COMPUTER AIDED LEARNING OF FORTRAN PROGRAMMING

BY

ABDUL MUHAMMAD

PGD / MCS / 162 / 96

DEPARTMENT OF MATHEMATICS / COMPUTER SCIENCE

FEDERAL UNIVERSITY OF TECHNOLOGY

MINNA, NIGERIA

MARCH, 1998

COMPUTER AIDED LEARNING OF FORTRAN PROGRAMMING

BY

ABDUL MUHAMMAD

(PGD /MCD / 162 / 96)

A PROJECT SUBMITTED TO THE DEPARTMENT OF MATHEMATICS / COMPUTER SCIENCE, FEDERAL UNIVERSITY OF TECHNOLOGY , MINNA, NIGERIA. IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF POST GRADUATE DIPLOMA IN COMPUTER SCIENCE

MARCH, 1998

CERTIFICATION

This project is certified to have been carried out by ABDULLAHI MUHAMMED, as part of the requirement for the award of the post graduate diploma in computer science in the department of mathematics and compute science, Federal University of Technology, Minna, Nigeria.

PROF. K . R ADEBOYE	PROF. K R. ADEBOYE
(PROJECT SUPERVISOR)	(HEAD OF DEPARTMENT)

(EXTERNAL EXAMINER)

ACKNOWLEDGMENT

In the Name of Almighty Allah, the most Beneficient and the most Merciful.

My profound gratitude goes to my supervisor PROF. K. R. ADEBOYE who is also the Head of Mathematics and Computer Science department for his, encouragement, and patience in reading through the manuscript and making valuable corrections despite his tight schedule. I am grateful sir. I pray unto Almighty Allah for his guidance and protection to you and your family in all endeavours (Amin).

My sincere thanks to all the Staff of the Department of Mathematics / Computer Science especially Prince AbdulRasheed Badmus who is also the coordinator of the post Graduate diploma course in computer science and all other Staff of the University community for making the course a successful one. May Allah reward all (Amin).

Over the years, particularly in the course of my academic pursuit, many of my family, lecturers and friends have really inspired me with honesty and positive spirit of hardwork with dedication and prayer. Indeed, I must offer special thanks to my late Parents (Mallama Aminat and DR Mohammed) for their Darling love and education given to me; my Grannies (Mallama Mariyam and ALH. Shuaibu) for their honest and tender care ; my Lecturers (Dr. Makarfi, Dr. Garba, Dr.Imdadi, Dr. Abdullahi, Prof. Rasheed, Dr. Babangida, Mr. Oyerinde and Mr. Quabili all of A. B. U. Zaria; Mr. Ishaku and Mr. Areo both are now from F.C.E., Zaria) for their positive Spirit of hardwork and dedication ; my Friends(Mall. Ibrahim, Mr. Taylor, Mall. Sariki, Mall. Yusuf, Mall. Ahmad, Mall. Abubakar, Yoms, Hamid, Rasaq, Abdulkarim, Eneye, Hauwa and Yeks) for whose friendly advice and wonderful time we shared together is most unquantifiable and finally to Engr. & Mrs. Abdullahi Adams for Hosting and allowing me to use his P.C to type this project.

I am equally indebted to my entire families (Mulikat, Amin, Moh'd, Mall. AbdulRahman, Dr. MomohJimoh, Habib, Saka, Bashir, Shehu, Meriyam, Sabdat, Hawawu, Halima, Aunty Jinