4

COUNT OF INTEGERS BETWEEN 70 AND 32
THAT ARE DIVISIBLE BY 7 IS 6

COUNT OF INTEGERS BETWEEN 0 AND 20
THAT ARE DIVISIBLE BY 4 IS 6

COUNT OF INTEGERS BETWEEN -10 AND 10
THAT ARE DIVISIBLE BY 10 IS 3
```

Remember what we said about the subprogram being a separate entity from the main program invoking it. Accordingly, note the following in the above example:

- It is allowed to use the same statement number in the main program and subprograms of the same file. Notice the statement number **10** in both the main program and the function subprogram
- It is also allowed to use the same variable name as index of **DO** loops in the main program and the subprogram. Notice the variable **J** in the above

5.8 Exercises

1. What will be printed by the following programs?

```
1.    LOGICAL FUNCTION PRIME(K)
      INTEGER N, K
      PRIME = .TRUE.
      DO 10 N = 2, K / 2
         IF (MOD(K , N) .EQ. 0) THEN
            PRIME = .FALSE.
         ENDIF
10    CONTINUE
      RETURN
      END
      LOGICAL PRIME
      PRINT*, PRIME(5), PRIME(8)
      END
```

```
2.    INTEGER FUNCTION FACT(K)
      INTEGER K,L
      FACT = 1
      DO 10 L = 2 , K
         FACT = FACT * L
10    CONTINUE
      RETURN
      END
      INTEGER FUNCTION COMB(N , M)
      INTEGER FACT
      IF (N .GT.M) THEN
         COMB = FACT(N) / (FACT(M) * FACT(N-M))
      ELSE
         COMB = 0
      ENDIF
      RETURN
      END
      INTEGER COMB
      PRINT*, COMB(4 , 2)
      END
```

```
3.    INTEGER K, M, N
      N = 0
      DO 10 K = -5 , 5
         N = N + 2
         DO 20 M = 3 , 1
            N = N + 3
20       CONTINUE
         N = N + 1
10    CONTINUE
      PRINT*, N
      END
```

```
4.    INTEGER ITOT, N
      READ*, N
      ITOT = 1
10    IF (N .NE. 0) THEN
         ITOT = ITOT * N
         READ*, N
         GOTO 10
      ENDIF
      READ*, N
20    IF (N .NE. 0) THEN
         ITOT = ITOT * N
         READ*, N
         GOTO 20
      ENDIF
      PRINT*,ITOT
      END
```

Assume the input is

```
2
0
3
0
4
```

```
5.    INTEGER FUNCTION CALC(A,B)
      INTEGER A,B,R, K
      R = 1
      DO 10 K=1,B
         R = R*A
10    CONTINUE
      CALC = R
      RETURN
      END
      INTEGER CALC
      READ*,M,N
      PRINT*,CALC(M,N)
      END
```

Assume the input is

```
2   5
```

```
6.    INTEGER KK, J, K
         KK = 0
2     IF ( KK.LE.0) THEN
         READ*, J , K
         KK = J - K
         GOTO 2
      ENDIF
      PRINT*,KK,J,K
      END
```

Assume the input is

```
2   3
-1   2
3   3
```

```
4    -3
2    5
4    3
```

```
7.      INTEGER K, J
        K = 2
25      IF ( K.GT.0 ) THEN
            DO 15 J = K, 3, 2
                PRINT*, K, J
15          CONTINUE
            K = K - 1
            GOTO 25
        ENDIF
        END
```

```
8.      INTEGER N, C
        LOGICAL FLAG
        READ*, N
        FLAG = .TRUE.
        C = N ** 2
22      IF ( FLAG ) THEN
            C = ( C + N ) / 2
            FLAG = C.NE.N
            PRINT*, C
            GOTO 22
        ENDIF
        END
```

Assume the input is

```
4
```

```
9.      INTEGER N, K
        READ*, N
        K = SQRT(REAL(N))
33      IF ( K*K .LT. N ) THEN
            K = K + 1
            GOTO 33
        ENDIF
        PRINT*, K*K
        END
```

Assume the input is

```
 6
```

```
10.     INTEGER J, K
        DO 10 K = 1,2
            PRINT*, K
            DO 10 J = 1,3
10              PRINT*,K,J
        END
```

```
11.     INTEGER X, K, M
        M = 4
        DO 100 K = M ,M+2
            X = M + 2
            IF ( K.LT.6 ) THEN
                PRINT*,'HELLO'
            ENDIF
100     CONTINUE
        END
```

```
12.     INTEGER SUM, K, J, M
        SUM = 0
        DO 1 K = 1,5,2
            DO 2 J = 7,-2,-3
                DO 3 M = 1980,1989,2
                    SUM = SUM + 1
3               CONTINUE
2           CONTINUE
1       CONTINUE
        PRINT*,SUM
        END
```

```
13.     LOGICAL T, F
        INTEGER BACK, FUTURE, K
        BACK = 1
        FUTURE = 100
        T = .TRUE.
        F = .FALSE.
        DO 99 K = BACK,FUTURE,5
            T = ( T.AND..NOT.T ) .OR. ( F.OR..NOT.F )
            F = .NOT.T
            FUTURE = FUTURE*BACK*(-1)
99      CONTINUE
        IF (T) PRINT*, 'DONE'
        IF (F) PRINT*, 'UNDONE'
        END
```

2. Find the number of iterations of the WHILE-LOOPS in each of the following programs:

```
1.      INTEGER K, M, J
        K = 80
        M = 5
        J = M-M/K*K
10      IF ( J.NE.0 ) THEN
            PRINT*, J
            J = M-M/K*K
            M = M + 1
            GOTO 10
        ENDIF
        END
```

```
2.      REAL W
        INTEGER L
        W = 2.0
        L = 5 * W
100     IF ( L/W.EQ.((L/4.0)*W) ) THEN
            PRINT*, L
            L = L + 10
            GOTO 100
        ENDIF
        END
```

3. Which of the following program segments causes an infinite loop?

```
(I)     J = 0
25      IF ( J.LT.5 ) THEN
            J = J + 1
            GOTO 25
        ENDIF
        PRINT*, J
```

```
II.   J = 0
25    IF ( J.LT.5 ) THEN
          J = J + 1
      ENDIF
      GOTO 25
      PRINT*, J
```

```
III.  X = 2.0
5     X = X + 1
      IF ( X.GT.4 ) X = X + 1
      GOTO 5
      PRINT*, X
```

```
IV.   M = 2
      K = 1
10    IF ( K.LE. M ) THEN
20        M = M + 1
          K = K + 2
          GOTO 20
      ENDIF
      GOTO 10
```

```
V.    X = 1
4     IF ( X.GE.1 ) GOTO 5
5     IF ( X.LE.1 ) GOTO 4
```

```
VI.   J = 1
33    IF ( J.GT.5 ) THEN
          GOTO 22
      ENDIF
      PRINT*, J
      J = J + 1
      GOTO 33
22    STOP
```

4. Convert the following WHILE loops to **DO** loops :

```
I.    ID = N
10    IF ( ID.LE.891234 ) THEN
          PRINT*, ID
          ID = ID + 10
          GOTO 10
      ENDIF
```

```
II.   L = 1
      SUM =0
3     IF (L.LE.15) THEN
          J = -L
2         IF (J.LE.0) THEN
              SUM =SUM+J
              J = J + 1
              GOTO 2
          ENDIF
          L = L+3
          GOTO 3
      ENDIF
      PRINT*,SUM
```

5. What will be printed by the following program :

```
      INTEGER ISUM, K, N
      ISUM = 0
      READ*, N
      DO 6 K = 1,N
          ISUM = ISUM +(-1)**(K-1)
6     CONTINUE
      PRINT*, ISUM
      END
```

If the input is:

a.

```
9
```

b.

```
8
```

c.

```
51
```

d.

```
98
```

6. The following program segments may or may not have errors. Identify the errors (if any).

```
1.    INTEGER K, J
      DO 6 K = 1,4
          DO 7 J = K-1,K
              PRINT*, K
6         CONTINUE
7     CONTINUE
      END
```

```
2.    INTEGER K, J
      K = 10
      J = 20
1     IF ( J.GT. K ) THEN
          K = K/2
          GOTO 1
      ENDIF
      END
```

7. Write a FORTRAN 77 program to calculate the following summation:

$$\sum_{k=1}^{200}\left((-1)^k \frac{5k}{k+1}\right)$$

8. Write a program that reads the values of two integers M and then prints all the odd numbers between the two integers.(Note: M may be less than or equal to N or vice-versa).
9. Write a program that prints all the numbers between two integers M and N which are divisible by an integer K. The program reads the values of M, N and K.
10. Write a program that prints all the perfect squares between two integers M and N. Your program should read the values of M and N. (Note: A perfect square is a square of an integer, example $25 = 5 \times 5$)
11. Using nested WHILE loops, print the multiplication table of integers from 1 to 10. Each multiplication table goes from 1 to 20. Your output should be in the form :

```
1 * 1 = 1
1 * 2 = 2
:
1 * 20 = 20
:
10 * 1 = 10
10 * 2 = 20
:
10 * 20 = 200
```

12. Rewrite the program in the previous question using nested **DO** loops.

13. Complete the **PRINT** statement in the following program to produce the indicated output.

```
      DO 1 K = 1,5
         PRINT*,
1     CONTINUE
      END
```

OUTPUT:

```
=****
*=***
**=**
***=*
****=
```

14. Complete the following program in order to get the required output.

```
      DO 10 K = 10,    (1)      ,    (2)
         PRINT*,(  (3)   , L =    (4)   , K )
10    CONTINUE
      END
```

The required output is :

```
5       6   7   8   9   10
5       6   7   8   9
5       6   7   8
5       6   7
5       6
5
```

5.9 Solutions to Exercises

Ans 1.

T F

12

33

6

25

7 4 -3

1 0 50

10

7

5

4

9
1
1 1
1 2
1 3
2
2 1
2 2
2 3
HELLO
HELLO
60
DONE

Ans 2.

1. 76
2. INFINITE LOOP

Ans 3.

II , III , IV , V

Ans 4.

I)

```
      DO 10 ID = N , 891234 , 10
         PRINT*, ID
10    CONTINUE
```

II)

```
      SUM = 0
      DO 3 L = 1 , 15 , 3
         DO 2 J = -L , 0 , 1
            SUM = SUM + J
2        CONTINUE
3     CONTINUE
```

Ans 5.

A) 1 B) 0 C) 1 D) 0

Ans 6

1) IMPROPER NESTING OF **DO** LOOPS
2) INFINITE LOOP

Ans 7.

```
      REAL SUM
      INTEGER K
      SUM = 0
      DO 10 K = 1 , 200
          SUM = SUM + (-1) ** K * (REAL(5*K) / ( K+1))
10    CONTINUE
      PRINT*, SUM
      END
```

Ans 8.

```
      INTEGER M , N , TEMP
      READ*, M , N
      IF( M .LT. N ) THEN
          TEMP = N
          N = M
          M = TEMP
      ENDIF
      DO 5 L = M , N
          IF( L/2 * 2 .NE. L ) PRINT*,L
5     CONTINUE
      END
```

Ans 9.

```
      INTEGER M , N , K , TEMP
      READ*, M , N , K
      IF( M .LT. N ) THEN
          TEMP = N
          N = M
          M = TEMP
      ENDIF
      DO 5 L = M , N
          IF( L/K * K .EQ. L ) PRINT*,L
5     CONTINUE
      END
```

Ans 10.

```
      INTEGER M , N , TEMP
      READ*, M , N
      IF( M .LT. N ) THEN
          TEMP = N
          N   = M
          M   = TEMP
      ENDIF
      DO 5 L = M , N
          IF( INT(SQRT(REAL(L)) ** 2 .EQ. L )) PRINT*,L
5     CONTINUE
      END
```

Ans 11.

```
      INTEGER I, J
      I = 1
10    IF(I .LE. 10 ) THEN
          J = 1
5         IF( J .LE. 20 ) THEN
               PRINT*, I, ' * ', J, ' = ', I*J
               J = J + 1
               GO TO 5
          ENDIF
          I = I + 1
          GO TO 10
      ENDIF
      END
```

Ans 12.

```
      INTEGER I, J
      DO 10 I = 1 , 10
          DO 10 J = 1 , 20
              PRINT*, I, ' * ', J, ' = ', I*J
10    CONTINUE
      END
```

Ans 13.

```
PRINT*, ('*', J = 1, K-1), '=' , ('*', M = 1 , 5-K)
```

Ans 14.

1) 5 2) -1 3) L 4) 5

6 ONE-DIMENSIONAL ARRAYS

It is fairly common in programs to read a large quantity of input data, process the data and produce the computations as output. Such large amounts of input data cannot be stored in simple variables. We need bigger data structures to store such data in memory. For example, consider a problem to compute the average, given the grades of a number of students as input, and list the grades of those students below average. The grades must be stored in the memory while reading because, after the average is computed, they have to be processed again (to list those below average). For a large number of students, simple variables cannot be used to store the grades. We require structures such as arrays. In this and the following chapter, we introduce data structures that allow storage of large amounts of data.

In the previous chapters, we learnt that a variable represents a single location in the memory. Unlike variables, a one-dimensional *array* (1-D array) represents a group of memory locations. Each member of an array is called an *element*. An element in an array is accessed by the array name followed by a *subscript* (also called an *index*) enclosed in parentheses. Subscripts are *integer* constants or expressions that indicate the location of the element within the array. All elements of an array store the same type of data. Thus all elements in an integer array will contain integer values. In FORTRAN, arrays *must* be declared at the beginning of a program or a subprogram.

6.1 One-Dimensional Array Declaration

Arrays must be declared using a declaration statement. If an integer array is to be declared, then the **INTEGER** declaration statement is used. Similarly, for declaring real, logical or character arrays, the respective declaration statement is used. Before executing a program, a computer should know the total memory space required by the program. Each array declaration informs the computer of the amount of memory space required by that array. Therefore, all arrays must be *declared*.

Example 1: *Declaration of an integer array LIST consisting of 20 elements.*

```
      INTEGER LIST (20)
```

Example 2: *Declaration of a logical array FLAG that consists of 30 elements.*

```
      LOGICAL FLAG (30)
```

Example 3: *Declaration of a character array NAMES that consists of 15 elements with each element of size 20.*

```
      CHARACTER NAMES (15)*20
```

Example 1, declares an array LIST consisting of 20 elements. The first element has the subscript 1 and the last element has the subscript 20. We may also declare arrays with subscript beginning from any integer, positive or negative, other than 1.

Example 4: *Declaration of a real array YEAR used to represent rainfall in years 1983 to 1994.*

```
      REAL YEAR (1983 : 1994)
```

The array YEAR has 12 elements. If an array is declared in the format *array_name (m:n)*, we have to ensure that *n* must be greater than *m*. Also note that both *m* and *n* can be either positive or negative integer as long as *n* is greater than *m*.

Example 5 : *Declaration of a real array TEMP with subscript ranging from -20 to 20.*

```
      REAL TEMP (-20:20)
```

A total of 41 elements in this array can be found using the formula *n - m + 1* where *n* is 20 and *m* is -20.

The declaration statement **DIMENSION** is also used to declare arrays. This statement assumes that the type of the array is implicitly defined. The **DIMENSION** statement can be combined with an explicit type statement declaring the type of the array. If an array is declared using the **DIMENSION** statement, and if the type of the array is not mentioned, it is decided implicitly by the first character of the array name, as in the case of undeclared variables.

Example 6 : *Declaration of arrays using the **DIMENSION** statement.*

```
      DIMENSION ALIST(100), KIT(-3:5), XYZ(15)
      INTEGER XYZ
      REAL BLIST(12), KIT
```

In this example, arrays ALIST, BLIST and KIT are of type **REAL**. Array XYZ is of type **INTEGER**. Since the type of array ALIST is not specified, it is treated as a real variable using the default rule for implicit variables.

6.2 One-Dimensional Array Initialization

The purpose of declaring arrays is to specify the number of elements in each array. By declaring an array, the memory space required by the array is only reserved and not initialized. Arrays can be filled with data using either the assignment statement or the **READ** statement.

6.2.1 *Initialization Using the Assignment Statement*

The following statements illustrate the initialization of arrays using the assignment statement, in different ways:

Example 1: *Declare a real array LIST consisting of 3 elements. Also initialize each element of LIST with the value zero.*

Solution:

```
      REAL LIST(3)
      DO 5 K = 1, 3
         LIST(K) = 0.0
5     CONTINUE
```

Example 2: *Declare an integer array POWER2 with subscript ranging from 0 up to 10 and store the powers of 2 from 0 to 10 in the array.*

Solution:

```
      INTEGER POWER2(0:10)
      DO 7 K = 0, 10
         POWER2(K) = 2 ** K
7     CONTINUE
```

6.2.2 *Initialization Using the READ Statement*

An array can be read as a whole or in part. To read the whole array, we may use the name of the array without subscripts. We can read part of an array by specifying specific elements of the array in the **READ** statement. We may also use the implied loop in reading arrays. Implied loops provide an elegant approach to reading arrays of varying lengths.

The rules that apply in reading simple variables also apply in reading arrays. Each **READ** statement requires a *new* line of input data. If the data in the input line is not enough, the **READ** statement ensures that the data is read from the immediately following input line or lines, until all the elements of the **READ** statement are read.

Example 1: *Read all the elements of an integer array X of size 4. The four input data values are in a single input data line as follows*

```
10, 20, 30, 40
```

Solution 1: (Without Array Subscript)

```
      INTEGER X(4)
      READ*, X
```

Solution 2: (Using an Implied Loop)

```
      INTEGER X(4), K
      READ*, (X(K), K = 1, 4)
```

Both **READ** statements *read* all four elements of the array X. However, in both solutions, only one **READ** statement is executed. Ideally, the four input data values may be placed in one input line. If the four values of the input data appear in more than one input line, then reading continues until all four values are read. The two solutions are equivalent with a subtle difference. The **READ** statement in Solution 2 may be used to read all four elements of the array or fewer than four elements by modifying the implied loop. In the next example, we will read one input data value per line.

Example 2: *Read all the elements of an integer array X of size 4. The four input data values appear in four input data lines as follows*

```
10
20
30
40
```

Solution:

```
      INTEGER X(4), J
      DO 22 J = 1, 4
         READ*, X(J)
22    CONTINUE
```

Notice the layout of the input data. Since four **READ** statements are executed in the **DO** loop, four input data lines are required each with one data value. The input data for this example can also be used for the previous example (Example 1) but the input of the previous example cannot be used for the current one. The next three examples further illustrate reading of one-dimensional arrays.

Example 3: *Read an integer one-dimensional array of size 100.*

Solution 1: (Using a WHILE Loop)

```
      INTEGER A(100), K
      K = 0
66    IF (K.LT.100) THEN
          K = K + 1
          READ*, A(K)
          GOTO 66
      ENDIF
```

Note that we require 100 lines of input with one data value per line since the **READ** statement is executed 100 times.

Solution 2: (Using a DO Loop)

```
      INTEGER A(100), K
      DO 77 K = 1, 100
          READ*, A(K)
77    CONTINUE
```

Note again that we require 100 lines of input with one data value per line since the **READ** statement is executed 100 times.

Solution 3: (Using an implied Loop)

```
      INTEGER A(100), K
      READ*, (A(K), K = 1, 100)
```

Note that we require one line with 100 data values since the **READ** statement is executed only once. Even if the input is given in 100 lines with one data value per line, the implied loop will correctly read the input.

Example 4: *Read the first five elements of a logical array PASS of size 20.* The input is:

```
T, F, T, F, F
```

Solution:

```
      LOGICAL PASS(20)
      INTEGER K
      READ*, (PASS(K), K = 1, 5)
```

Example 5: *Read the grades of N students into an array SCORE. The value of N is the first input data value followed by N data values in the next input line.* Assume the input is:

```
6
55, 45, 37, 99, 67, 58
```

Solution:

```
      INTEGER SCORE(100), K, N
      READ*, N
      READ*, (SCORE(K), K = 1, N)
```

In this example, the value of N is 6 and the six grades in the second input line are stored as the first six elements of the array SCORE. The rest of the array SCORE is not

initialized. Note that the value of N may range from 1 to 100 depending on the first data value in the input. If the input data were given as follows:

```
4
42, 77, 89, 70
```

the value of N will be 4 and only four elements of the array SCORE are initialized. We assume here that the value of N will never go beyond 100 and that there will *k+1* data values in the input where *k* represents the first data value.

6.3 Printing One-Dimensional Arrays

Just as in the case of reading an array, printing an array without subscripts will produce the whole array as output. If some elements of the array are not initialized before printing, question marks appear in the output indicating elements that do not have a value. Each **PRINT** statement starts printing in a new line. If the line is not long enough to print the array, the output is printed in more than one line.

Example : *Read an integer array X of size 4 and print*:

i. the entire array X in one line;

ii. one element of array X per line; and

iii. array elements greater than 0.

Solution:

```
      INTEGER X(4), K
      READ*, X
C PRINTING THE ENTIRE ARRAY IN ONE LINE
      PRINT*, 'PRINTING THE ENTIRE ARRAY'
      PRINT*, X
C PRINTING ONE ARRAY ELEMENT PER LINE
      PRINT*, 'PRINTING ONE ARRAY ELEMENT PER LINE'
      DO 33 K = 1, 4
          PRINT*, X(K)
33    CONTINUE
C PRINTING ARRAY ELEMENTS GREATER THAN 0
      PRINT*, 'PRINTING ARRAY ELEMENTS GREATER THAN 0'
      DO 44 K = 1, 4
          IF(X(K) .GT. 0) PRINT*, X(K)
44    CONTINUE
      END
```

If the input is given as

```
7, 0, 2, -4
```

the output of the program is as follows:

```
PRINTING THE ENTIRE ARRAY
7    0    2    -4
PRINTING ONE ARRAY ELEMENT PER LINE
7
0
2
-4
PRINTING ARRAY ELEMENTS GREATER THAN 0
7
2
```

6.4 Errors in Using One-Dimensional Arrays

There are many errors that may occur in the use of arrays. These errors may appear, if the following rules are not followed:

- Array subscripts must not go beyond the array boundaries.
- Array subscripts must always appear as integer expressions.
- The value assigned to an array element, either using the **READ** statement or the assignment statement, must match in type with the array type. This rule, as in the case of simple variables, does not hold for integer and real variables.
- Arrays must be declared before its elements are initialized.

We will now illustrate a few errors through examples. Assume the following declarations:

```
      INTEGER GRADE(25), LIST(3)
      LOGICAL MEM(20)
      CHARACTER TEXT(5) * 3
```

The following statements illustrate *incorrect* initializations of arrays:

Initialization	Type of Error
`GRADE(26) = 0.0`	array subscript 26 is out of range
`LIST(2.0) = X * 3`	array subscript 2.0 is not an integer
`TEXT(4) = 100`	array TEXT is a character array
`MEM(3) = 'WRONG'`	array MEM is a logical array
`READ*, (GRADE(K), K = 1, 100)`	array GRADE has only 25 elements
`ARR(2) = 3`	ARR is not declared as an array

6.5 Complete Examples on One-Dimensional Arrays

In this section, we illustrate the use of one-dimensional arrays through complete examples.

Example 1: *Counting Odd Numbers: Read an integer N and then read N data values into an array. Print the count of those elements in the array that are odd.*

Solution:

```
      INTEGER A(50), COUNT, N , K
      READ*, N, (A(K), K = 1, N)
      COUNT = 0
      DO 44 K = 1, N
          IF (MOD (A(K), 2) .EQ. 1) COUNT = COUNT + 1
44    CONTINUE
      PRINT 'COUNT OF ODD ELEMENTS = ', COUNT
      END
```

If the input is:

```
7, 35, 66, 83, 22, 33, 1, 89
```

The value of variable N in this example is 7. The next seven input data values are placed in the array. There are 5 odd values among the seven elements of the array. For the given input, the output is as follows:

```
COUNT OF ODD ELEMENTS = 5
```

Example 2: *Reversing a One-Dimensional Array: Write a FORTRAN program that reads an integer one-dimensional array of size N. The program then reverses the elements of the array and stores them in reverse order in the same array. For example, if the elements of the array are:*

33 20 2 88 97 5 71

the elements of the array after reversal should be:

71 5 97 88 2 20 33

The program prints the array, one element per line.

Solution:

```
      INTEGER NUM(100), TEMP
      READ*, N, (NUM(L), L = 1, N)
      DO 41 K = 1, N / 2
          TEMP = NUM(K)
          NUM(K) = NUM(N + 1 - K)
          NUM(N + 1 - K) = TEMP
41    CONTINUE
      DO 22 L = 1, N
          PRINT*, NUM(L)
22    CONTINUE
      END
```

Note that we used an implied loop to read the array and a **DO** loop to print the array. Since the problem asks for an array of size N to be read, we first read N and then use an implied loop to read N elements into the array. One common mistake here is to declare an array of size N. This is not allowed since the size of an array in a declaration statement must be an integer constant (except in the case of subprograms where it may be a dummy argument as we shall see in an example later in this chapter). The array is reversed by exchanging the elements of the array. The expression N+1-K gives the index of the element corresponding to K from the end of the array. Thus, using this expression, the first element is exchanged with the last, the second element is exchanged with the second last and so on. This operation is called *swapping*. The swapping of elements in the array stops at the middle element.

Example 3: *Manipulating One-Dimensional Arrays: Write a FORTRAN program that reads a one-dimensional integer array X of size 10 elements and prints the maximum element and its index in the array.*

Solution:

```
      INTEGER X(10), MAX, INDEX, K
      READ*, X
      MAX = X(1)
      INDEX = 1
      DO 1 K = 2, 10
          IF (X(K) .GT. MAX) THEN
              INDEX = K
              MAX = X(K)
          ENDIF
1     CONTINUE
      PRINT*, 'MAXIMUM:', MAX, ' INDEX:',INDEX
      END
```

In the above program, we need to keep track of the position of the maximum element within the array. The variable MAX stores the current maximum and the variable

INDEX represents the position of the maximum element in the array. Whenever a new maximum is found by the **IF** statement condition, we update both variables MAX and INDEX.

Example 4: *Printing Perfect Squares: Read 4 data values into an array LIST (of size 10) and print those values that are perfect squares (1, 4, 9, 25 .. are perfect squares). Assume that the input is:*

```
81, 25, 10, 169
```

Solution:

```
      INTEGER LIST(10), N, K
      LOGICAL PSQR
C   STATEMENT FUNCTION TO CHECK FOR PERFECT SQUARES
      PSQR(N) = INT(SQRT(REAL(N))) ** 2 .EQ. N
      READ*, (LIST(K), K = 1, 4)
      K = 0
55    IF (K .LE. 4) THEN
          IF(PSQR(LIST(K))) PRINT*, LIST(K)
          K = K + 1
          GOTO 55
      ENDIF
      END
```

In this example, only four elements of the array LIST are initialized by the **READ** statement. The other six elements are not initialized. Notice the use of the logical statement function PSQR that checks whether its argument N is a perfect square. The simple **IF** statements check if the four elements of the array LIST are perfect squares. For the given input, the output is as follows:

```
81
25
169
```

6.6 One-Dimensional Arrays and Subprograms

One-dimensional arrays can be passed to a subprogram or can be used locally within a subprogram. In both the cases, the array must be declared within the subprogram. The size of such an array can be declared as a constant or as a variable. Variable-sized declaration of one-dimensional arrays in a subprogram is allowed only if both the variable size is a dummy argument and the array itself is a dummy argument. The following examples illustrate the use of one-dimensional arrays in a subprogram.

Example 1: *Summation of Array Elements: Read 4 data values into an array LIST (of size 10) and print the sum of all the elements of array LIST using a function SUM.*

Solution:

```
      INTEGER LIST(10), SUM, K
      READ*, (LIST(K), K = 1, 4)
      PRINT*, SUM(LIST, 4)
      END
      INTEGER FUNCTION SUM(MARK, N)
      INTEGER N, MARK(N)
      SUM = 0
      DO 13 J = 1, N
            SUM = SUM + MARK(J)
13    CONTINUE
      RETURN
      END
```

In this example, four elements of the array LIST are read by the **READ** statement. The function SUM is called and the sum of the first four elements of array LIST is printed. The first argument to the function is the one-dimensional array LIST. The second argument is passed as the size of the array. In function SUM, the argument N is used in the declaration of the array MARK. The declaration ***INTEGER*** *MARK(N)* implies that the size of the array MARK is the value of N. This type of declaration is allowed in *functions* and *subroutines* only. The elements of the array MARK are added and the result is returned as the function value.

If the input to this program is as follows:

```
19, 25, 10, 82
```

the output would be as follows:

```
136
```

Example 2: *A Function to Compare One-Dimensional Arrays: Write a program that has a* logical *function* COMPAR. *The function gets A, B, and N as arguments. A and B are integer one-dimensional arrays of equal size. N is an integer that represents the size of arrays A and B. The function compares the elements of A and B. If all elements of A are equal to the corresponding elements of B, the function returns the value .TRUE.. Otherwise, it returns a .FALSE. value. In the main program, N is read. The program also reads two one-dimensional arrays (each of maximum size 100). Only N elements of each array are read. The program then calls the function* COMPAR. *If the value returned is .TRUE., it prints one of the arrays. Otherwise, it prints the two arrays.*

Solution:

```
      LOGICAL FUNCTION COMPAR(A, B, N)
      INTEGER N, A(N), B(N), K
      COMPAR = .TRUE.
      DO 10 K = 1, N
          IF (A(K).NE.B(K)) THEN
              COMPAR = .FALSE.
              RETURN
          ENDIF
10    CONTINUE
      RETURN
      END
      LOGICAL COMPAR
      INTEGER A(100), B(100), K, N
      READ*, N, (A(K), K=1,N), (B(K), K=1,N)
      IF (COMPAR(A,B,N)) THEN
          PRINT*, 'A = B = ', (A(K), K=1,N)
      ELSE
          PRINT*, 'A = ', (A(K), K=1,N)
          PRINT*, 'B = ', (B(K), K=1,N)
      ENDIF
      END
```

Notice how the array declarations are different in the main program from the subprogram. Array A is declared as A(100) in the main program while it is declared with *variable size* as A(N) in the subprogram.

Example 3: *Counting Negative Numbers within a One-Dimensional Array: Write a subroutine* FIND *that takes a one-dimensional array and its size as two input arguments. It returns the count of the negative and non-negative elements of the array.*

Solution:

```
      SUBROUTINE FIND(A, N, COUNT1, COUNT2)
      INTEGER N, A(N), COUNT1, COUNT2, K
      COUNT1 = 0
      COUNT2 = 0
      DO 13 K = 1,N
      IF (A(K).LT.0) THEN
          COUNT1= COUNT1 + 1
      ELSE
          COUNT2= COUNT2 + 1
      ENDIF
13    CONTINUE
      RETURN
      END
```

The variable COUNT1 counts the negative numbers in the array. The variable COUNT2 counts the non-negative integers in the array.

Example 4: *Updating the Values in a One-Dimensional Array: The two input arguments to a certain subroutine* UPDATE *is an array* A *of real numbers and its size* N. *The subroutine replaces the value of every element in* A *with its absolute value. Write the subroutine* UPDATE *and a main program which will invoke (call) the subroutine. The maximum size of the array is 100.*

Solution:

```
      SUBROUTINE UPDATE(A,N)
      INTEGER K, N
      REAL A(N)
      DO 44 K = 1,N
          A(K) = ABS(A(K))
44    CONTINUE
      RETURN
      END
      INTEGER J, N
      REAL A(100)
      READ*, N, (A(J),J=1,N)
      PRINT*, 'THE ORIGINAL ARRAY: ', (A(J),J=1,N)
      CALL UPDATE(A,N)
      PRINT*, 'THE NEW ARRAY: ', (A(J),J=1,N)
      END
```

6.7 Exercises

1. What is printed by the following programs?

```
1.    INTEGER A(3), J
      A(1) = 1
      DO 30 J = 2, 3
          A(J) = 3 * A(J - 1)
30    CONTINUE
      PRINT*, A
      END
```

```
2.    INTEGER X(3), Y(3), K
      LOGICAL Z(3)
      READ*, X
      READ*, Y
      DO 80 K = 1, 3
          Z(K) = X(K) .EQ. Y(K)
80    CONTINUE
      IF(Z(1) .AND. Z(2) .AND. Z(3)) THEN
          PRINT*, 'EQUAL ARRAYS '
      ELSE
          PRINT*, 'DIFFERENT ARRAYS'
      ENDIF
      END
```

Assume the input for the program is:

```
1, 5, 7
7, 5, 1
```

```
3.    INTEGER A(4), B(4), G, K, N
      G(K) = K ** 2
      READ*, A
      DO 60 N = 1, 4
          B(N) = G(A(5 - N))
60    CONTINUE
      PRINT*, B
      END
```

Assume the input for the program is:

```
10, 20, 30, 40
```

```
4.      SUBROUTINE FUN(A)
        INTEGER A(4), TEMP
        TEMP = A(1)
        A(1) = A(2)
        A(2) = A(3)
        A(3) = A(4)
        A(4) = TEMP
        RETURN
        END
        INTEGER LIST(4)
        READ*, LIST
        CALL FUN (LIST)
        PRINT*, LIST
        END
```

Assume the input for the program is:

```
3, 6, 9, 2
```

```
5.      INTEGER X(3), Y(3)
        LOGICAL EQUAL
        READ*, X
        READ*, Y
        IF (EQUAL (X, Y))THEN
            PRINT*, 'EQUAL ARRAYS '
        ELSE
            PRINT*, 'DIFFERENT ARRAYS'
        ENDIF
        END
        LOGICAL FUNCTION EQUAL(X, Y)
        INTEGER X(3), Y(3), K
        LOGICAL Z(3)
        DO 45 K = 1, 3
            Z(K) = X(K) .EQ. Y(K)
        45 CONTINUE
        EQUAL = Z(1) .AND. Z(2) .AND. Z(3)
        RETURN
        END
```

Assume the input for the program is:

```
1, 5, 7
7, 5, 1
```

```
6.      INTEGER A(2), B(3), C(4), D(3)
        READ*, A, D(1)
        READ*, B, D(2)
        READ*, C, D(3)
        PRINT*, A
        PRINT*, B
        PRINT*, C
        PRINT*, D
        END
```

Assume the input for the program is:

```
1,2,3,4,5
6,7,8,9,10
11,12,13,14,15
16,17,18,19,20
```

```
7.      INTEGER A(3), K
        READ*, A
        DO 10 K = 1,3
            A(3) = A(3) + A(K)
10      CONTINUE
        PRINT*, A(3)
        END
```

Assume the input for the program is:

```
10,20,30
```

```
8.      INTEGER X(5), Y(5), N, K
        READ*, N, (X(K),Y(K),K=1,N)
        DO 5 K=X(N),Y(N)
            PRINT*, ('X',J=X(K),Y(K))
5       CONTINUE
        END
```

Assume the input for the program is:

```
4,1,2,3,3,3,4,2,4
```

```
9.      NTEGER A(0:4), K
        DO 10 K = 1,2
            READ*, A
10      CONTINUE
        READ*,(A(K), K = 0,2)
        DO 30 K = 1,20,3
            A(MOD(K,4)) = A(MOD(K,5))
30      CONTINUE
        PRINT*, A
        END
```

Assume the input for the program is:

```
1,2,3,4,5,6,7,8
9,10,11
12,13,14,15
18,19,20
```

```
10.     LOGICAL X(0:4)
        INTEGER J, K
        X(0) = .TRUE.
        DO 30 J = 0,4
            K = MOD(J+1,5)
            X(K) = .NOT. X(J)
30      CONTINUE
        PRINT*, X
        END
```

```
11.     INTEGER A(5), B(5), K
        REAL F, Z
        READ*, (A(K),K=1,4), (B(K),K=1,4)
        Z = F(A,B)
        PRINT*, Z
        END
        REAL FUNCTION F(L,M)
        INTEGER L(5), M(5), K
        F = 0
        DO 10 K = 1,4
            IF (L(K).EQ.M(L(K))) THEN
                F = M(K) + K
            ELSE
                RETURN
            ENDIF
10      CONTINUE
        F = F + K
        RETURN
        END
```

Assume the input for the program is:

```
3,1,2,4,1,2,3,4
```

```
12.     INTEGER A(100), I, J, N
        REAL ENDAVE
        DO 2 I=1,4
            READ*, N, (A(J),J=1,N)
            PRINT*, ENDAVE(A,N)
2       CONTINUE
        END
        FUNCTION ENDAVE(X,V)
        INTEGER V, X(V)
        REAL ENDAVE
        ENDAVE = (X(1)+X(V)) / 2.0
        END
```

Assume the input for the program is:

```
4 5 7 3 1
5 7 3 1 4 5
3 1 5 4
1 2
```

```
13.     INTEGER FUNCTION SUM(X,N)
        INTEGER J, N
        REAL X(N), Z
        Z = 0
        DO 10 J = 1,N
            Z = Z +X(J)
10      CONTINUE
        SUM = Z
        RETURN
        END
        INTEGER SUM
        REAL A(4), B(4)
        READ*, A, B
        PRINT*, SUM (A,2)/SUM(B,3)
        END
```

Assume the input for the program is:

```
4 5 3 4 2 1 1 0
```

```
14.   SUBROUTINE EXCESS(RESULT, OPA, OPB, N)
      INTEGER OPA(10), OPB(10), RESULT(10), CARRY
      CARRY = 0
      DO 10 K = N,1,-1
          RESULT(K+1) = MOD(OPA(K)+OPB(K)+CARRY,10)
          CARRY = (OPA(K)+OPB(K)+CARRY) / 10
10    CONTINUE
      RESULT(1) = CARRY
      RETURN
      END
      INTEGER A(10), B(10), C(10)
      READ*, N
      READ*, (A(K),K=1,N)
      READ*, (B(K),K=1,N)
      CALL EXCESS(C,A,B,N)
      PRINT*,(C(K), K=1,N+1)
      END
```

Assume the input for the program is:

```
7
4 5 6 7 0 9 4
8 3 7 5 2 0 8
```

```
15.   SUBROUTINE INTER(A, NA, B, NB, C, NC)
      INTEGER NA, NB, A(NA), B(NB), C(NA), K, M, NC
      NC = 0
      DO 10 K = 1, NA
          DO 20 M = 1, NB
              IF (A(K).EQ. B(M)) THEN
                  NC = NC + 1
                  C(NC) = A(K)
                  GOTO 10
              ENDIF
20        CONTINUE
10    CONTINUE
      RETURN
      END
      INTEGER X(9), Y(9), Z(9), L, NX, NY, NZ
      READ*, NX, (X(L), L = 1,NX)
      READ*, NY, (Y(L), L = 1,NY)
      CALL INTER (X,NX,Y,NY,Z,NZ)
      PRINT*, (Z(J), J = 1,NZ)
      END
```

Assume the input for the program is:

```
5 12 23 45 65 67 84
4 84 64 12 21
```

2. The following program segments may or may not have errors. For each one of the segments, identify the errors(if any). Assume the following declarations :

```
      INTEGER M(4)
      LOGICAL L
```

```
a.    DO 5 K = 2,5,2
          READ*, M(K-1)
5     CONTINUE
```

Assume the input for the program is:

```
20,40,50,30,60
```

```
b.    DO 10 K = 1,4
          M(K+1) = -K
10    CONTINUE
      END
```

3. Consider the following subroutine :

```
      SUBROUTINE CHECK(A,B,C,N)
      INTEGER A(10), B(5)
      C = 0
      DO 10 M = 1,N
          C = C + A(M)*B(M)
10    CONTINUE
      RETURN
      END
```

If the only declaration and assignment statement in the main program are the following:

```
      INTEGER X(5), M(10), A
      A = 3
```

Which of the following **CALL** statements is correct assuming that X and M have some value ?

```
A)    CALL CHECK(M,X,C)
```

```
B)    CALL CHECK(M(10),X(5),C,5)
```

```
C)    CALL CHECK(M,X,B,A+2)
```

```
D)    CALL CHECK(M,X,N,A)
```

```
E)    CALL CHECK
```

4. The following function returns TRUE if the integer number X is found in an integer array A which has N elements. It returns FALSE otherwise. Complete the missing line.

```
      LOGICAL FUNCTION FOUND(A, X, N)
      INTEGER N, A(N), X, K
      DO 20 K=1,N
          IF(A(K) .EQ. X) THEN
              FOUND = .TRUE.
              ---------------
          ENDIF
20    CONTINUE
      FOUND = .FALSE.
      RETURN
      END
```

5. The following subroutine has 4 parameters: A, N, X and Y, where A is an integer array of size N and X and Y are integer numbers. The subroutine changes each element of A that has the value X by the value Y. Complete the missing line.

```
      SUBROUTINE CHANGE(A, N, X, Y)
      INTEGER N, A(N), X, Y, K
      DO 20 K=1,N
          IF(A(K) .EQ. X) THEN
              ----------------------
          ENDIF
20    CONTINUE
      RETURN
      END
```

6. Write a program to initialize a real 1-D array SERIES with the first 8 terms of the series 1, 4, 16, 64, ...
7. Write a logical function subprogram ZERO that takes a 1-D integer array LIST of size 5 and checks if all the elements of array LIST are zero. Write a main program to test the function.
8. Write a program to read a 1-D integer array X and check if all the elements of array X are in increasing order. Print a proper message.
9. Write a subroutine REVRSE to reverse a 1-D real array DAT with 5 elements. Write a main program to test the subroutine.
10. Write a program which reads the elements of three 1-Dimensional arrays A, B and C each of size N (where N<10). The program stores these elements in an array D of size M (where M = 3×N) such that the elements of D array will be as follows :

    A(1) B(1) C(1) A(2) B(2) C(2) ... A(N) B(N) C(N)

11. Write a program that reads a 1-D integer array of 10 elements and prints the element that appears the maximum number of times. (If there is more than one element, it prints the first one only).
12. Write a program to read a 1-D array AR1 of size 15 and another 1-D array AR2 of size 75. The program then finds and prints the number of occurrences of the array AR1 in the array AR2.
13. Write a program that reads ten integers and stores them into a one-dimensional array X.. The main program then calls a subroutine SUMS passing it the one-dimensional array. The subroutine computes the sum S of all the ten elements and the sum of the square of these ten values. Finally the main program prints the sum S and the sum of the squares S2.

6.8 Solutions to Exercises

Ans 1.

1. 1 3 9
2. DIFFERENT ARRAYS
3. 1600 900 400 100
4. 6 9 2 3
5. DIFFERENT ARRAYS
6. 1 2
   6 7 8
   11 12 13 14
   3 9 15
7. 120
8. X
   XX
   XXX
9. 20 20 13 13 13

10. F F T F T
11. 13.0
12. 3.0
    6.0
    2.5
    2.0
13. 2
14. 1 2 9 4 2 3 0 2
15. 12

Ans 2.

a) End of file encountered (The program needs 2 lines of input)

b) Subscript out of range; m(5) is undefined

Ans 3.

C

Ans 4.

```
      RETURN
```

Ans 5.

```
      A(K) = Y
```

Ans 6.

```
      REAL SERIES(8)
      INTEGER K
      DO 12 K = 1, 8
          SERIES(K) = 4**(K-1)
12    CONTINUE
      END
```

Ans 7.

```
      LOGICAL FUNCTION ZERO(LIST, N)
      INTEGER N, LIST(N), K
      ZERO = .TRUE.
      K = 0
18    IF (K .LE. N .AND. ZERO) THEN
          IF(LIST(K) .NE. 0) ZERO = .FALSE.
          K = K + 1
          GOTO 18
      ENDIF
      RETURN
      END
      LOGICAL ZERO
      INTEGER LIST(5)
      IF (ZERO(LIST, 5)) THEN
          PRINT*, 'ALL ELEMENTS ARE ZEROS'
      ELSE
          PRINT*, 'NOT ALL ELEMENTS ARE ZEROS'
      ENDIF
      END
```

Ans 8.

```
      INTEGER X(3)
      READ*, X
      IF(X(1) .LT. X(2) .AND. X(2) .LT. X(3)) THEN
          PRINT*, 'INCREASING ORDER'
      ELSE
          PRINT*, 'NOT INCREASING ORDER'
      ENDIF
      END
```

Ans 9.

```
      SUBROUTINE REVERSE (DAT)
      REAL DAT(5), TEMP
      TEMP   = DAT(5)
      DAT(5) = DAT(1)
      DAT(1) = TEMP
      TEMP   = DAT(2)
      DAT(2) = DAT(4)
      DAT(4) = TEMP
      RETURN
      END
      REAL DAT(5)
      READ*, DAT
      CALL REVERSE(DAT)
      PRINT*, DAT
      END
```

Ans 10.

```
      INTEGER A(10) , B(10) , C(10) , D(30), N, M, K, J
      READ*, N
      M = 3 * N
      J = 1
      READ*, (A(K), K= 1 ,N),(B(K),K=1,N),(C(K),K=1,N)
      DO 10 K = 1 , N
          D(J)   = A(K)
          D(J+1) = B(K)
          D(J+2) = C(K)
          J = J + 3
10    CONTINUE
      PRINT*, (D(K) , K = 1 ,M)
      END
```

Ans 11.

```
      INTEGER A(10) , FREQ(10) , MAXFRQ , LOC, I, J
      READ*, A
      DO 10 I = 1 ,10
          FREQ(I) = 0
10    CONTINUE
      DO 20 I = 1 ,10
          DO 30 J = 1 ,10
              IF(A(J) .EQ. A(I)) FREQ(I) = FREQ(I) + 1
30        CONTINUE
20    CONTINUE
      MAXFRQ = FREQ(1)
      LOC  = 1
      DO 40 J = 1 ,10
          IF(MAXFRQ .LT. FREQ(J)) THEN
              MAXFRQ = FREQ(J)
              LOC = J
          ENDIF
40    CONTINUE
      PRINT*, ' THE ELEMENT WITH IS MAX APPEARANCE IS ',A(LOC)
      END
```

Ans 12.

```
      INTEGER COUNT , AR1(15),AR2(75), K, COUNT, M
      LOGICAL FOUND
      READ*,AR1
      READ*,AR2
      COUNT = 0
      DO 10 K=1,61
          FOUND = .TRUE.
          DO 20 M = K,K+14
              IF(AR1(M-K+1).NE. AR2(M)) FOUND=.FALSE.
20        CONTINUE
          IF(FOUND) COUNT = COUNT+1
10    CONTINUE
      PRINT*,'COUNT = ' , COUNT
      END
```

Ans 13.

```
      INTEGER X(10) , S , S2, J
      READ*, (X(J), J =1,10)
      CALL SUMS(X , S ,S2)
      PRINT*, ' THE SUM OF VALUES =', S
      PRINT*, ' THE SUM OF THE SQUARE OF VALUES =', S2
      END
      SUBROUTINE SUMS (X , S ,S2)
      INTEGER X(10) , S , S2, K
      S = 0
      S2 = 0
      DO 20 K = 1 ,10
          S = S + X(K)
          S2 = S2 + X(K) ** 2
20    CONTINUE
      RETURN
      END
```

7 TWO-DIMENSIONAL ARRAYS

A two-dimensional array (2-D array) is a tabular representation of data consisting of rows and columns. A two-dimensional array of size $m \times n$ represents a matrix consisting of m rows and n columns. Figure 1 shows a two-dimensional array X of size 2×3. An element in a two-dimensional array is addressed by its row and column; for example, X(2,1) refers to the element in row 2 and column 1 which has a value 6.

4	2	5
6	7	3

Figure 1 : A two-dimensional array X of size 2×3

Two-dimensional arrays can be pictured as a group of one-dimensional arrays. If we consider a one-dimensional array as a column, then a two-dimensional array X of size 2 × 3 can be considered as consisting of three one-dimensional arrays; each one-dimensional array containing 2 elements. In fact, since each location in the memory has a single address, the computer stores a two-dimensional array as a one-dimensional array with column 1 first, followed by column 2 and so on. Figure 2 shows the storage of array X (Figure 1) in the memory.

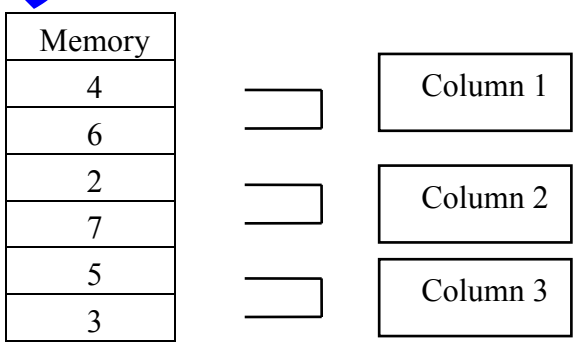

Figure 2 : Storage of the Two-Dimensional Array X in Memory

7.1 Two-Dimensional Array Declaration

Two-dimensional arrays must be declared using declaration statements like **INTEGER**, **REAL** etc. or the **DIMENSION** statement. The array declaration consists of the name of the array followed by the number of rows and columns in parentheses. This information in the declaration statements is required in order to reserve memory space.

For example, if an array X is declared with 2 rows and 3 columns, there are six elements in the array. Therefore, six memory locations must be reserved for such an array.

Example 1 : Declaration of an integer array MAT consisting of 3 rows and 5 column.

```
      INTEGER MAT(3,5)
```

Example 2 : Declaration of a character array CITIES that consists of 9 elements in 3 rows and 3 columns and each element is of size 15.

```
      CHARACTER CITIES (3,3) * 15
```

Example 3 : Declaration of arrays using the DIMENSION statement.

```
DIMENSION X(10,10), M(5,7), Y(4,4)
INTEGER X
REAL M
```

In this example, arrays M and Y are of type **REAL**. Array X is of type **INTEGER**. Note that the type of arrays M and Y is specified in the two declaration statements. The type of Y is not specified and is taken as **REAL** by default.

Example 4 : More array declarations: *Consider the following declarations :*

```
      DIMENSION C(10,10), NUM(0:2, -2:1), VOL(4,2)
      INTEGER ID(3,3)
      REAL MSR(100,100), Z(4:7,8)
      CHARACTER WORD(5,5)*3, C
      LOGICAL TF(5,7)
```

Arrays ID, NUM are integer arrays. Arrays MSR, VOL, Z are real arrays. Array ID has a total of 9 elements in its 3 rows and 3 columns. The starting subscript value of row and column of each array is assumed to be 1 unless it is specified otherwise. In the declaration of arrays NUM and Z, the starting subscript is different than 1. Array NUM has 12 elements with rows numbered as 0, 1, 2; and columns numbered as -2, -1, 0, 1. Array Z has 32 elements with rows numbered from 4 up to 7 and columns numbered from 1 up to 8. Array WORD is a character array that has 5 rows and 5 columns, and stores 3 characters in each element. Array C is a character array and can store 1 character in each of its 100 elements (10 rows and 10 columns). Array TF is a logical array with 35 elements in 5 rows and 7 columns; each can store either a .TRUE. or a .FALSE. value.

7.2 Two-Dimensional Array Initialization

A two-dimensional array can be initialized in two possible ways. We can initialize either by rows or by columns. Initializing row after row is known as row-wise initialization. Similarly, initializing column after column is known as column-wise initialization. Remember, a two-dimensional array is always stored in the memory as a one-dimensional array column by column. The initialization may be done using assignment statements or **READ** statements.

7.2.1 Initialization Using the Assignment Statement

Example 1: Declare an integer array ID consisting of 3 rows and 3 columns and initialize array ID row-wise as an identity matrix (i.e. all elements of the main diagonal must be 1 and the rest of the elements must be 0).

Solution:

```
      INTEGER ID(3,3), ROW, COL
C INITIALIZING ROW-WISE
      DO 17 ROW = 1, 3
          DO 17 COL = 1, 3
              IF (ROW .EQ. COL) THEN
                  ID(ROW, COL) = 1
              ELSE
                  ID(ROW, COL) = 0
              ENDIF
17    CONTINUE
```

In this example, nested do loops are used. In fact, we need the nested loops to go to each element of a two-dimensional array. Note here that the index of the outer do loop is ROW which is also the row subscript of array ID. The inner loop index COL corresponds to the columns (the use of the variables ROW and COL has no significance; we could have used any other **INTEGER** variables). Notice how the value of COL varies within each iteration of the outer loop. When the value of ROW is 1, COL changes its value in the following sequence : 1, 2, 3, and 4. This means the first row has been initialized. Similarly, the next two rows are initialized. Since we initialized row after row, the array ID is initialized row-wise.

In general, if the outer loop index is the row subscript, then we are moving row-wise inside the array. Similarly, if the outer loop index is the column subscript, then we are moving column-wise inside the array.

Example 2 : Declare a real array X consisting of 2 rows and 3 columns and initialize array X column-wise. Each element of array X should be initialized to its row number.

Solution:

```
      REAL X(2,3)
      INTEGER J, K
C INITIALIZING COLUMN-WISE
      DO 27 J = 1, 3
          DO 27 K = 1, 2
              X(K, J) = K
27        CONTINUE
```

7.3 Initialization Using the READ statement

As was the case in one-dimensional arrays, a two-dimensional array can be read as a whole or in part. To read the entire array, we may just use the name of the array without subscripts. In such case, the array is read column-wise. We can read part of an array by specifying specific elements of the array in the **READ** statement. We can either read row-wise or column-wise. Remember that each **READ** statement requires a new line of input data. If the data in the input line is not enough, the **READ** statement ensures that the data is read from the immediately following input line or lines, until all the elements of the **READ** statement are read

Example 1: Read all the elements of an integer array MATRIX of size 3×3 column-wise (i.e. the first element of input data is the first element of the first column of MATRIX, the second element of input data is the second element of the first column, the third element of input is the third element of the first column, the fourth element of input is the first element of the second column, and so on).

The input data is given as follows:

```
3   4   8
5   9   2
1   6   0
```

The contents of array MATRIX after reading the input data is as follows:

3	5	1
4	9	6
8	2	0

Solution 1: (Without Array Subscripts)

```
      INTEGER MATRIX(3, 3)
C READING COLUMN-WISE
      READ*, MATRIX
```

Solution 2: (Using Implied Loops)

```
      INTEGER MATRIX(3, 3), J, K
C READING COLUMN-WISE
      READ*,((MATRIX(K,J), K = 1, 3), J =1, 3)
```

Solution 3: (Using DO and Implied Loop)

```
      INTEGER MATRIX(3, 3), J, K
C READING COLUMN-WISE
      DO 28 J = 1, 3
         READ*, (MATRIX(K,J), K = 1, 3)
28    CONTINUE
```

In all the three solutions, the array MATRIX is read column-wise. In Solution 1, the array MATRIX is read without any subscripts. In such cases, the computer reads the array column-wise, since all arrays are stored in the memory column-wise. In Solution 2, the outer loop index is J which corresponds with the column. Hence, the array is read column-wise. In Solution 3, the outer loop index is also J and, therefore, the array is read column-wise. The difference between the three solutions is that in Solution 1 and 2, only one **READ** statement is executed and, therefore, only one input line of data is required. If the input data is not given in one line, then data is read from the next line or the one after, until all data is read. In Solution 3, since three **READ** statements are executed, a minimum of three lines of input data is required.

Example 2: Read all the elements of an integer array X of size 3×5 row-wise (i.e. the first element of input data is the first element of the first row of array X, the second element of input is the second element of the first row, the third element of input is the third element of the first row, the fourth element of input is the fourth element of the first row, the fifth element of input is the fifth element of the first row, the sixth element of input is the first element of the second row and so on).

The input data is given as follows:

```
7   5   9   3   2
4   6   5   9   2
1   2   7   6   0
```

The contents of array X after reading the input data is as follows:

7	5	9	3	2
4	6	5	9	2
1	2	7	6	0

Solution 1 : (Using Implied Loops)

```
      INTEGER X(3, 5), J, K
      READ*, ((X(K, J) , J = 1, 5), K = 1, 3)
```

Solution 2 : (Using DO and an implied Loop)

```
      INTEGER X(3, 5), J, K
C READING COLUMN-WISE
      DO 33 K = 1, 3
         READ*, (X(K,J), J = 1, 5)
33    CONTINUE
```

In both solutions, the array X is read row-wise, since the outer loop index is K which corresponds to the row of array X. The difference between the two solutions is that in Solution 1, only one **READ** statement is executed and, therefore, only one input line of data is required. If the input data is not given in one line, then data is read from the next line or the one after, until all data is read. In Solution 2, since three **READ** statements are executed, a minimum of three lines of input data is required.

7.4 Printing Two-Dimensional Arrays

Just as in the case of reading a two-dimensional array, printing an array without subscripts will produce the whole array as output. In such a case, the array is printed column-wise. If some elements of the array are not initialized before printing, question marks appear in the output indicating elements that do not have a value. Each **PRINT** statement starts printing in a new line. If the line is not long enough to print the array, the output is printed in more than one line.

Example: Read a 3 ×3 integer array WHT column-wise and print:

i. the entire array row-wise in one line;

ii the entire array column-wise in one line;

iii. one row per line;

iv. one column per line ;

v. the sum of column 3 ;

Solution:

```
      INTEGER WHT(3, 3), SUM, J, K
C READING WHT COLUMN-WISE
      READ*, WHT
C PRINTING THE ENTIRE ARRAY WHT ROW-WISE
      PRINT*, 'PRINTING THE ENTIRE ARRAY ROW-WISE'
      PRINT*, (WHT(K, J), J = 1, 3), K = 1, 3)
C PRINTING THE ENTIRE ARRAY WHT COLUMN-WISE
      PRINT*, 'PRINTING THE ENTIRE ARRAY COLUMN-WISE'
      PRINT*, WHT
C PRINTING ONE ROW OF WHT PER OUTPUT LINE
      PRINT*,'PRINTING ONE ROW PER LINE'
      DO 35 K = 1, 3
         PRINT*, (WHT(K,J), J = 1, 3)
35    CONTINUE
C PRINTING ONE COLUMN OF WHT PER OUTPUT LINE
      PRINT*, 'PRINTING ONE COLUMN PER LINE'
      DO 45 J = 1, 3
         PRINT*, (WHT(K,J), K = 1, 3)
45    CONTINUE
C PRINTING THE SUM OF COLUMN 3
      SUM = 0
      DO 55 K = 1, 3
         SUM = SUM + WHT (K , 3)
55    CONTINUE
      PRINT*, 'SUM OF COLUMN 3 IS', SUM
      END
```

If the input is

```
5, 2, 0
3, 1, 8
4, 6, 7
```

The contents of WHT after reading are as follows:

5	3	4
2	1	6
0	8	7

The output of the program is as follows :

```
PRINTING THE ENTIRE ARRAY ROW-WISE
    5   3   4   2   1   6   0   8   7
PRINTING THE ENTIRE ARRAY COLUMN-WISE
    5   2   0   3   1   8   4   6   7
PRINTING ONE ROW PER LINE
    5   3   4
    2   1   6
    0   8   7
PRINTING ONE COLUMN PER LINE
    5   2   0
    3   1   8
    4   6   7
SUM OF COLUMN 3 IS 17
```

7.5 Complete Examples on Two-Dimensional Arrays

In this section, we illustrate the use of two-dimensional arrays through complete examples.

Example 1: *More on Reading Two-Dimensional Arrays: Write a FORTRAN program that reads a two dimensional array of size 5 × 4 row-wise. Each value is read from a*

separate line of input. The program then prints the same array column-wise such that the elements of the first column are printed on the first line of output and the elements of the second column are printed on the second line of output and so on.

Solution :

```
      INTEGER TDIM(5 , 4) , ROW , COL
      DO 10 ROW = 1, 5
          DO 12 COL = 1, 4
              READ*, TDIM(ROW , COL)
12        CONTINUE
10    CONTINUE
      DO 30 COL = 1, 4
          PRINT*, (TDIM(ROW , COL), ROW = 1 , 5)
30    CONTINUE
      END
```

Let us first consider the reading segment. Reading is done using two nested loops. The outer loop index corresponds to the rows of the two-dimensional array. The inner one corresponds to the columns. Hence, the array TDIM is read row-wise. Note that the **READ** statement is executed 20 times and therefore 20 input lines are required with one data value per line.

In the printing segment, we used an implied loop inside a **DO** loop. Remember that we were asked to print each column on one line of output. This tells us that each column must be printed using one and only one **PRINT** statement. Using two nested **DO** loops will cause each element to be printed on a separate line. Therefore, we used an implied loop for the elements of the columns. Consider the case of the first column. The value of COL is fixed to 1 by the **DO** loop whereas the value of ROW in the implied loop varies from 1 to 5 covering all the elements of the first column. The same logic applies to the rest of the columns.

Consider next the following segment as a substitute for the reading segment in the above program.

```
      READ*, ((TDIM(ROW,COL), COL= 1, 4), ROW= 1, 5)
```

In the previous reading segment, we used nested **DO** loops and the data values were given one in each line. Here, we use nested implied loops. When using nested implied loops, the values can be provided either on one line or on multiple lines. This results from the fact that in the nested **DO** loops, we execute $5 \times 4 = 20$ **READ** statements and each statement takes input from a different line. In the nested implied loops, we execute only one **READ** statement.

In general, the index of the outer loop indicates the way the array is read or printed. If the outer loop index represents the row, the array is read or printed row-wise. If the outer loop index represents the column, the array is read or printed column-wise.

Example 2: *Summation of Even Numbers in a Two-Dimensional Array: Write a FORTRAN program that reads a two-dimensional array of size 3 × 4 column-wise. It then computes and prints the sum of all even numbers in the array.*

Solution:

```
      INTEGER A(3,4), SUM, J, K
      READ*, ((A(K,J), K = 1, 3), J = 1, 4)
      SUM = 0
      DO 1 K = 1, 3
          DO 2 J = 1, 4
              IF (MOD(A(K,J), 2) .EQ. 0) THEN
                  SUM = SUM + A(K,J)
              ENDIF
2         CONTINUE
1     CONTINUE
      PRINT*, SUM
      END
```

In this example, after reading the array column-wise, we go to each element of the array A using the nested **DO** loops. The intrinsic function MOD is used to check if the remainder is zero when each element is divided by two. Only those elements in the array which return a zero value for the function MOD are added to the variable SUM.

Example 3 : *Manipulating Two-Dimensional Arrays: Write a FORTRAN program that reads a two-dimensional array of size 3 × 3 row-wise. The program finds the minimum element in the array and changes each element of the array by subtracting the minimum from each element. Print the updated array row-wise in one output line.*

Solution:

```
      INTEGER A(3,3), MIN, J, K
      READ*, ((A(K,J), J = 1, 3), K = 1, 3)
      MIN = A(1,1)
      DO 3 K = 1, 3
          DO 3 J = 1, 3
              IF (A(K,J) .LT. MIN) THEN
                  MIN = A(K,J)
              ENDIF
3     CONTINUE
      DO 4 K = 1, 3
          DO 4 J = 1, 3
              A(K,J) = A(K, J) - MIN
4     CONTINUE
      PRINT*, ((A(K,J), J = 1, 3), K = 1, 3)
      END
```

The array A cannot be changed unless the minimum element in the array is found. All the elements in the array are checked for the minimum element in the first nested **DO** loop. The array is updated in the second nested **DO** loop by replacing each element of the array by subtracting the minimum from that element.

7.6 Two-Dimensional Arrays and Subprograms

Two-dimensional arrays can be passed to a subprogram or can be used locally within the subprogram. Unlike one-dimensional arrays, it is not recommended to pass a variable-sized two-dimensional array to a subprogram (even though this does not produce an error, it may give wrong results). Whenever a two-dimensional array is passed to a subprogram, the row and column size of the array may be declared using a constant in both the main and the subprogram.

Example 1: *Counting Zero Elements: Read a 3 × 2 integer array MAT row-wise. Using a function COUNT, count the number of elements in MAT with the value equal to 0.*

Solution:

```
      INTEGER MAT(3,2), COUNT, J, K
      READ*, (MAT (K, J), J = 1, 2), K =1, 3)
      PRINT*, 'COUNT OF ELEMENTS WITH VALUE 0 IS ', COUNT (MAT)
      END
      INTEGER FUNCTION COUNT(MAT)
      INTEGER MAT(3,2), J, K
      COUNT = 0
          DO 77 K = 1, 3
              DO 77  J = 1, 2
                  IF(MAT(K, J) .EQ. 0) COUNT = COUNT + 1
77        CONTINUE
      RETURN
      END
```

The input of the program is

```
12, 0, 1, 9, 2, 0
```

The output of the program is as follows:

```
COUNT OF ELEMENTS WITH VALUE 0 IS       2
```

In this example, another possibility is to call the function COUNT by passing three arguments: MAT, M and N where M and N are the variables representing the row and the column size of array MAT. The declaration of MAT within the function COUNT may then be given as follows: **INTEGER** MAT(M,N). This type of variable-sized two-dimensional array declaration is allowed in a subprogram. However, the use of such declarations is not recommended due to reasons beyond the scope of this book.

Example 2: *Addition of Matrices: Write a subroutine CALC(A, B, C, N) that receives 2 two-dimensional arrays A and B of size 10 × 10. It returns the result of adding the two arrays (matrices) in another array C of the same size.*

Solution:

```
      SUBROUTINE CALC(A, B, C, N)
      INTEGER A(10,10), B(10,10), C(10,10), N
      DO 10 K = 1,N
          DO 15 J = 1,N
              C(K,J) = A(K,J) + B(K,J)
15        CONTINUE
10    CONTINUE
      RETURN
      END
```

7.7 Common Errors in Array Usage

We have already seen errors that may occur in the use of one-dimensional arrays in the previous chapter. Such errors can occur in using two-dimensional arrays as well. The following errors are commonly seen while using arrays :

1. Array declaration is missing: All arrays must be declared. Otherwise, a message would appear as '**FUNCTION** array name IS NOT DEFINED.' Since the array declaration is missing, the computer assumes it to be a function. Therefore, the misleading message appears.

2. Array subscript is out-of-bounds: This error occurs when an array subscript is outside the range of the array elements. For example, for a one-dimensional array X declared as **INTEGER** X(10), the expression X(12) would produce an error. Similarly, in a 2-D array Y declared as **INTEGER** Y (-3:2, 5), the expression Y(-5,1) would produce an error.
3. Array subscript is not an integer: All array subscripts must be integers. This error occurs when an array subscript is real. For example, for a one-dimensional array X declared as **INTEGER** X(10), the expression X(2.0) would produce an error. Similarly, in a 2-D array Y of size 3×2, an expression Y(1,3.0) would produce an error.
4. Array size is a variable in the main program: All array sizes must be **integer constants**, if the array is declared in the main program. This error occurs when an array subscript is a variable. For example, a one-dimensional array X declared in a main program as **INTEGER** X(N) would produce an error. In a subprogram, a declaration such as **INTEGER** X(N) is valid as long as **both** X and N are dummy arguments. Similar declarations can be made for two-dimensional arrays as long as the array name, its column-size and its row-size are dummy arguments. Such declarations (for example **INTEGER** Y(M,N)) are valid in a subprogram but may not be used due to reasons beyond the scope of this book.

7.8 Exercises

1. What is printed by the following programs ?

```
1.      INTEGER X(3,3), J
        READ*, X
        PRINT*, X
        PRINT*, (X(J,J), J = 1, 3)
        PRINT*, (X(J,3), J = 1, 3)
        END
```

Assume the input is:

```
1, 5, 7
7, 5, 1
3, 8, 9
```

```
2.      REAL B(2,3), F
        INTEGER J, K
        F(X, Y) = X + Y * 2
        READ*, ((B(J,K), K = 1, 2), J = 1, 2)
        DO 2 J = 1, 2
            B(J,3) = F(B(J,1), B(J,2))
2       CONTINUE
        PRINT*, B
        END
```

Assume the input is:

```
10, 20, 30, 40
```

```
3.      SUBROUTINE ADD(A, B, C)
        INTEGER A(2,2), B(2,2), C(2,2) , J, K
        DO 33 J = 1, 2
            DO 22 K = 1, 2
                C(J,K) = A(J,K) + B(J,K)
22          CONTINUE
33      CONTINUE
        RETURN
        END
        INTEGER X(2,2), Y(2,2), Z(2,2)
        READ*, X, Y
        CALL ADD (X, Y, Z)
        PRINT*, Z
        CALL ADD (Z, Y, X)
        PRINT*, X
        END
```

Assume the input is:

```
3, 6, 9, 2
7, 4, 5, 1
```

```
4.      INTEGER A(3,3) , J, K
        READ*, ((A(K,J),K=1,3),J=1,3)
        PRINT*, A
        PRINT*, ((A(K,J),J=1,2),K=1,3)
        PRINT*, A(3,2)
        PRINT*, (A(K,2),K=3,1,-2)
        END
```

Assume the input is:

```
1 2 3
4
5 6 7 8
9
```

```
5.      INTEGER A(2,2) , J, K
        READ*, A
        DO 3 J = 1,2
            PRINT*, (A(J,K), K=1,2)
3       CONTINUE
        END
```

Assume the input is:

```
1 2 3 4
```

```
6.      INTEGER TDAR(3,3), ODAR(10), ROW, COL, J, K, M, N
        NUM(M,N) = M + N - 1
        READ*, TDAR
        READ*,ROW,COL
        DO 10 J = 1,3
            DO 10 K = 1,3
                ODAR(NUM(J,K)) = TDAR(J,K)
10       CONTINUE
        PRINT*, ODAR(NUM(ROW,COL)), ODAR(NUM(COL,ROW))
        END
```

Assume the input is:

```
9 6 4 3 2 1 8 5 7
2 3
```

```
7.      INTEGER A(2,2), B(2,2), C(2,2), X, Y, K, M
        D(M,N) = M + N
        READ*, A, B
        DO 35 K = 1,2
            DO 35 M = 1,2
                X = A(K,M)
                Y = B(K,M)
                C(M,K) = D(X,Y)
35      CONTINUE
        DO 22 K = 1,2
            PRINT*, (C(K,M), M=1,2)
22      CONTINUE
        END
```

Assume the input is:

```
3 7 2 6
5 8 4 1
```

```
8.      INTEGER A(10,10), B(10), L, K, N
        READ*, N, ((A(K,L),K=1,N),L=1,N), (B(K),K=1,N)
        PRINT*, C(A,B,N)
        END
        REAL FUNCTION C(A,B,N)
        INTEGER A(10,10),B(10), L, N
        C = 0.0
        DO 44 L = 1,N
            IF (L/3*3 .NE.L) B(L) = A(L,L)
            C = B(L) * A(L,L)
44      CONTINUE
        RETURN
        END
```

Assume the input is:

```
3 1 1 1 2 2 2 3 3 3 4 4 4
```

```
9.      INTEGER A(5,5), J, K, M, N
        READ*, N, ((A(K,J),J=1,N),K=1,N)
        CALL TEST(A,N,M)
        PRINT*, M
        END
        SUBROUTINE TEST (X,Y,Z)
        INTEGER X(5,5), Y, Z, J, K
        Z = X(1,1)
        DO 10 K = 1,Y
            DO 10 J = 1, Y
                IF (Z.GT.X(K,J))Z=X(K,J)
10      CONTINUE
        RETURN
        END
```

Assume the input is:

```
3 1 3 6 -3 0 4 5 9 -1
```

2. Assume the array declaration :

```
        INTEGER Z(10,10)
```

is given. Which of the following **READ** statements will read the array column-wise if the data is given one value per line ? :

```
I.      READ*, Z
```

```
II    DO 20 J = 1,10
      READ*, (Z(K,J),K=1,10)
20    CONTINUE
```

```
III.  DO 10 K = 1,10
          DO 10 J = 1,10
              READ*, Z(J,K)
10 CONTINUE
```

3. Complete the missing parts in the program given below to construct the following matrix :

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

```
      INTEGER A(4,4), K, L
      DO 10 K =1,4
      DO 10        (1)
          IF (       (2)       ) THEN
          A(K,L) =    (3)
          ELSE
          A(K,L) =    (4)
          ENDIF
10    CONTINUE
      END
```

4. Write a program to initialize row-wise each element of a real 2-D array PRD of size 3 × 4 with the product of its row and column numbers. Print this array column-wise.
5. Write a function subprogram IDINIT that takes a 2-D integer array IMAT of size 3 × 3 and initializes the array as an identity matrix. Write a main program to test the function.
6. Write a program to read a 2-D integer array X of size 3 × 4. Store the sum of each row in a 1-D array ROW and the sum of each column in a 1-D array COL. Print arrays ROW and COL.
7. Write a FORTRAN program that reads an (8×10) 2-D **REAL** array TAB row-wise and finds the percentage of elements in array TAB that are perfect squares. (Hint: 25 is a perfect square since 25 = 5 × 5).
8. Write a FORTRAN program that reads an integer N and then reads a two dimensional (N × N) array MAT row-wise. The program prints the column in an array MAT whose sum is the maximum. Assume N is less than or equal to 10. For example, if N is 3 and if MAT is as follows:

$$\begin{bmatrix} 2 & 1 & 4 \\ 3 & 5 & 7 \\ 8 & 2 & 9 \end{bmatrix}$$

then the output should be:

```
4 7 9
```

7.9 Solutions to Exercises

Ans 1.

1. 1 5 7 7 5 1 3 8 9
   1 5 9
   3 8 9
2. 10.0 30.0 20.0 40.0 50.0 110.0
3. 10 10 14 3
   17 14 19 4
4. 1 2 3 4 5 6 7 8 9
   1 4 2 5 3 6
   6
   6 4
5. 1 3
   2 4
6. 1 1
7. 8 15
   6 7
8. 12.0
9. -3

Ans 2.

I , II , III

Ans 3

1) L = 1 , 4 2) K + L .EQ. 5 3) 1 4) 0

Ans 4.

```
      REAL PRD(3,4)
      INTEGER J, K
      DO  10   K = 1, 3
          DO 20  J = 1, 4
              PRD(K, J) = K * J
20        CONTINUE
10    CONTINUE
      PRINT*, PRD
      END
```

Ans 5.

```
      SUBROUTINE IDINIT(IMAT)
      INTEGER IMAT(3,3), J, K
      DO  77    K = 1, 3
          DO  77  J = 1, 3
          IMAT(K, J) = 0
          IF (K .EQ. J) IMAT(K, J) = 1
77    CONTINUE
      RETURN
      END
      INTEGER IMAT(3,3), K
      READ*, IMAT
      CALL IDINIT(IMAT)
      DO 77  K = 1, 3
          PRINT*, IMAT(K,1),IMAT(K,2),IMAT(K,3)
77    CONTINUE
      END
```

Ans 6.

```
      INTEGER X(3,4) , ROW(3) , COL(4), J, K
      READ*, X
      DO 55  K = 1, 3
          ROW(K) = 0
          DO 55   J = 1, 4
              ROW (K) = ROW(K) + X(K, J)
55    CONTINUE
      DO 66  J = 1, 4
          COL(J) = 0
          DO 66 K = 1, 3
              COL (J) = COL(J) + X(K, J)
66    CONTINUE
      PRINT*, ROW
      PRINT*, COL
      END
```

Ans 7.

```
      INTEGER CNT, I, J
      REAL TAB(8,10)
      DO 10 I = 1 ,8
          READ*, (TAB(I,J), J =1,10)
10    CONTINUE
      CNT = 0
      DO 20 I = 1 ,8
          DO 30 J = 1 ,10
             IF(INT(SQRT(TAB(I,J)))**2.EQ.TAB(I,J)))CNT=CNT+1
30        CONTINUE
20    CONTINUE
      PER = CNT / 80.0 * 100
      PRINT*, ' THE PERCENTAGE = ' , PER
      END
```

Ans 8.

```
      INTEGER MAT(10,10) , N , SUM , MAXSUM , COL, I, J
      READ*, N
      DO 10 I = 1 ,N
          READ*, (MAT(I,J), J =1,N)
10    CONTINUE
      SUM = 0
      COL = 1
      DO 20 K = 1 ,N
          SUM = SUM + MAT(K,I)
20    CONTINUE
      MAXSUM = SUM
      DO 30 J = 2 , N
          SUM = 0
          DO 40 K = 1 , N
              SUM = SUM + MAT(K,J)
40        CONTINUE
          IF(SUM .GT. MAXSUM) THEN
              MAXSUM = SUM
              COL = J
          ENDIF
30    CONTINUE
      PRINT*,(MAT(K,COL),K = 1, N)
      END
```

8 OUTPUT DESIGN AND FILE PROCESSING

8.1 Output Formatting

The print statement we have been using in the previous chapters is a list-directed output statement. In list-directed output, the output list determines the precise appearance of printed output. In other words, we have no control over the format of the output. To control the manner in which the output is printed or to produce an output in a more readable form, we use **FORMAT** statements. To use a **FORMAT** statement, we must modify the **PRINT** statement by replacing the '*' with a **FORMAT** statement label. The general form of a formatted **PRINT** statement is

```
      PRINT K, expression list
```

The **FORMAT** statement number **k** identifies a format to be used by the print statement. The statement number can be any positive **INTEGER** constant up to five digits. Recall that statement numbers are placed in columns 1 through 5. The *expression list* specifies the value(s) to be printed. The general form of the **FORMAT** statement is

```
K     FORMAT(specification list)
```

A **FORMAT** statement is a non-executable statement. It can appear anywhere in the program before or after the associated print statement. The *specification list* in the **FORMAT** statement specifies both the vertical spacing and the horizontal spacing to be used when printing an output. The first character of the specification list, called the *carriage control character*, is used to control the vertical spacing. The rest of the specification list consists of various format specifications and controls the horizontal spacing.

FORTRAN provides format specifications for blank spaces, integer, real, character and logical types. Commas are used to separate specifications in the specification list. Before printing the line, the computer constructs each output line internally in a memory area called the *output buffer*. The length of each line in the buffer is 133 characters. The first character is used to control the vertical spacing and the remaining 132 characters represent the line to be printed. The buffer is filled with blanks before it is used to construct an output line.

The following are some of the carriage control characters used to control the vertical spacing:

- `' '`: single spacing (start printing at the next line)
- `'0'`: double spacing (skip one line then start printing)

- `'-'`: triple spacing (skip 2 lines then start printing)
- `'1'`: new page (move to the top of the next page before printing)
- `'+'`: no vertical spacing (start printing at the beginning of the current line irrespective of what was printed before)

The six format specifications presented below allow the control of horizontal spacing. In the following sections we will use

```
....+....1....+....2....+....3....+....4.
```

as a header to the output to indicate the horizontal spacing, Notes that the above line is not part of the output.

8.1.1 I Specification

The **I** specification is used to print integer expressions. The general form of I specification is **{Iw}**, where **w** is a positive integer representing the number of positions to be used to print the integer value. To find the minimum number of positions necessary to print a number, we count the number of digits in the integer including the minus sign. For example, if we want to print -25, the value of **w** should be at least 3. In the case where the value of **w** is more than 3, the number -25 is printed right-justified. If the value of **w** is less than 3, the number -25 cannot be printed and asterisk (*) characters appear in the output. In this case, the number of asterisks is equal to **w**.

In other words, to print an integer number using I specification, we start filling the positions from right to left. The extra positions to the left of the integer (if any) will be filled with blanks. If the positions are not enough to represent the number, the positions are filled with asterisks indicating that the specification is not enough to print the integer number.

Example 1: What is the minimum I specification needed to print each of the following integers?

345, 67, -57, 1000, 123456

Solution:

Number	I specification
345	I3
67	I2
-57	I3
1000	I4
123456	I6

Example 2: *What will be printed by the following program?*

```
      INTEGER M
      M = -356
      PRINT 10, M
10    FORMAT(' ',  I4)
      END
```

Solution:

```
....+....1....+....2
-356
```

Notice that the carriage control character ' ' did not appear in the output. This characters indicates that the output line is single spacing.

Example 3: *If the **FORMAT** statement in the previous example is modified as follows:*

```
      FORMAT('1', I6)
```

What will be printed?

Solution:

The printed output in this case will start on a *new page*, because of the carriage control character '1':

(new page)

```
....+....1....+....2....+....3....+....4.
-356
```

Example 4: *If the **FORMAT** statement in the previous example is modified as follows:*

```
      FORMAT('-', I3)
```

What will be printed?

Solution:

```
....+....1....+....2....+....3....+....4.


***
```

Notice that the printed output in this case has two empty lines before the data. The reason is the carriage control character '-' which means triple spacing. Moreover, the data is printed as three asterisks because the format specification I3 is not enough for the number -356.

Example 5: *Assume K = -244 and M = 12. The following **PRINT** statements will produce the shown outputs.*

```
a.    PRINT 10,  K
10    FORMAT(' ', I4)
```

```
....+....1....+....2....+....3....+....4.
-244
```

```
b.    PRINT 20,  K,  M
20    FORMAT(' ', I5, I6)
```

```
....+....1....+....2....+....3....+....4.
 -244     12
```

```
c.    PRINT 30,  K
      PRINT 35,  M
30    FORMAT(' ', I3)
35    FORMAT('0', I2)
```

```
....+....1....+....2....+....3....+....4.
***

12
```

```
d.    PRINT 40,  K + M
40    FORMAT(' ', I5)
```

```
....+....1....+....2....+....3....+....4.
 -232
```

```
e.    PRINT 50,  K / M
50    FORMAT(' ', I3)
```

```
....+....1....+....2....+....3....+....4.
-20
```

```
f.    PRINT 60,  M + 1.0
60    FORMAT(' ', I3)
```

```
ERROR MESSAGE: TYPE MISMATCH
```

```
g.    PRINT 70,  -345
70    FORMAT(' ', I7)
```

```
....+....1....+....2....+....3....+....4.
   -345
```

```
h.    PRINT 80,  -39 / 3 * 2
80    FORMAT(' ', I3)
```

```
....+....1....+....2....+....3....+....4.
-26
```

```
i.    PRINT 90, K
      PRINT 95, M
90    FORMAT(' ', I4)
95    FORMAT('+', I8)
```

```
....+....1....+....2....+....3....+....4.
-244  12
```

```
j.    PRINT 98, K
      PRINT 98, M
98    FORMAT(' ', I4)
```

```
....+....1....+....2....+....3....+....4.
-244
  12
```

8.1.2 F Specification

The **F** specification is used to print real values. The general form of the **F** specification is **{Fw.d}**, where **w** is a positive integer representing the total number of positions to be used to print the real number and **d** represents the number of positions to be used to print the fractional part of the real number. Note that **w** must satisfy the relation $w \geq d + 1$.

To find the number of positions needed to print a real number, we count the number of significant digits in the real number including the decimal point and the minus sign. For example, if we want to print -91.35, we need a total of six positions, two of them to the right of the decimal point, so the specification should be at least F6.2. To print the real number, we count from right to left **d** positions and place the decimal point at position **d+1**. We start placing the integer part of the real number from right to left and the fractional part of the real number from left to right. The extra positions to the left of the decimal point (if any) are filled with blanks, while the extra positions to the right of the decimal point (if any) are filled with zeros. If the number of positions to the left of the decimal point is not enough to represent the integer part of the real number, all **w** positions are filled with asterisks. If the number of positions to the right of the decimal

point is not enough to represent the fractional part of the real number, the number will be rounded to just fill the specified number of decimal positions.

Example 1: *What is the minimum F specification needed to print the following real numbers?:*

823.67509, 0.002, .05, -.05, -0.0008

Solution:

Number	F specification
823.67509	F9.5
0.002	F5.3
.05	F3.2
-.05	F4.2
98.	F3.0
98.0	F4.1
-0.0008	F7.4

Example 2: *What will be printed by the following program?*

```
      REAL X
      X = 31.286
      PRINT 10, X
10    FORMAT('1',  F6.3)
      END
```

Solution:

The printed output on *a new page* is as follows:

```
....+....1....+....2....+....3....+....4.
31.286
```

Example 3: *If the* ***FORMAT*** *statement in the previous example is modified as follows:*

```
      FORMAT(' ', F8.3)
```

What will be printed?

Solution:

```
....+....1....+....2....+....3....+....4.
  31.286
```

Example 4: *If the* ***FORMAT*** *statement in the previous example is modified as follows:*

```
      FORMAT(' ', F8.4)
```

What will be printed?

Solution:

```
....+....1....+....2....+....3....+....4.
 31.2860
```

Example 5: *If the* ***FORMAT*** *statement in the previous example is modified as follows:*

```
      FORMAT('   ', F5.3)
```

What will be printed?

Solution:

```
....+....1....+....2....+....3....+....4.
*****
```

Example 6: *If the **FORMAT** statement in the previous example is modified as follows:*

```
      FORMAT('   ', F6.2)
```

What will be printed?

Solution:

```
....+....1....+....2....+....3....+....4.
31.29
```

Example 7: *Assume X = -366.126, Y = 6.0 and Z = 20.97. The following **PRINT** statements will produce the shown outputs.*

```
a.    PRINT 10, X
10    FORMAT(' ', F11.5)
```

```
....+....1....+....2....+....3....+....4.
 -366.12600
```

```
b.    PRINT 20, X
20    FORMAT(' ', F8.3)
```

```
....+....1....+....2....+....3....+....4.
-366.126
```

```
c.    PRINT 30, Z
      PRINT 35, Y
30    FORMAT(' ', F4.1)
35    FORMAT('0', F4.2)
```

```
....+....1....+....2....+....3....+....4.
21.0

6.00
```

```
d.    PRINT 40,  X / Y
40    FORMAT(' ', F7.3)
```

```
....+....1....+....2....+....3....+....4.
-61.210
```

```
e.    PRINT 50,  Y + 0.00001
50    FORMAT(' ', F7.5)
```

```
....+....1....+....2....+....3....+....4.
6.00001
```

```
f.    PRINT 60,  Z - 5
60    FORMAT(' ', F5.2)
```

```
....+....1....+....2....+....3....+....4.
15.97
```

```
g.    PRINT 70,  Z
70    FORMAT('+', I5)
```

```
ERROR MESSAGE: TYPE MISMATCH
```

```
h.    PRINT 80,  -144 / 24 + 35.2
80    FORMAT(' ', F4.1)
```

```
....+....1....+....2....+....3....+....4.
```

```
29.2
```

```
i.     PRINT 85,  Y
       PRINT 85,  Z
85     FORMAT(' ', F6.2)
```

```
....+....1....+....2....+....3....+....4.
  6.00
 20.97
```

```
j.     PRINT 90,  Y
       PRINT 95,  Z
90     FORMAT(' ', F6.2)
95     FORMAT('-', F6.2)
```

```
....+....1....+....2....+....3....+....4.
  6.00


 20.97
```

8.1.3 X Specification

The **X** specification is used to insert blanks between the values we intend to print. The general form of this specification is **nX**, where **n** is a positive integer representing the number of blanks.

Example 1: *The following program:*

```
       REAL A, B
       A = -3.62
       B = 12.5
       PRINT 5, A,  B
5      FORMAT(' ',  F5.2, F4.1)
       END
```

prints the following output:

```
....+....1....+....2....+....3....+....4.
-3.6212.5
```

The output is not readable because the two printed values are not separated by blanks. If we modify the format statement using **X** specification as follows:

```
       FORMAT(' ', F5.2, 3X, F4.1)
```

the output becomes:

```
....+....1....+....2....+....3....+....4.
-3.62 12.5
```

The **X** specification can be used as a carriage control character. The following pairs of **FORMAT** statements print the same output.

```
10     FORMAT(' ', I2)
```

is equivalent to

```
10     FORMAT(1X, I2)
```

and

```
20     FORMAT(' ', 2X, F4.1)
```

is equivalent to

```
20    FORMAT(3X, F4.1)
```

8.1.4 Literal Specification

The literal specification is used to place character strings in a **FORMAT** statement as part of the specification list. The character string must be enclosed between two single quotation marks.

Example 1: *What will be printed by the following program?*

```
      REAL AVG
      AVG = 65.2
      PRINT 5, AVG
5     FORMAT(' ' ,'THE AVERAGE IS = ', F4.1)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
THE AVERAGE IS = 65.2
```

Example 2: *The following program prints the message FORTRAN77 on top of a new page.*

```
      PRINT 30
30    FORMAT('1', 'FORTRAN77')
      END
```

The output printed at the *a new page* is:

```
....+....1....+....2....+....3....+....4.
FORTRAN77
```

8.1.5 A Specification

The **A** specification is used to print character expressions. The general form of the **A** specification is **Aw**, where **w** represents the length of the character string. If the string has more than **w** characters, only the left-most **w** characters will appear in the output line. On the other hand, if the string has fewer than **w** characters, its characters are right-justified in the output line with blanks to the left. The integer **w** may be omitted. If **w** is omitted, the number of characters is determined by the length of the character string.

Example 1: *What will be printed by the following program?*

```
      PRINT 55, 'ICS-101'
55    FORMAT(' ',A7)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
ICS-101
```

Example 2: *What will be printed by the following program?*

```
      CHARACTER TEXT*5
      TEXT = 'KFUPM'
      PRINT 55, TEXT, TEXT, TEXT
55    FORMAT(' ', A, 3X, A3, 3X, A9)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
KFUPM    KFU        KFUPM
```

8.1.6 L Specification

The **L** specification is used to print logical expressions. The general form of **L** specification is **Lw**. The letter T or F is printed if the logical expression is true or false respectively. The printed letter is right-justified.

Example 1: *What will be printed by the following program?*

```
      PRINT 5, .TRUE.
5     FORMAT(' ',L1)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
T
```

Example 2: *What will be printed by the following program?*

```
      LOGICAL X,  Y
      X = .TRUE.
      Y = .FALSE.
      PRINT 15, X,  X
15    FORMAT(' ', L1, 2X,  L5)
      PRINT 20, Y,  Y
20    FORMAT(' ', L1, 2X,  L7)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
T       T
F         F
```

8.2 Specification Repetition: Another Format Feature

If we have consecutive identical specifications, we can replace them by an integer constant followed by the identical specification(s) to indicate repetition. For example, the specifications: I4, I4, I4 can be replaced by 3I4. Also, the specifications: I2, 3X, I2, 3X, I2, 3X, I2, 3X can be replaced by 4(I2, 3X). The following pairs of **FORMAT** statements illustrate the use of repetition constants:

```
10    FORMAT('0', 3X, I2, 3X, I2)
```

is equivalent to

```
10    FORMAT('0', 2(3X, I2))
```

and

```
20    FORMAT(' ',F5.1, F5.1, F5.1, 5X, I3, 5X, I3, 5X, I3, 5X, I3)
```

is equivalent to

```
20    FORMAT(' ',3F5.1,4(5X, I3))
```

8.3 Carriage Control Specification

The carriage control character is normally specified as the first character in the format specification list. It can be specified as a blank or the characters 0,1,-, +. But in the case where it is not specified as part of the specification list, the first character in the buffer output is taken as the carriage control character. If the first character of the buffer output is one of the carriage control characters (a blank, 0, 1, +, -), then the proper action is taken. If the first character is not among the carriage control characters, then the output is system dependent. The following example illustrates a specification list where carriage control character is missing:

Example:

```
      PRINT 10
10    FORMAT('1995')
      END
```

The output, on *a new page*, would be as follows:

```
....+....1....+....2....+....3....+....4.
995
```

Notice that the first character '1' was considered as a new page carriage control character.

8.4 File Processing

In many applications, the amount of data read and/ or produced is huge. Providing data interactively is not efficient, thus a different way to handle data is needed, namely, files. Another reason for using files comes from the repetitive use of the same data every time the program is run; making the data entry task very tedious. The third reason is that data in many real applications is taken or recorded by instruments or devices then used for analysis and computations.

8.4.1 Opening Files

Before using a file for input or output, it must be prepared for that operation. Files that are used for input must exist prior to their usage. To prepare a file for input, the following **OPEN** statement must precede any read statement from that file:

```
      OPEN(UNIT = INTEGER EXPR, FILE = FILENAME, STATUS = 'OLD')
```

where UNIT equals an integer expression in the range of 0 to 99. Avoid using 5 and 6 as unit numbers since they are already assigned for the keyboard and the screen. The filename is a character string containing the actual name of the file followed by the file extension. In the IBM mainframe, the file name is separated from the file extension by a space and if the extension is omitted, it is assumed to be FILE. Upon opening a file for reading, the reading will take place from the beginning of the file.

Files that are used for output may not exist before being used. If the file does not exist, it will be created whereas if it exists its contents will be erased. To prepare a file for output, the following statement must precede any write statement to that file:

```
OPEN(UNIT = INTEGER EXPR, FILE = FILENAME, STATUS = 'NEW')
```

or

```
OPEN(UNIT = INTEGER EXPR, FILE = FILENAME, STATUS ='UNKNOWN')
```

The second statement is preferred in our system because the first one assumes that the file does not exist and, therefore, if it exists an error occurs.

Example 1: *Assume that you want to use file POINTS DATA as an input file. The following statement will then appear before any read statement from the file:*

```
OPEN(UNIT = 1, FILE = 'POINTS DATA', STATUS = 'OLD')
```

Example 2: *Assume that you want to use file RESULT DATA as an output file. The following statement will then appear before any write statement to the file:*

```
OPEN(UNIT = 1, FILE = 'RESULT DATA', STATUS = 'UNKNOWN')
```

8.4.2 Reading from Files

To read from a file, the file must have been opened. The **READ** statement will be in the following form:

```
READ(UNIT, *) VARIABLE LIST
```

where UNIT is the same value that is used in the open statement. The rules of reading are exactly the same as the ones you have already seen, the only difference being that data is taken from the file.

Example 1: *Find the sum of three exam grades taken from file EXAM DATA.*

Solution:

```
INTEGER EXAM1, EXAM2, EXAM3, SUM
OPEN(UNIT = 10, FILE = 'EXAM DATA', STATUS = 'OLD')
READ(10, *) EXAM1, EXAM2, EXAM3
SUM = EXAM1 + EXAM2 + EXAM3
PRINT*, SUM
END
```

In many cases, the number of data values in a file is not known and we would like to do some calculations on the data values the file contains. For these cases, the read statement will look as follows:

```
READ(UNIT, *, END = NUMBER) VARIABLE LIST
```

where number is the label of the statement where control will be transferred after all the data from the file is read.

Example 2: *Find the average of real numbers that are stored in file NUMS DATA. Assume that we do not know how many values are in the file and that every value is stored on a separate line.*

Solution:

```
      REAL NUM, SUM, AVG
      INTEGER COUNT
      OPEN(UNIT = 12, FILE = 'NUMS DATA', STATUS = 'OLD')
      SUM = 0.0
      COUNT = 0
333   READ(12, *, END = 999) NUM
          SUM = SUM + NUM
          COUNT = COUNT + 1
          GOTO 333
999   AVG = SUM / COUNT
      PRINT*, AVG
      END
```

8.4.3 Writing to Files

To write to a file, the file must have been opened using an **OPEN** statement and the **WRITE** statement must be used in the following form:

```
      WRITE(UNIT, *) EXPRESSION LIST
```

where **UNIT** is the same value that is used in the **OPEN** statement. The rules of writing to a file are exactly the same as those of the print statement. The * in the **WRITE** statement indicates that the output is free formatted. If format is needed, the format statement number is used instead.

Example: *Create an output file CUBES DATA that contains the table of the cubes of integers from 1 to 20 inclusive.*

Solution:

```
      INTEGER NUM
      OPEN(UNIT = 20, FILE = 'CUBES DATA', STATUS = 'UNKNOWN')
      DO 22 NUM = 1, 20
          WRITE(20, *) NUM, NUM**3
22    CONTINUE
      END
```

Format statement could be used with the write statement in the same way it is used with the print statement. The * in the write statement is replaced with the format statement number.

8.4.4 Working with Multiple Files

In any program, more than one file may be open at the same time for either reading or writing. The same unit number that is used in one file should not be used with any other file in the same program. The number of the files that can be open at the same time is limited by the number of units, which is dependent on the computer you are using.

Example: *Create an output file THIRD that contains the values in file FIRST followed by the values in file SECOND. Assume that every line contains one integer number and we do not know how many values are stored in files FIRST and SECOND.*

Solution:

```
      INTEGER NUM
      OPEN(UNIT = 15, FILE = 'FIRST', STATUS = 'OLD')
      OPEN(UNIT = 17, FILE = 'SECOND', STATUS = 'OLD')
      OPEN(UNIT = 19, FILE = 'THIRD', STATUS = 'UNKNOWN')
123   READ(15, *, END = 456) NUM
          WRITE(19, *) NUM
          GOTO 123
456   READ(17, *, END = 789) NUM
          WRITE(19, *) NUM
          GOTO 456
789   STOP
      END
```

8.4.5 Closing Files

After using a file in our program, that file must be closed. The operating system of the computer we are using normally closes all the files that are open at the end of the program execution. But in some cases, we may need to read the data in the file more than one time. This can be done by closing the file after we finish reading from it and then re-opening the file to read the same data again. We may also need to read from files that were created by our program. This is achieved by closing the file as an output file then re-opening it as an input file. The **CLOSE** statement looks as follows:

```
      CLOSE(UNIT)
```

where unit is the same value that is used in the open statement. You can only close files that are already open.

8.4.6 Rewinding Files

After reading from the file the reading head moves forward towards the end of the file. In certain situations, we may need to restart reading from the beginning of the file which is done by closing the file then re-opening it again. Another method of doing the same thing is through the **REWIND** statement:

```
      REWIND(UNIT)
```

where unit is the same value that is used in the open statement. You can rewind files that are open for reading only.

8.5 Exercises

8.5.1 Exercises on Output Design

1. What will be printed by each of the following programs?

```
1.    REAL X
      X = 123.8367
      PRINT 10, X,  X,  X
10    FORMAT(' ', F7.2, 2X, F6.2, F9.5)
      END
```

```
2.    INTEGER J, K, N
      K = 123
      J = 456
      N = 789
      PRINT 10, K
      PRINT 11, J
      PRINT 12, N
10    FORMAT(' ', I3)
11    FORMAT('+', 3X, I3)
12    FORMAT('+', 6X, I3)
      END
```

```
3.    REAL X1,  X2
      INTEGER N1,  N2
      READ*, X1,  X2
      READ*, N1,  N2
      PRINT 10, X1,  X2
      PRINT 11, N1,  N2
      PRINT 12, X1/X2
10    FORMAT('1',F5.2,  2X,  F3.1)
11    FORMAT('0', I3,  2X,  I2)
12    FORMAT('+', 12X,  F6.2)
      END
```

Assume the input for the above program is:

```
81.6   9.2
-125   48
```

```
4.    PRINT 20, -35,  0.0,  12 * 10.0,  125 / 5
20    FORMAT(1X, I3, '+', F3.1, 'IS NOT EQUAL', F6.1,'-',I2)
      END
```

```
5.    LOGICAL FLAG,  P,  Q
      READ*, P,  Q
      FLAG = .NOT. P .AND. .NOT. Q
      PRINT 33, P,  'AND', Q
      PRINT 44, P .OR. Q,  FLAG
33    FORMAT(' ', L2, 2X,  A,  L3)
44    FORMAT('-', L1, 2X,  L1)
      END
```

Assume the input for the above program is:

```
T   F
```

```
6.    REAL X, Y
      INTEGER N
      X = 25.0
      Y = -35.0
      N = -35
      PRINT 40, X,  SQRT(X)
      PRINT 50, Y,  ABS(Y)
      PRINT 60, N,  ABS(N)
40    FORMAT(' ', 'X=', 2X, F4.1, 2X, 'SQUARE ROOT = ', F4.1)
50    FORMAT(' ', 'Y=', 2X, F5.1, 2X, 'ABSOLUTE VALUE = ',F5.1)
60    FORMAT(' ', 'N=', 2X, I3, 2X, 'ABSOLUTE VALUE = ', I2)
      END
```

```
7.    CHARACTER*6 CITY
      CITY = 'RIYADH'
      PRINT 1, 'THE CAPITAL IS', 2X, CITY
1     FORMAT(' ', A,  2X,  A4)
      END
```

```
8.      INTEGER ARR(5), K
        READ*, ( ARR(K), K = 1,  5)
        DO 70 K = 1,  5
        PRINT 10, ARR(K)
70      CONTINUE
10      FORMAT(' ', I4)
        END
```

Assume the input for the above program is:

```
10   20   30   40   50
```

```
9.      INTEGER ARR(5), K
        READ*, ( ARR(K), K = 1,  5)
        PRINT 10, ( ARR(K), K = 1,  5)
10      FORMAT(' ', 5I2)
        END
```

Assume the input for the program is:

```
10   20   30   40   50
```

```
10.     INTEGER ARR(5), K
        READ*, ( ARR(K), K = 1,  5)
        PRINT 10, ( ARR(K), K = 1,  5)
10      FORMAT(' ', 5(I2,2X))
        END
```

Assume the input for the program is:

```
10   20   30   40   50
```

```
11.     REAL MAT(2,3), I, J
        READ*,(( MAT(I, J), I=1,2),J=1,3)
        DO 10 I= 1,  2
            PRINT 55, (MAT(I, J), J=1,3)
10      CONTINUE
55      FORMAT(' ', 3( F4.1, 2X))
        END
```

Assume the input for the program is:

```
10   20   30   40   50   60
```

```
12.     REAL A(30), B(30), DOT, Z
        INTEGER K, N
        READ*, N,  (A(K), B(K), K=1,  N)
        Z = DOT(N, A, B)
        PRINT 10, Z
10      FORMAT('1', 'DOT PRODUCT = ', F5.1)
        END
        REAL FUNCTION DOT(M, X, Y)
        INTEGER M, I
        REAL X(M),Y(M), SUM
        SUM = 0.0
        DO 123 I = 1,  M
            SUM = SUM + X(I)* Y(I)
123     CONTINUE
        DOT = SUM
        RETURN
        END
```

Assume the input for the program is:

```
4   1   2   3   4   5   6   7   8
```

```
13.    INTEGER N1, N2
       REAL S1, S2
       READ*, N1, N2
       READ*, S1
       READ*, S2
       READ*, N1
1      FORMAT('0', I4, '+', I2, 2X, '=', I4)
2      FORMAT(' ', A, 3X, F5.2)
3      FORMAT('+', 7X, F10.2)
       PRINT 1, N1, N2, N1+N2
       PRINT 2, 'S1', S1
       PRINT 3, S2
       END
```

Assume the input for the program is:

```
37
101   4113   25.0
-30.459   210.0
427.5   48
23
```

2. Indicate the validity of the following statements:
   1. The **FORMAT** statement can be placed anywhere between the declaration statements and the **END** statement of a FORTRAN77 program.
   2. Two or more **PRINT** statements can refer to the same format statement. For example, if X and Y are real variables then the following program segment:

```
       PRINT 5, X
       PRINT 5, Y
5      FORMAT(4X, F5.2)
```

   is correct.

3. Complete the following programs in order to get the required outputs:

```
1.     REAL X
       X = 5.98
       PRINT 1, X
       PRINT 2, X
1      FORMAT(                          )
2      FORMAT(                          )
       END
```

The required output is:

```
....+....1....+....2....+....3....+....4.
    X=5.980   X=6.0
```

```
2.     INTEGER B
       REAL A, C
       A = 3.1
       B = 12.5
       C = 127.66
       PRINT 1520, A, B, C
1520   FORMAT(                              )
       END
```

The required output is:

```
....+....1....+....2....+....3....+....4.
  3.10  12 127.7
```

```
3.    REAL A,
      INTEGER J
      A = -5.62705
      J = 23
      PRINT 5, A, J
5     FORMAT(                          )
      END
```

The required output is:

```
....+....1....+....2....+....3....+....4.
        -5.63     23
```

```
4.    INTEGER Z
      REAL X, Y
      X = 5.00
      Y = 59.996
      Z = 3125
      PRINT 5, X,  Y, Z
5     FORMAT(                                  )
      END
```

The required output is:

```
....+....1....+....2....+....3....+....4.
X= 5.00 Y=  60.00  Z= ***
```

```
5.    PRINT 1, 'FORTRAN'
      PRINT 2, 'I LIKE'
1     FORMAT(                                 )
2     FORMAT(                                 )
      END
```

```
THE REQUIRED OUTPUT IS:
....+....1....+....2....+....3....+....4.
 I LIKE FORTRAN
```

```
6.    INTEGER Y
      REAL X
      X = -20.2451
      Y = 25
      PRINT 6, X, 'AND', Y
6     FORMAT(                                )
      END
```

The required output is:

```
....+....1....+....2....+....3....+....4.
    -20.25   AND   25
```

4. Write a program segment to print the heading "FORTRAN-77--LANGUAGE" centered at the top of a new page. assume the output line contains 80 characters.

5. Write a program that reads any real number, separates the integer and real parts of the number and prints it in the format shown below. For example, if the input is as follows:

```
123.45
```

your formatted output should be as follows:

```
....+....1....+....2....+....3....+....4.
123.450=123+0.450
```

6. Consider the following program

```
      INTEGER X
      REAL Y
      X = 469
      Y = 17.38
      PRINT2, X, Y
2     FORMAT(                    )
      END
```

Given the following format statements below:

```
a.    2   FORMAT(5X, I3, 2X, F4.1)
```

```
b.    2   FORMAT(6X, I3, 2X, F4.1)
```

```
c.    2   FORMAT(1X, I8, F6.1)
```

Which of the above **FORMAT** statements can be used in place of the **FORMAT** statement in the program to print the output as follows?

```
....+....1....+....2....+....3....+....4.
     469  17.4
```

7. The output of the program given below is as follows

```
....+....1....+....2....+....3....+....4.
 TEST = -3.527  M=***

M = 2531  TEST = -3.5270
 M =    -3.53  M=2531
```

Place the proper **FORMAT** statement numbers with the **PRINT** statements such that the output is as given above.

```
      REAL TEST
      INTEGER M
      TEST = -3.527
      M = 2531
      PRINT   A   , TEST, M
      PRINT   B   , M, TEST
      PRINT   C   , TEST, M
10    FORMAT(2X, 'TEST = ',F6.3, 2X, 'M=', I3)
20    FORMAT(2X, 'M = ',F8.2, 2X, 'M=', I4)
30    FORMAT('0','M =',I5, 2X, 'TEST = ', F7.4)
      END
```

8.5.2 Exercises on FILES

1. Consider the following statement:

```
      READ(8, *, END = 10) A
```

Which of the following statements is (are) correct about the above statement?

1. The value of A will be read from the area after Assume the input for the program is:.

2. At the end of the file, this read statement will transfer control to statement labeled 10.

3. The value of A will be read from the file linked to unit 8.

2. Which of the following statements is/are FALSE about files:

1. The statement that assigns unit number 9 to the input file "DATA" is:

```
OPEN (UNIT = 9, FILE = 'DATA', STATUS = 'OLD')
```

2. The **OPEN** statement for a data file must precede any **READ** or **WRITE** statements that uses that file.
3. A statement that reads two numbers from a file may look like:

```
READ ( 9, *, END = 31) K, L
```

4. The **OPEN** statement for a file should be executed only once in the program.
5. A statement that writes two numbers into a file may look like:

```
PRINT(9, *) K, L
```

6. A file is a collection of data records.
7. A file is usually used only once.
8. A file can be opened at the same time with two different unit numbers.
9. Two files with the same unit number can not be opened at the same time.
10. We store data in files when we do not need them any more.

3. What will be printed by the following programs?

```
1.    INTEGER M, K
      OPEN ( UNIT = 10, FILE = 'INPUT DATA', STATUS = 'OLD')
      READ ( 10, *, END = 10) ( M, K = 1,100)
10    PRINT*, M, K-1
      END
```

Assume that the file 'INPUT DATA' contains the following:

```
1 2 3
4 5
6 7 8 9
6
```

```
2.    INTEGER J, K
      OPEN ( UNIT = 3, FILE = 'FF1', STATUS = 'OLD')
      DO 50 J=1,100
      READ ( 3,*,END = 60) K
50    CONTINUE
60    PRINT*,'THE VALUES ARE:'
      PRINT*,K,J
      END
```

The contents of the file 'FF1' are:

```
20 50 67 45 18 -2 -20
88 66 77 105 55 300
```

```
3.    INTEGER M
      OPEN ( UNIT = 10, FILE = 'INPUT',STATUS = 'OLD')
      READ (10,*) M
20    IF ( M.NE.-1) THEN
          PRINT*,M
          READ(10, *, END = 30) M
          GOTO 20
      ENDIF
      PRINT*, 'DONE'
30    PRINT*, 'FINISHED'
      END
```

Assume that the file 'INPUT' contains the following :

```
7
```

```
3
9
4
-1
```

```
4.    INTEGER N, K
      OPEN ( UNIT = 12, FILE = 'INFILE', STATUS = 'OLD')
      READ*,N
      DO 10 K=1,N
      PRINT*, N
      READ(12,*,END = 15) N
10    CONTINUE
      PRINT*,N
15    CONTINUE
      END
```

Assume the input for the program is:

```
4
```

Given that the file 'INFILE' contains the following data

```
2
3
```

```
5     INTEGER A, B
      OPEN ( UNIT = 10, FILE = 'INPUT DATA', STATUS = 'OLD')
      OPEN ( UNIT = 11, FILE = 'OUTPUT DATA', STATUS = 'NEW')
      READ*,A,B
      READ(10,*) A,B,A
      WRITE(11,*) A, B
      READ(10, *, END = 10) A, B
10    WRITE(11,*) A, B
      END
```

Assume the input for the program is:

```
10 11
```

Assume that the file 'INPUT DATA' contains the following data

```
4 5
6 7
8
```

What will be written in the file 'OUTPUT DATA' file ?

```
6.    INTEGER S, T, U
      OPEN ( UNIT = 10, FILE = 'INPUT',STATUS = 'OLD')
10    READ(10, *, END = 30) S, T
      U = S
      T = U
      U = S
      IF ( S.NE.T) THEN
          U = 1
      ELSE
          U = 0
      ENDIF
      GOTO 10
30    PRINT*, U, S, T
      END
```

Assume the file 'INPUT' contains the following data:

```
3
4
5
6
7
```

```
8
```

```
7.    INTEGER X(6), M, K
      OPEN ( UNIT = 10, FILE = 'INPUT1', STATUS = 'OLD')
      OPEN ( UNIT = 11, FILE = 'INPUT2', STATUS = 'OLD')
      M = 0
10    M = M + 1
      READ(10,*) X(M)
      IF ( X(M).GT.0) GOTO 10
20    M = M + 1
      READ(11,*) X(M)
      IF ( X(M).GT.0) GOTO 20
      PRINT 1, (X(K),K=1,M)
1     FORMAT(' ',I2,I2,I2,I2,I2,I2)
      END
```

Assume you have two files 'INPUT1' and 'INPUT2' with the following data:

INPUT1	INPUT2
3	6
8	0
0	7
5	0

```
8.    INTEGER N, K
      OPEN(UNIT=22, FILE = 'INPUT', STATUS = 'OLD')
33    READ (22,*) N
      IF (N.EQ.0) GOTO 44
      PRINT*, ('*', K=1,N)
      GOTO 33
44    PRINT*, 'HISTOGRAM'
      END
```

Given that the file 'INPUT' contains the following data

```
5
2
4
0
```

4. A set of three real numbers are read from the file TEST and the number associated to the file is 10. The output is then written to a new file called REST and the number associated to the file is 12. Write a FORTRAN 77 program to do the above operations.
5. Write a FORTRAN 77 program to copy an old file "TEST1" to a new "TEST2". It is assumed that each line of "TEST1" contains a student ID and his garde out of 100. The number of data lines in the old file is not known.
6. Write a FORTRAN 77 program which will read values from a data file, the file name is: INPUT and its type is DATA.
    1. Open the INPUT file.
    2. Open a new output file called: ODD DATA.
    3. open a new output file called: EVEN DATA. It is not known exactly how many data there is in the INPUT file.
    4. Use the read (... END =..) to read the values from the file one by one and
    5. If the value is odd, write it in the file: ODD DATA.
    6. If the value is even, write it in the file: EVEN DATA.

7. A file called INPUT is assumed to contain an unknown number of lines, however, we know that every line contains exactly two numbers. Write a program that reads each line from file INPUT and prints the smaller of the two numbers in a file called SMALL and the larger in a file called BIG.
8. The following incomplete program was written to compare two files 'INFOR1' and 'INFOR2'. If the data in the files is the same then the program prints the message 'SAME FILES'. Otherwise the program prints 'DIFFERENT FILES'. Each line in both files contain two integer numbers followed by one logical value. Assume both files have the same number of records. Complete the program:

```
      INTEGER X1, X2, X3, X4
      LOGICAL    (1)   ,    (2)   ,FLAG
      OPEN ( UNIT = 1, FILE = 'INFOR1', STATUS = 'OLD')
                         (3)
      FLAG =    (4)
10    READ (1,*,END =    (5)   ) X1, X2, VAL1
      READ (2,*) X3, X4, VAL2
      IF ( X1.EQ.X3 .AND.          (6)           ) THEN
         GOTO 10
      ELSE
         FLAG = .FALSE.
      ENDIF
20    IF ( FLAG) THEN
         PRINT*,     (7)
      ELSE
         PRINT*,     (8)
      ENDIF
      END
```

8.6 Solutions to Exercises

8.6.1 Solutions to Exercises on Output Design

Ans 1.

1.

```
....+....1....+....2....+....3....+....4.
 123.84  123.84123.83670
```

2.

```
....+....1....+....2....+....3....+....4.
123456789
```

3.

(new page)

```
....+....1....+....2....+....3....+....4.
81.60  9.2

***  48       8.87
```

4.

```
....+....1....+....2....+....3....+....4.
 -35+0.0IS NOT EQUAL 120.0-25
```

5.

```
....+....1....+....2....+....3....+....4.
  T  AND  F


T  F
```

6.

```
....+....1....+....2....+....3....+....4.
X=  25.0  SQUARE ROOT =  5.0
Y=  -35.0  ABSOLUTE VALUE = 35.0
N=  -35  ABSOLUTE VALUE = 35
```

7.

```
....+....1....+....2....+....3....+....4.
THE CAPITAL IS  RIYA
```

8.

```
....+....1....+....2....+....3....+....4.
  10
  20
  30
  40
  50
```

9.

```
....+....1....+....2....+....3....+....4.
1020304050
```

10.

```
....+....1....+....2....+....3....+....4.
10  20  30  40  50
```

11.

```
....+....1....+....2....+....3....+....4.
10.0  30.0  50.0
20.0  40.0  60.0
```

12.

(new page)

```
....+....1....+....2....+....3....+....4.
DOT PRODUCT = 100.0
```

13.

```
....+....1....+....2....+....3....+....4.
23+**  = 124
S1   ***** 427.50
```

Ans 2.

1. VALID
2. VALID

Ans 3.

1.

```
1      FORMAT (5X, 'X=',F5.3)
2      FORMAT ('+', 14X, 'X=', F3.1)
```

2.

```
1520   FORMAT (3X, F4.2, 2X, I2, 1X, F5.1)
```

3.

```
5      FORMAT (' ', 9X, F5.2, 5X, I2)
```

4.

```
5      FORMAT (3X, 'X= ', F4.2,1X, 'Y= ',2X,F5.2, 2X,'Z= ', I3)
```

5

```
1      FORMAT (' ', 8X, A)
2      FORMAT ('+', 1X, A)
```

6.

```
6      FORMAT (' ', 4X, F6.2, 3X, A, 3X, I2)
```

Ans 4.

```
       PRINT 10
10     FORMAT('1', 30X, 'FORTRAN-77--LANGUAGE')
```

Ans 5.

```
       REAL X, RPART
       INTEGER IPART
       READ*, X
       IPART = X
       RPART = X - IPART
       PRINT 5,  X, IPART, RPART
5      FORMAT (' ', F7.3, '=', I3, '+', F5.3)
       END
```

Ans 6.

b or c

Ans 7.

(a) 10
(b) 30
(c) 20

8.6.2 Solutions to Exercises on Files

Ans 1.

2 3

Ans 2.

```
      4       5       7       8       10
```

Ans 3.

```
      6       10
      THE VALUES ARE:
   88  3
      7
   3
   9
   4
   DONE
   FINISHED
      4
   2
   3
      6 5
   8  5
      0  7  7
      3  8  0  6  0
      *****
   **
   ****
   HISTOGRAM
```

Ans 4.

```
      REAL RN1, RN2, RN3
      OPEN( UNIT = 10, FILE = 'TEST', STATUS = 'OLD' )
      OPEN( UNIT = 12, FILE = 'REST', STATUS = 'UNKNOWN' )
      READ(10, *) RN1, RN2, RN3
      WRITE(12, *) RN1, RN2, RN3
      END
```

Ans 5.

```
      INTEGER ID, GRD
      OPEN( UNIT = 1, FILE = 'TEST1', STATUS = 'OLD' )
      OPEN( UNIT = 2, FILE = 'TEST2', STATUS = 'UNKNOWN' )
5     READ(1, *, END = 10) ID, GRD
      WRITE(2, *) ID, GRD
      GOTO 5
10    PRINT*, 'DONE'
      END
```

Ans 6.

```
      INTEGER NUM
      OPEN( UNIT = 20, FILE = 'INPUT DATA', STATUS = 'OLD' )
      OPEN( UNIT = 30, FILE = 'ODD DATA', STATUS = 'UNKNOWN' )
      OPEN( UNIT = 40, FILE = 'EVEN DATA', STATUS = 'UNKNOWN' )
100   READ(20, *, END = 200) NUM
      IF ( MOD( NUM, 2) .EQ. 1 ) THEN
          WRITE(30, *) NUM
      ELSE
          WRITE(40, *) NUM
      ENDIF
      GOTO 100
200   PRINT*, 'DONE'
      END
```

Ans 7.

```
      INTEGER N1, N2
      OPEN( UNIT = 11, FILE = 'INPUT', STATUS = 'OLD' )
      OPEN( UNIT = 12, FILE = 'SMALL', STATUS = 'UNKNOWN' )
      OPEN( UNIT = 13, FILE = 'BIG', STATUS = 'UNKNOWN' )
20    READ(11, *, END = 25) N1, N2
      IF ( N1 .LT. N2 ) THEN
          WRITE(12, *) N1
          WRITE(13, *) N2
      ELSE
          WRITE(12, *) N2
          WRITE(13, *) N1
      ENDIF
      GOTO 20
25    PRINT*, 'DONE'
      END
```

Ans 8.

1. VAL1
2. VAL2
3. **OPEN**(UNIT = 2, FILE = 'INFOR2', STATUS = 'OLD')
4. TRUE.
5. 20
6. X2 .EQ. X4 .AND. VAL1 .EQV. VAL2
7. 'SAME FILES'
8. 'DIFFERENT FILES'

9 APPLICATION DEVELOPMENT: SORT & SEARCH

In this chapter, we introduce a number of applications developed in FORTRAN. The methodology we follow to develop these applications will be shown as we consider each application in detail.

Sorting and *Searching* are two applications discussed in this chapter. When sorting, we sort (order) elements of a list in either an increasing or a decreasing order. Searching, on the other hand, is the process of finding an element within a list.

9.1 Sorting

Sorting is the process of ordering the elements of any list either in increasing (or ascending) or decreasing (or descending) order. Here, we discuss a method for sorting a list of elements (values) into order, according to their arithmetic values. It is also possible to sort elements that have *character* values since each character has a certain arithmetic value for its representation. This will be discussed in details in Chapter 10.

Sorting in increasing order means that the smallest element in value should be first in the list. Then comes the next smallest element, followed by the next smallest and so on. Figure 1 shows three lists: unsorted (unordered) list, the list sorted in increasing order, and the same list sorted in decreasing order The exact reverse happens in sorting a list in decreasing order. In the literature, one can find a number of well established techniques for achieving this goal (sorting). Techniques such as *insertion sort, bubble sort, quick sort, selection sort*, etc. differ in their complexity and speed. In the following section, we introduce a simple sorting technique and its FORTRAN implementation.

Unsorted	Increasing order	Decreasing order
73	18	89
65	40	73
52	52	65
18	65	65
89	65	52
65	73	40
40	89	18

Figure 1: Unsorted and sorted lists

9.1.1 A Simple Sorting Technique

The idea of this sorting technique is to select the minimum (or the maximum depending on whether the sorting is in increasing or decreasing order) value within the list and assign it to be the first element of the list. Next, we take the remaining elements and select the minimum among them and assign it to be the second element. This process is repeated until the end of the list is reached. To select the minimum within a list of elements, one has to compare all the elements and keep the minimum value updated.

In the following subroutine, this sorting technique is implemented. Two loops are used in this procedure. The first moves through the elements of the array one after the other and stops at the element before the last element in the array. For each of these elements comparisons are conducted between that element and the rest of the array. So, the second loop moves over the rest of the array elements starting at the element next to the one being considered in the first loop. For example, if the first loop is at element number 3, the second loop would move over the elements from 4 to the last. Within the second loop, element 3 is compared with all the remaining elements starting from the fourth element to the last to make sure that element 3 is less than all of them. If element 5, for example, was found to be less than element 3, we swap the two elements. As we move ahead with the first loop, we are sure that the element we leave is the smallest among the elements that follow it. The FORTRAN subroutine that implements this sorting technique is as follows:

```
      SUBROUTINE SORT (A, N)
      INTEGER N, A(N), TEMP, K, L
      DO 11 K = 1, N - 1
          DO 22 L = K+1, N
              IF (A(K).GT.A(L)) THEN
                  TEMP = A(K)
                  A(K) = A(L)
                  A(L) = TEMP
              ENDIF
22        CONTINUE
11    CONTINUE
      RETURN
      END
```

Let us now run the above subroutine when the value of N is 5 and the array A consists of the following:

3	-2	4	9	0

After the first pass (the first iteration of the K-loop), the list becomes:

-2	3	4	9	0

After the second iteration of the K-loop, the list becomes:

-2	0	4	9	3

Notice that the 0, the smallest within the 4 remaining elements is the one swapped to the second position. After the third iteration of the K-loop, the list becomes:

-2	0	3	9	4

After the fourth iteration of the K-loop, the list becomes:

-2	0	3	4	9

9.2 Searching

As part of any system, information or data might need to be stored in some kind of data structure. One example is one-dimensional arrays. Assume that information about students in some university is stored. Assume again that the IDs of students registered in the current semester are stored in an array STUID. Suppose that an instructor asks the registrar to check whether a student, who has an 882345 as his ID, is registered this semester or not. For the registrar to conduct this check, he has to *search* within the array STUID for the student who has the ID 882345.

A number of search techniques are well known in computer science. These techniques locate a value within a set of values stored in some data structure. A simple searching technique, namely **sequential search**, is introduced in the next section.

9.2.1 Sequential Search

Sequential search starts at the beginning of a list (array) and looks at each element sequentially to see if it is the one being searched. This process continues until either the element is found or the list ends, that is all the elements in the list have been checked.

The FORTRAN function that implements this algorithm follows. The function SEARCH searches for the element K in the array A of size N. If the element is found, the index of the element is returned. Otherwise, a zero value is returned.

```
      INTEGER FUNCTION SEARCH(A, N, K)
      INTEGER N, A(N), K, J
      LOGICAL FOUND
      SEARCH = 0
      J = 1
      FOUND = .FALSE.
10    IF (.NOT. FOUND .AND. J .LE. N) THEN
          IF (A(J) .EQ. K) THEN
              FOUND = .TRUE.
              SEARCH = J
          ELSE
              J = J + 1
          ENDIF
          GOTO 10
      ENDIF
      RETURN
      END
```

When the element K is found, the function returns with the position of K. Otherwise, after all the elements have been checked, the function returns with the value zero.

9.3 An Application: Maintaining student grades

Question: Write a program that reads IDs *of students together with their* grades *in some exam. The number of students is read first. The input is given such that each line contains the ID of the student and his grade. Assume the following input :*

```
7
886767    94
878787    35
898982    82
867878    63
```

```
867676    55
898777    75
886788    22
```

After reading the IDs and the grades, the program must allow us to interactively do the following:

1. *SORT according to ID*
2. *SORT according to GRADES*
3. *CHANGE a GRADE*
4. *EXIT the program*

Solution:

We will first write a subroutine MENU that gives us the various options listed in the problem and also reads an option. The subroutine MENU is as follows :

```
      SUBROUTINE MENU (OPTION)
      INTEGER OPTION
      PRINT*, 'GRADES MAINTENANCE SYSTEM '
      PRINT*, ' 0. EXIT THIS PROGRAM'
      PRINT*, ' 1. SORT ACCORDING TO ID '
      PRINT*, ' 2. SORT ACCORDING TO GRADES '
      PRINT*, ' 3. CHANGE A GRADE '
      PRINT*, ' ENTER YOUR CHOICE :'
      READ*, OPTION
      RETURN
      END
```

We will now rewrite the subroutine SORT since we need to sort one array and also make the corresponding changes to another array. For example, if we are sorting the array of grades, the swapping of elements in this array must be reflected in the array of IDs as well. Otherwise, the grade of one student would correspond to the ID of another. After sorting, we will print the two arrays in the subroutine. The new subroutine TSORT is as follows:

```
      SUBROUTINE TSORT (A, B, N)
      INTEGER N, A(N), B(N), TEMP, J, K, L
      DO 11 K = 1, N - 1
          DO 22 L = K+1, N
              IF (A(K).GT.A(L)) THEN
                  TEMP = A(K)
                  A(K) = A(L)
                  A(L) = TEMP
                  TEMP = B(K)
                  B(K) = B(L)
                  B(L) = TEMP
              ENDIF
22        CONTINUE
11    CONTINUE
      PRINT*, 'SORTED DATA : '
      DO 33 J = 1, N
          PRINT*, A(J), B(J)
33    CONTINUE
      RETURN
      END
```

Note that we are sorting array A but making all the corresponding changes in array B. To this subroutine, we can pass the array of grades as array A and the array of IDs as array B. The subroutine then returns the array of grades sorted but at the same time

makes the corresponding changes to the array of IDs. If to this subroutine, we pass the array of IDs as array A and the array of grades as array B, the subroutine returns the array of IDs sorted but at the same time makes the corresponding changes to the array of grades.

To change a grade, we are given the ID of the student. We need to search the array of IDs for the given ID. We can use the function SEARCH we developed in Section 9.2. We can pass the array of IDs to the dummy array A and the ID to be searched to the dummy argument K. Note that the function SEARCH returns a zero if the ID being searched is not found.

Using the subroutines MENU and TSORT, and the function SEARCH, we develop the main program as follows :

```
      INTEGER GRADES(20), ID(20)
      INTEGER SEARCH, SID, NGRADE, OPTION, K, N
      PRINT*, 'ENTER NUMBER OF STUDENTS'
      READ*, N
      DO 10 K = 1, N
          PRINT*, 'ENTER ID AND GRADE OF STUDENT ', K
          READ*, ID(K), GRADES(K)
10    CONTINUE
      CALL MENU (OPTION)
15    IF (OPTION .NE. 0) THEN
          IF (OPTION .EQ. 1) THEN
              CALL TSORT(ID, GRADES, N)
          ELSEIF (OPTION .EQ. 2) THEN
              CALL TSORT(GRADES, ID, N)
          ELSEIF (OPTION .EQ. 3) THEN
              PRINT*, 'ENTER ID \& THE NEW GRADE'
              READ*, SID, NGRADE
              K = SEARCH(ID, N, SID)
              IF (K.NE.0) THEN
                  GRADES(K) = NGRADE
              ELSE
                  PRINT*, 'ID : ' ,SID, ' NOT FOUND'
              ENDIF
          ELSE
              PRINT*, 'INPUT ERROR '
          ENDIF
          CALL MENU (OPTION)
          GOTO 15
      ENDIF
      END
```

The main program first reads the two arrays ID and GRADES each of size N. Then it displays the menu and reads an option from the screen into the variable OPTION using subroutine MENU. If the input option is 1, the subroutine TSORT is called in order to sort IDs. If the input option is 2, the subroutine TSORT is called in order to sort the grades. If the input option is 3, the ID to be searched (SID) and the new grade (NGRADE) are read, and the function SEARCH is invoked. If the ID is found, the corresponding grade in array GRADES is changed. Otherwise, a message indicating that the SID is not found is printed. The main program runs until option 4 is chosen.

9.4 Exercises

1. Modify the application given in Section 9.3 as follows:

a. Add an option that will list the grade of a student given his ID.
b. Given a grade, list all IDs who scored more than the given grade.
c. Add an option to find the average of all the grades.
d. Add an option to find the maximum grade and the corresponding ID.
e. Add an option to find the minimum grade and the corresponding ID.
f. Add an option to list the IDs of all students above average.

2. The seating arrangement of a flight is stored in a data file FLIGHT containing six lines. each line contains three integers. a value of 1 represents a reserved seat, and a value of 0 represents an empty seat. the contents of flight are:

```
1    0    1
0    1    1
1    0    0
1    1    1
0    0    1
0    0    0
```

write an interactive program which has a menu with the following options:

0. Exit
1. Show number of empty seats
2. Show Empty seats
3. Reserve a seat
4. Cancel a seat

The program first reads from the data file FLIGHT and stores the data in a two-dimensional integer array seats of size 6 × 3 row-wise. then:

a. If option 1 is chosen, the main program passes the array seats to an integer function NEMPTY which returns the number of empty seats. Then the main program prints this number.
b. If option 2 is chosen, the main program passes the array seats to a subroutine ESEATS which returns the number of empty seats and the positions of all empty seats in a two-dimensional integer array EMPTY of size 18 × 2. Then, the main program prints the array EMPTY row-wise.
c. If option 3 is chosen, the user is prompted to enter the row number and the column number of the seat to be reserved. the main program then passes these two integers together with the array SEATS to a logical function RESERV which reserves a seat if it is empty and returns the value .true. to the main program. If the requested seat is already reserved or if the row or column number is out of range the function returns the value .false. to the main program. The main program then prints the message SEAT RESERVED or SEAT NOT AVAILABLE respectively.
d. If option 4 is chosen, the user is prompted to enter the row number and the column number of the seat to be canceled. the main program then passes these two integers together with the array SEATS to a logical function CANCEL which cancels a seat if it is reserved and returns the value .true. to the main program. if the requested seat is already empty or if the row or column number is out of range the function returns the

value .false. to the main program. The main program then prints the message SEAT CANCELED or WRONG CANCELLATION respectively.

e. If option 0 is chosen, the main program stops immediately if no changes were made to the array seats. otherwise, the main program closes the data file flight and then opens it to write into the data file the new seating arrangement stored in the array seats before stopping.

9.5 Solutions to Exercises

1. For each of the following subprograms, appropriate changes must be made to the subroutine MENU on page 190 and the main program on page 192.

a.

```
      SUBROUTINE LISTGR(ID, GRADES, N )
      INTEGER N, GRADES(N), ID(N), SID, SEARCH, K
      PRINT*, 'ENTER STUDENT ID'
      READ*, SID
C USING SEARCH FUNCTION ON PAGE 189
      K = SEARCH(ID, N, SID)
      IF (K .NE. 0)THEN
          PRINT*,'GRADE OF ID #', SID,' IS ', GRADE(K)
      ELSE
          PRINT*,'ID #', SID,' DOES NOT EXIST'
      ENDIF
      RETURN
      END
```

b.

```
      SUBROUTINE LISALL(ID, GRADES, N )
      INTEGER N, GRADES(N), ID(N), SGR, SEARCH, K
      PRINT*, 'ENTER STUDENT GRADE'
      READ*, SGR
      PRINT*,'ID OF STUDENTS WITH GRADE = ', SGR
      DO 10 K = 1, N
          IF( GRADE(K) .GE. SGR)  PRINT*, ID(K)
10    CONTINUE
      RETURN
      END
```

c.

```
      REAL FUNCTION AVERAG(GRADES, N)
      INTEGER N, GRADES(N), K
      REAL SUM
      SUM = 0
      DO 10 K = 1, N
          SUM = SUM + GRADE(K)
10    CONTINUE
      AVERAG = SUM / N
      RETURN
      END
```

d.

```
      SUBROUTINE LISMAX(ID, GRADES, N)
      INTEGER N, GRADES(N), ID(N), INDEX, MAXGRD, K
      INDEX = 1
      MAXGRD = GRADES(1)
      DO 10 K = 1, N
          IF( GRADES(K) .GT. MAXGRD) THEN
              MAXGRD = GRADES(K)
              INDEX  = K
          ENDIF
10    CONTINUE
      PRINT*,'MAXIMUM GRADE = ', MAXGRD
      PRINT*,'ID OF STUDENT WITH MAXIMUM GRADE = ', ID(INDEX)
      RETURN
      END
```

e.

```
      SUBROUTINE LISMIN(ID, GRADES, N )
      INTEGER N, GRADES(N), ID(N), INDEX, MINGRD, K
      INDEX = 1
      MINGRD = GRADES(1)
      DO 10 K = 1, N
          IF( GRADES(K) .LT. MINGRD) THEN
              MINGRD = GRADES(K)
              INDEX  = K
          ENDIF
10    CONTINUE
      PRINT*,'MINIMUM GRADE = ', MINGRD
      PRINT*,'ID OF STUDENT WITH MINIMUM GRADE = ', ID(INDEX)
      RETURN
      END
```

f.

```
      SUBROUTINE LISIDS(ID, GRADES, N )
      INTEGER N, GRADES(N), ID(N), K
      REAL AVERAG, AVG
C USING AVERAGE FUNCTION IN  PART C
      AVG  = AVERAG (GRADES, N)
      PRINT*, 'ID OF STUDENTS  ABOVE AVERAGE'
      DO 10 K = 1, N
          IF( GRADE(K) .GT. AVG)  PRINT*, ID(K)
10    CONTINUE
      RETURN
      END
```

Ans 2.

```
      INTEGER SEATS(6,3), EMPTY(18,2), NEMPTY, OPTION,ROW,CLMN
      INTEGER J, K
      LOGICAL RESERV, CANCEL, CHANGE
      OPEN(UNIT=40, FILE = 'FLIGHT', STATUS = 'OLD')
      DO 10 J = 1, 6
          READ(40,*)(SEATS(J,K), K=1,3)
10    CONTINUE
      CHANGE = .FALSE.
      CALL MENU(OPTION)
15    IF(OPTION .NE. 0)THEN
          IF(OPTION .EQ. 1)THEN
              PRINT*,'THE NUMBER OF EMPTY SEATS = ', NEMPTY(SEATS)
          ELSEIF(OPTION .EQ. 2)THEN
              CALL ESEATS(SEATS, EMPTY, N)
              PRINT*,'EMPTY SEATS:'
              DO 20 J = 1, N
                  PRINT*,(EMPTY(J,K), K = 1, 2)
20            CONTINUE
          ELSEIF(OPTION .EQ. 3)THEN
              PRINT*,'ENTER NEEDED SEATS ROW AND COLUMN NUMBER'
              READ*,ROW, CLMN
              IF(RESERV(SEATS, ROW, CLMN))THEN
                  PRINT*,'SEAT RESERVED'
                  CHANGE = .TRUE.
              ELSE
                  PRINT*,'SEAT NOT AVAILABLE'
              ENDIF
          ELSEIF(OPTION .EQ. 4)THEN
              PRINT*,'ENTER ROW# AND COLUMN# OF THE SEAT TO CANCEL'
              READ*,ROW, CLMN
              IF(CANCEL(SEATS, ROW, CLMN))THEN
                  PRINT*,'SEAT CANCELED'
                  CHANGE = .TRUE.
              ELSE
                  PRINT*,'WRONG CANCELLATION'
              ENDIF
          ELSE
              PRINT*,'WRONG OPTION'
          ENDIF
          CALL MENU(OPTION)
          GOTO 15
      ENDIF
      IF(CHANGE)THEN
          CLOSE(40)
          OPEN(UNIT=40, FILE = 'FLIGHT', STATUS = 'OLD')
          DO 25 J = 1, 6
              WRITE(40,*)(SEATS(J,K), K = 1, 3)
25        CONTINUE
      ENDIF
      END
```

```
      SUBROUTINE MENU(OPTION)
      INTEGER OPTION
      PRINT*,'***** FLIGHT RESERVATION *****'
      PRINT*,'1. NUMBER OF EMPTY SEATS'
      PRINT*,'2. EMPTY SEATS '
      PRINT*,'3. RESERVE SEAT'
      PRINT*,'4. CANCEL SEAT'
      PRINT*,'5. EXIT'
      PRINT*,' ENTER YOUR OPTION:'
      READ*,OPTION
      RETURN
      END
```

```
      INTEGER FUNCTION NEMPTY(SEATS)
      INTEGER SEATS(6,3), J, K
      NEMPTY = 0
      DO 30 J = 1 , 6
          DO 35 K = 1 , 3
              IF(SEATS(J,K) .EQ. 0 )THEN
                  NEMPTY = NEMPTY + 1
              ENDIF
35        CONTINUE
30    CONTINUE
      RETURN
      END
```

```
      SUBROUTINE ESEATS(SEATS, EMPTY, N)
      INTEGER N, SEATS(6,3), EMPTY(18,2), J, K
      N = 1
      DO 40 J = 1, 6
          DO 45 K = 1, 3
              IF(SEATS(J,K) .EQ. 0 )THEN
                  EMPTY(N,1)= J EMPTY(N,2)= K
                  N = N + 1
              ENDIF
45        CONTINUE
40    CONTINUE
      N = N - 1
      RETURN
      END
```

```
      LOGICAL FUNCTION RESERV(SEATS, ROW, CLMN)
      INTEGER SEATS(6,3), ROW, CLMN
      RESERV = .FALSE.
      IF(ROW .GE. 1 .AND. ROW .LE. 6)THEN
          IF(CLMN .GE. 1 .AND. CLMN .LE. 3)THEN
              IF(SEATS(ROW,CLMN) .EQ. 0 )THEN
                  SEATS(ROW,CLMN) = 1
                  RESERV = .TRUE.
              ENDIF
          ENDIF
      ENDIF
      RETURN
      END
```

```
      LOGICAL FUNCTION CANCEL(SEATS, ROW, CLMN)
      INTEGER SEATS(6,3), ROW, CLMN
      CANCEL = .FALSE.
      IF(ROW .GE. 1 .AND. ROW .LE. 6)THEN
          IF(CLMN .GE. 1 .AND. CLMN .LE. 3)THEN
              IF(SEATS(ROW,CLMN) .EQ. 1 )THEN
                  SEATS(ROW,CLMN) = 0
                  CANCEL = .TRUE.
              ENDIF
          ENDIF
      ENDIF
      RETURN
      END
```

10 ADVANCED TOPICS

In this chapter, we will expand on earlier topics discussed in this book. We introduce more advanced character operations, N-dimensional arrays, double precision and complex data types.

10.1 Character Operations

FORTRAN provides the capability of operating on character data. But what kinds of operations make sense on character strings ? Certainly the arithmetic operators: +, -, *, / and logical operators: NOT, AND, OR do not make sense with respect to character data. In this section, we shall highlight the kinds of operations that we can apply on strings.

10.1.1 Character Assignment

Character constants can be assigned to character variables using an assignment statement. If the length of a character constant is shorter than the character variable length, blanks are added to the right of the constant. If the length of a character constant is longer than the character variable length, the excess characters on the right are ignored.

Example 2: *What will be printed be the following program?*

```
      CHARACTER *5 MSG1 , MSG2
      MSG1 = 'GOOD'
      MSG2 = 'EXCELLENT'
      PRINT*, MSG1, MSG2
      END
```

Solution:

```
GOOD EXCEL
```

Notice that MSG1 contains the word GOOD followed by 1 blank; an equivalent statement would be

```
      MSG1 = 'GOOD '
```

while MSG2 contains 'EXCEL'.

Example 2: *What will be printed be the following program?*

```
      CHARACTER *5 MSG1 , MSG2
      MSG1 = 'GOOD1'
      MSG2 = 'EXCELLENT'
      PRINT*, MSG1, MSG2
      END
```

Solution:

```
GOOD1EXCEL
```

Notice that there is no automatic blanks between the values of character variables.

A character variable can be used to initialize another character variable as follows:

```
      CHARACTER BTYPE1*3 , BTYPE2*3
      BTYPE1 = 'AB+'
      BTYPE2 = BTYPE1
```

Both variables, BTYPE1 and BTYPE2, contain the character string 'AB+'.

10.1.2 Comparison of Character Strings

To perform the comparison, the following points have to be considered:

1. A collating sequence includes all possible characters from lowest to the highest values. Two standard sequences are known: ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code). In the following table the number that represent a character is equal to the sum of its row number and column number. b represents the space character. Gaps in the tables represent unprintable or control characters.

ASCII Table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32	b	!	“	#	$	%	&		(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

EBCDIC Table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32																
48																
64	~~b~~										~~e~~	.	<	(	+	|
80	&										!	$	*	)	;	¬
96	-	/									|	,	%	_	>	?
112											:	#	@	‘	=	“
128		a	b	c	d	e	f	g	h	i						
144		j	k	l	m	n	o	p	q	r						
160		~	s	t	u	v	w	x	y	z						
176																
192	{	A	B	C	D	E	F	G	H	I						
208	}	J	K	L	M	N	O	P	Q	R						
224	\		S	T	U	V	W	X	Y	Z						
240	0	1	2	3	4	5	6	7	8	9						

These sequences are based on the numeric value used to represent a character in order to store that character in the computer memory. The ASCII and the EBCDIC sequences use different numeric values for each character. An important point to note here is that the numeric values associated with alphabetic characters do not appear in a continuous numeric sequence in either the ASCII or the EBCDIC character sets. But the numeric values of numeric characters ('0','1', etc.) appear in a continuous sequence in both character sets. Also note that the numeric characters appear after the alphabetic characters in the EBCDIC collating sequence while they appear before in the ASCII collating sequence.

2. All of the relational operators: .EQ. , .NE. , .LT. , .LE. , .GT. and .GE. can be used to compare character strings.
3. In order to compare two strings they must be equal in length. If one string is shorter than the other, FORTRAN adds blanks to the right of the shorter string so that they become of equal length.
4. The comparison of two strings starts from left to right character by character.
5. In order for two strings to be equal, they must be identical, character by character. For example, the string 'ICS ' is not equal to ' ICS' because of different position of the blank character.
6. If a character string is less than another character string, it is implied that the first string precedes the second string in the order indicated in the collating sequence. Thus 'ABC' is less than 'BCD'.
7. For clarity, sometimes, we use b to represent a blank.

Example: *What will be printed be the following program?*

```
      CHARACTER WORD1*5 , WORD2*5
      WORD1 = 'MAN'
      WORD2 = 'WOMAN'
      IF (WORD1 .LT. WORD2) THEN
          PRINT*, WORD1
      ELSE
          PRINT*, WORD2
      ENDIF
      END
```

Solution: To perform the comparison between WORD1 and WORD2 in the above program, two blanks have to be added to the right of WORD1 to be equal in length with WORD2; an equivalent statement would be **WORD1 = 'MANbb'** . Since M is less than W in the collating sequence the output would be:

```
MAN
```

10.1.3 Extraction of Substrings

Each character in a string of size N can be referred to by a number called a character position. The first position in a string is character position 1 and the last character is character position N. By specifying a starting position and a stopping position in a string, we can identify parts of a string called the *substring* . If TEXT is a character variable of size N, then TEXT(I:J) is a substring starting with the Ith character of TEXT and ending with the Jth character of TEXT, where I and J are integer values. J must be greater than or equal I; otherwise an execution error would occur. In addition, both I and J must be in the range 1,2,3,...n; otherwise they would not correspond to any character position within the variable. If I is omitted (i.e. TEXT(:J)), it is assumed to be 1. If J is omitted (i.e. TEXT(I:)), it is assumed to be N.

Example 1: *What will be printed be the following program?*

```
      CHARACTER *10 A , B
      A = 'FORTRAN 77'
      B = 'PASCAL'
      PRINT 10, A(1:4) , A(9:) , B(:3)
10    FORMAT (' ' , A4, 2X, A2, 2X, A3)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
FORT 77 PAS
```

Example 2: *Vowel Determination: Write a program that reads a character string of length 100. The program should print all the vowels in the string.*

Solution:

```
      CHARACTER TEXT*100 , VOWELS(5)*1
      READ*, (VOWELS(K), K = 1, 5)
      READ*, TEXT
      DO 10 I = 1, 100
          DO 20 J = 1, 5
              IF (TEXT(I:I) .EQ. VOWELS(J)) PRINT*, VOWELS(J)
20        CONTINUE
10    CONTINUE
      END
```

Example 3: *What will be printed be the above program if the input is*:

```
'A' 'E' 'I' 'O' 'U'
```

```
'CAT + DOG = FIGHT'
```

Solution:

```
A
O
I
```

10.1.4 String Concatenation

New character strings may be formed by combining two or more character strings. This operation is known as concatenation and is denoted by a double slash placed between the character strings to be combined.

Example: *What will be printed be the following program?*

```
      CHARACTER DAY*2, MONTH*3, YEAR*4
      DAY = '03'
      MONTH = 'MAY'
      YEAR = '1993'
      PRINT 55, MONTH//DAY//YEAR,MONTH//'-'//DAY//'-'//YEAR
55    FORMAT (' ',A9, 5X, A13)
      END
```

Solution:

```
....+....1....+....2....+....3....+....4.
MAY031993 MAY-03-1993
```

10.1.5 Character Intrinsic Functions

Just as there are some intrinsic functions for numeric data such as INT, REAL, SQRT, and MOD, there are a number of intrinsic functions designed for use with character strings. These functions are:

10.1.6 Function INDEX(c1 , c2)

The function **INDEX** takes as arguments two character strings c1 and c2. The functions returns an integer value giving the first occurrence of string c2 within string c1; otherwise zero is returned.

Example 1: *What will be printed be the following program?*

```
      CHARACTER FRUIT*6
      FRUIT = 'BANANA'
      PRINT*, INDEX(FRUIT,'NA')
      END
```

Solution:

```
     3
```

Example 2: *What will be printed be the following program?*

```
      CHARACTER STR*18
      STR = 'TO BE OR NOT TO BE'
      K = INDEX(STR, 'BE')
      J = INDEX(STR(K+1:), 'BE') + K
      PRINT*, K , J
      END
```

Solution:

```
     4    17
```

Notice that the value of J represent the location of the second occurrence of the string 'BE' in STR.

10.1.7 Function LEN(c)

The function **LEN** takes as an argument one character string c. It returns the integer length of the string c. The function is used primarily in functions and subroutines that have character string arguments.

Example 1: What will be printed the following program segment:

```
      CHARACTER TEXT*10
      PRINT*, LEN(TEXT)
```

Solution:

```
10
```

Example 2: *Frequency of Blanks: Write a function that accepts a character string and returns the number of blanks in the string.*

Solution:

```
      INTEGER FUNCTION NB(X)
      CHARACTER * (*) X
      NB = 0
      DO 10 I = 1 , LEN(X)
         IF (X(I:I) .EQ. ' ') NB = NB + 1
10    CONTINUE
      RETURN
      END
```

10.1.8 Function CHAR(i)

The function **CHAR** takes as an argument an integer value **i** and returns the ith character in the collating sequence.

Example: *What is the output of the following program?*

```
      INTEGER N
      N = 65
      PRINT*, CHAR(N)
      END
```

Solution: *Assuming ASCI code representation the program will print*

```
A
```

10.1.9 Function ICHAR(c)

ICHAR the function is the reverse of function **CHAR**. It takes as an argument a single character c and returns its position in the collating sequence. The first character in the collating sequence corresponds to position 0 and the last to n-1, where n is the number of characters in the collating sequence.

Example 1: *What is the output of the following program?*

```
      INTEGER J
      J = ICHAR('C') - ICHAR('A')
      PRINT*, J
      END
```

Solution: *Assuming ASCI code representation the program will print*

```
      2
```

Example 2: *Character Code Determination: What is the output of the following program?*

```
      CHARACTER CH(26)*1
      INTEGER CODE(26)
          READ*, CH
      DO 10 I = 1, 26
          CODE(I) = ICHAR(CH(I))
10    CONTINUE
          PRINT*, CODE
      END
```

Assume the input is

```
'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O' 'P' 'Q'
'R' 'S' 'T' 'U' 'V' 'W' 'X' 'Y' 'Z'
```

Solution:

```
193 194 195 196 197 198 199 200 201 209 210 211 212 213 214 215 216
217 226 227 228 229 230 231 232 233
```

10.1.10 Functions LGE, LGT, LLE, LLT

These functions allow comparisons to be made based on an ASCII collating sequence. They produce one of the two logical values: .TRUE., .FALSE.. Each function takes as arguments two character strings. The function LGE(STRG1, STRG2) is true if STRG1 is greater than or equal to STRG2. The LGT, LLE, LLT functions perform the comparisons *greater than* , *less than or equal* and *less than* respectively. For example, LLT('ABC', 'XYZ') would produce a .TRUE. value.

10.2 N-Dimensional Arrays

In chapter 5, one-dimensional and two-dimensional array data structures were introduced. FORTRAN provides for arrays of up to seven dimensions. A two dimensional array data structure is one that varies in two attributes, a three dimensional array data structure is one that varies in three attributes, a four dimensional array data structure is one that varies in four attributes, and an N dimensional array data structure is one that varies in N attributes. Because of similarities between two and higher dimensional arrays, this section presents three dimensional arrays only. Higher dimensional arrays are treated similarly. An example of three-dimensional arrays is the grades of students in several classes for several quizzes; such an array is declared in FORTRAN as

```
      REAL GRADES (50 , 5 , 4)
```

Where we have 50 students, 5 quizzes and 4 classes. In three dimensional arrays, as in two-dimensional arrays, the elements are stored *column-wise* with the first subscript changing fastest, the second subscript changing more slowly, and the third subscript changing the slowest. For the array declaration

```
      REAL A (2 , 2 , 2)
```

The elements are stored in the following order:

A(1,1,1)

A(2,1,1)

A(1,2,1)

A(2,2,1)
A(1,1,2)
A(2,1,2)
A(1,2,2)
A(2,2,2)

To access a three-dimensional array, a nesting of three **DO** loops is common. Also an implied **DO** loop can be used.

Example

If we have the declaration:

```
      INTEGER  A (3, 4, 5)
```

*then the following three **READ** statements do the same job of storing data in the three dimensional array **A:***

```
      READ*, A
```

```
      READ*,((A((I, J, K), I = 1, 3), J = 1, 4), K = 1,5)
```

```
      DO 10 K = 1, 5
          DO 10 J = 1, 4
              DO 10 I = 1, 3
                  READ* , A (I, J, K)
10    CONTINUE
```

10.3 Double Precision Data Type

Some applications require that calculations are performed with more precision than is normally provided by the real data type. The real data type has only seven significant digits, while the double precision data type has fourteen digits of significance.

10.3.1 Double Precision Definition

To declare variables of double precision type we use **DOUBLE PRECISION** statement as follows:

```
      DOUBLE PRECISION LIST OF VARIABLES
```

or

```
      REAL*8 LIST OF VARIABLES
```

10.3.2 Double Precision Operations

The operations that are done on variables declared as double precision will be carried out internally with fourteen significant digits. All the operations that are done on real data type, can also be done on double precision data type such as addition, subtraction, multiplication, division, and exponentiation. Expressions that involve mixed types like double precision, real, and integer will be converted automatically to double precision.

Reading double precision variables is possible and up to fourteen digits to the right of the decimal point are taken from the input stream. Printing double precision values is also possible and the output will show fourteen digits to the right of the decimal point if no formatting is used. The **FORMAT** statement can be used to print double precision

values, the **D** specification may be used to print double precision numbers. **Dw.d** format specifier is used where **w** represents the total width and **d** represents the number of digits to the right of the decimal point.

10.3.3 Double Precision Intrinsic Functions

There is a large number of mathematical functions that has real arguments and/or real results. There exists an extension to these functions to work with double precision with only one simple change, which is prefixing the function name with the letter **D** like DSIN(DX), DLOG(DX), DEXP(DX), DABS(DX), etc. DX indicates that the argument to these functions is of the type double precision.

10.4 Complex Data Type

Some applications require that calculations are performed using complex numbers rather than real numbers. A complex number is represented by two real numbers where the first is the real part and the second is the imaginary part.

10.4.1 Complex Data Type Definition

To declare variables of complex type, the following declaration statement should be used in your program:

```
COMPLEX LIST OF VARIABLES
```

10.4.2 Complex Operations

The complex constants appear in the program as two real numbers separated by a comma and enclosed between a pair of parentheses as shown below:

Example 1

```
COMPLEX VALUE
VALUE = (2.0, 3.0)
```

The operations that are done on variables defined as complex will be carried out in the same way as defined mathematically. Here is the definition of some of these operations:

Addition (a+ib) + (c+id) = (a+c) + i (b+d)

Subtraction (a+ib) - (c+id) = (a-c) + i (b-d)

Multiplication (a+ib) * (c+id) = (ac-bd) + i (ad+bc)

Division
$$\frac{(a+ib)}{(c+id)} = \frac{(ac+bd)}{(c^2+d^2)} + i\frac{(cb-da)}{(c^2+d^2)}$$
$$where \quad i = \sqrt{-1}$$

When a complex variable is read, two real numbers are taken from the input stream; one for the real part and the other for the imaginary part. Printing a complex variable will result also in two real numbers representing the real part and the imaginary part. If formatting is to be used then two **FORMAT** specifies are needed of type **F**.

10.4.3 Complex Intrinsic Functions

There is a large number of mathematical functions that has real arguments and/or real results. There exists an extension to these functions to work with complex type with only one simple change which is prefixing the function name with the letter **C** like

CSIN(CX), CLOG(CX), CEXP(CX), CABS(DX), etc. CX indicates that the argument to these functions is of the complex type. In addition there are four functions for complex type which are:

Function	Description
REAL(CX)	gives the real part of the argument
AIMAG(CX)	gives the imaginary part of the argument
CMPLX(X,Y)	gives the complex number X + i Y
CONJG(CX)	gives the conjugate of the argument

10.5 Exercises

1. What will be printed by the following programs?

```
1.      CHARACTER X(1:2)*2
        READ*, X
        PRINT 11, X
11      FORMAT (1X, 2X, I2, 2X, I2)
        END
```

Assume the input is:

```
'12' '34'
```

```
2.      CHARACTER INPUT*60, SPACE*1
        INTEGER KK, JJ
        INPUT = 'THIS IS A TEST.'
        SPACE = ' '
        KK = 1
10      JJ = INDEX(INPUT(KK:),SPACE)
        KK = KK + JJ
        PRINT*, INPUT(:KK-1)
        IF (KK.LT.INDEX(INPUT,'.')) GOTO 10
        END
```

```
3.      CHARACTER STR*10
        INTEGER LL, J, NUM
        STR = '1234'
        LL = INDEX(STR,' ')
        NUM = 0
        DO 10 J = LL-1,1,-1
            NUM = NUM + (ICHAR(STR(J:J)) - ICHAR('0'))*10**J
10      CONTINUE
        PRINT*, NUM
        END
```

```
4.      CHARACTER*7 STR, SUB*6
        INTEGER L, K
        L = 3
        SUB = 'AA'
        STR = '++++++++'
        K = INDEX(SUB,' ')
        IF (K.NE.0) L = LEN(STR) - K + 1
        STR (L/2+1:) = SUB(:K-1)
        PRINT*, STR, K, L
        END
```

```
5.    CHARACTER*1 A, B
      A = 'B'
      B = 'C'
      PRINT 11, B
11    FORMAT(1X,'B=',A)
      END
```

```
6.    CHARACTER*8 F, K, X
      F(K) = K(1:2)//'REF'//K(6:8)
      X = 'CANDEULL'
      PRINT*, F(X)
      END
```

```
7.    INTEGER FUNCTION LENGTH(A)
      CHARACTER *(*) A
      LENGTH = LEN(A)
      RETURN
      END
      CHARACTER*9 A, B, C*6
      INTEGER LENGTH
      READ*, A, B, C
      PRINT*, (LENGTH(A)+LENGTH(B)+LENGTH(C))/5
      END
```

Assume the input is:

```
   'AN' 'EASY' 'EXAM'
```

```
8.    CHARACTER X*9, Y*4
      INTEGER L
      X = 'ABDABDA'
      Y = 'HIJK'
10    L = INDEX(X, 'A')
      IF (L.NE.0) THEN
          X(L:L) = '*'
          GOTO 10
      ENDIF
      PRINT*, LEN(X), X//Y
      END
```

```
9.    CHARACTER*30 S1, S2
      S1 = 'TODAY IS SATURDAY'
      S2 = 'EXAM 201 + EXAM 101'
      PRINT 11, S1(10:)
      PRINT 22, S2(10:)
11    FORMAT(' ',10X,A)
22    FORMAT(A)
      END
```

```
10.   LOGICAL LEQ, X, Y, EQAL(4)
      CHARACTER*20 L(8)
      INTEGER K, L
      LEQ(X,Y) = .NOT.X.AND..NOT.Y
      READ*, L
      K = 1
      DO 10 J = 1,7,2
          EQAL(K) = LEQ(LGT(L(J),L(J+1)), LLT(L(J),L(J+1)))
          K = K + 1
10    CONTINUE
      PRINT*, EQAL
      END
```

Assume the input is:

```
    'EXAM DAY','VACATION DAY','SUCCESS','FAILURE'
    'EASY','DIFFICULT','BE HAPPY','BE HAPPY'
```

```
11.    INTEGER WC, CC, J, K
       CHARACTER SENT*30, BLANK
       WC = 0
       SENT = 'I HAVE FORTRAN CLASSES.'
       J = 0
       BLANK = ' '
       CC = INDEX(SENT(J+1:),' .') - 1
10     K = INDEX(SENT(J+1:),BLANK)
       IF (K.NE.0 .AND.  J.LT.CC) THEN
           WC = WC + 1
           J = K
           GOTO 10
       ENDIF
       IF (CC.NE.0) WC = WC + 1
       CC = CC - WC + 1
       PRINT*, WC, CC, J
       END
```

```
12.    CHARACTER*1 FUNCTION LCHAR(STR)
       CHARACTER*20 STR
       INTEGER LAST
       LAST = 20
       10 IF (STR(LAST:LAST).EQ.' ') THEN
           LAST = LAST - 1
           GOTO 10
       ENDIF
       LCHAR = STR(LAST:LAST)
       RETURN
       END
       CHARACTER LCHAR*1, LINE*20
       READ*, LINE
       PRINT*, LCHAR(LINE)
       END
```

Assume the input is:

```
    'GOOD FINAL EXAM'
```

```
13.   SUBROUTINE INSERT(STR,SUBSTR,AFTER,RESULT,FLAG)
      CHARACTER *(*) STR, SUBSTR, AFTER, RESULT
      LOGICAL FLAG
      INTEGER IPOS
      IPOS = INDEX(STR,AFTER)
      IF (IPOS.EQ.0) THEN
          FLAG = .FALSE.
      RETURN
      ENDIF
      FLAG = .TRUE.
      LENAFT = LEN(AFTER)
      LENWOR = LEN(SUBSTR)
      LENSTR = LEN(STR)
      INSPOS = IPOS+LENAFT
      RESULT = STR(:INSPOS)//SUBSTR//STR(INSPOS:)
      RETURN
      END
      CHARACTER STR*13, S1*7, S2*3, RES1*22, RES2*28
      LOGICAL FLAG
      READ*, STR
      READ*, S1, S2
      CALL INSERT(STR,S1,S2,RES1,FLAG)
      READ*, S1, S2
      CALL INSERT(RES1,S1,S2,RES2,FLAG)
      IF (FLAG) THEN
      PRINT 5, RES2
      ELSE
      PRINT 6
      ENDIF
5     FORMAT(' ','RESULT = "',A,"")
6     FORMAT(' ','NO MATCH')
      END
```

Assume the input is:

```
   'ICS 101 EXAM'
   'FORTRAN', '101'
   'FINAL','101'
```

```
14.   CHARACTER*4 ONE, TWO, THREE, FOUR
      ONE = '+'
      TWO = ONE // ONE
      THREE = ONE // TWO
      FOUR = TWO // (ONE // ONE)
      PRINT*, 'ONE =', ONE
      PRINT*, 'TWO =', TWO
      PRINT*, 'THREE=',THREE
      PRINT*, 'FOUR =',FOUR
      END
```

```
15.    CHARACTER CH*3
       INTEGER A(3),I, J, K, L, M, N
       READ*, (A(J),J=1,2)
       L = 1
       M = 2
       N = 1
       CH = 'ICS'
       DO 10 I = 1,2
           DO 20 J = L,M,N
              PRINT*, (CH(K:K),K=1,A(J))
20         CONTINUE
           K = L
           L = M
           M = K
           N = -1
10     CONTINUE
       END
```

Assume the input is:

```
1 2
```

2. How many characters one can store in each variable in the following declaration?

```
CHARACTER*10 A, B(-2:3), C(2,5:10)*5
```

3. Assume that the only declaration statements in a FORTRANprogram are the following:

```
INTEGER A(1:10),B(3,5)
CHARACTER*7 NUM(50), NAME, CH, C
```

Which of the following statement(s) is (are) correct FORTRAN statement(s) ?

```
1.     NUM(2)(2:2) = '2'
2.     A(3:3) = 2
3.     (A(K) = A(K)+2, K = 1,10)
4.     NAME(:3) = NAME(3:)
5.     NUM(2) = B(2,2)
```

4. From the INPUT strings :

```
'THIS' 'ASY' 'VERY' 'EXAM'
```

generate the message

```
THIS IS EASY
```

by completing the print statement in the following program

```
CHARACTER A(2,2)*4
READ*, A
PRINT*,
END
```

Hint (Use substring and concatenation of the INPUT strings)

5. Complete the missing parts to produce the expected output:

```
CHARACTER*11 NAME, COURSE*6
NAME = 'COMPUTER'
COURSE = 'ICS101'
NAME(  (1)   ) = COURSE(   (2)   )
PRINT*, NAME
END
```

The expected output :

```
COMPUTER101
```

Q6) A palindrome is a word of text that is spelled the same forward and backward. The string 'RADAR' is an example of palindrome. Write a FORTRAN program to tell whether an INPUT string of length 60 is a palindrome or not.

7. Write a FORTRAN program that will do the following :
   - Read N, the number of students.
   - Read N data lines, each line contains a student ID, major, course code and grade. The program stores the data into a two-dimensional character array (CLASS) of size 20×4 such that each element has a length of 7 characters.
   - Print all those students who have a major CE and a course code ICS101 and a grade A.
8. Write a FORTRAN program which reads a character string STR of length 7 characters, and an integer array LIST of 7 elements. Then the program should print the string in the order of the numbers stored in the array LIST.

   For example: If STR = 'RNFROTA' and LIST = 3 5 1 6 4 7 2

   Then your program outputs the 3rd, 5th, 1st,... characters from STR.

   The output should look like the following (Use **FORMAT**)

```
....+....1....+....2....+....3....+....4.
DECODED STRING = FORTRAN
```

   Assume the following data:

```
'RNFROTA'
3,5,1,6,4,7,2
```

9. Write a FORTRAN program that accepts a string INPUT (at most 60 characters long), and a string PAT (exactly one character long). Then it should find the number of times string PAT is found in the string INPUT and replace every occurrence of PAT by '*'.
10. Consider the following FORTRAN statements

```
CHARACTER * 3 STR*5, X
STR = 'APPLE'
```

Which of the following statements will place the string APL in variable X?

```
i.    X = STR(1:1)//STR(3:3)//STR(4:4)
ii.   X = STR(1:1)//STR(3:4)
iii.  X = STR(1:2)//STR(3:4)
iv.   X = STR(:2)//STR(3:)
```

11. Write a FORTRAN program that:
    - a) Reads a sentence of upto 70 characters long.
    - b) Replaces each blank within the sentence by the character '$' and prints out the new sentence.
    - c) Places each vowel in the sentence into a new character string called NEW and prints out the string NEW.

    Note: The sentence is terminated by a full stop.

    Vowels are alphabets A, E, I, O and U.

10.6 Solutions to Exercises

Ans 1.

1. ERROR: TYPE MISMATCH IN **FORMAT**
2. THIS
   THIS IS
   THIS IS A
   THIS IS A TEST.
3. 43210
4. ++AA 3 5
5. B=C
6. CAREFULL
7. 4
8. 9*BD*BD* HIJK
9. EXAM 101 SATURDAY
10. F F F T
11. 1 -1 0
12. M
13. RESULT = 'ICS 101FINAL FORTRAN EXAM '
14. ONE =+
    TWO =+
    THREE=+
    FOUR =+
15. I
    IC
    IC
    I

Ans 2.

A) 10
B) 60
C) 60

Ans 3

1 and 4

Ans 4.

```
      PRINT*, A(1,1)//' '//A(1,1)(3:4)//' E'//A(2,1)
```

Ans 5.

(1) 9:10
(2) 4:6

Ans 6.

```
      CHARACTER INPUT*60
      LOGICAL PALIN
      INTEGER K
      READ*, INPUT
      PALIN = .TRUE.
      K = 1
10    IF(PALIN .AND. K .LE. 30) THEN
          IF (INPUT(K:K) .NE. INPUT(61-K:61-K)) PALIN = .FALSE.
          K = K + 1
          GOTO 10
      ENDIF
      PRINT*, PALIN
      END
```

Ans 7.

```
      CHARACTER*7 CLASS(20,4)
      LOGICAL COND1, COND2, COND3
      INTEGER K, N
      READ*,N
      DO 10 K = 1, N
          READ*, (CLASS(K,J), J = 1 , 4)
10    CONTINUE
      DO 20 K = 1 , N
          COND1 = CLASS(K,2) .EQ. 'CE'
          COND2 = CLASS(K,3) .EQ. 'ICS101'
          COND3 = CLASS(K,4) .EQ. 'A'
          IF(COND1 .AND. COND2 .AND. COND3) PRINT*,CLASS(K,1)
20    CONTINUE
      END
```

Ans 8.

```
      CHARACTER STR*7
      INTEGER LIST(7)
      INTEGER K
      READ*, STR
      READ*, (LIST(K), K = 1 , 7)
      PRINT1, (STR(LIST(K): LIST(K)), K = 1 , 7)
1     FORMAT(1X, 'DECODED STRING = ', 7A)
      END
```

Ans 9.

```
      CHARACTER INPUT*60, PAT*1
      READ*, INPUT
      READ*, PAT
      NT = 0
10    K = INDEX(INPUT, PAT)
      IF (K .NE. 0) THEN
          NT = NT + 1
          INPUT(K:K) = '*'
          GOTO 10
      ENDIF
      PRINT*, 'THE NUMBER OF TIMES PAT OCCURRED = ', NT
      END
```

Ans 10.

I amd II

Ans 11.

```
      CHARACTER SENT*70, NEW*70, VOWLS*5
      INTEGER K, M
      READ*, SENT
      VOWLS = 'AEIOU'
      NEW = ' '
10    K = INDEX(SENT , ' ')
      IF (K .NE. 0) THEN
          SENT(K:K) = '$'
          GOTO 10
      ENDIF
      PRINT*, SENT
      M = 0
      DO 20 K = 1 , 70
          IF (INDEX(VOWLS , SENT(K:K)) .NE. 0) THEN
              M = M + 1
              NEW(M:M) = SENT(K:K)
          ENDIF
20    CONTINUE
      PRINT*, NEW
      END
```

Index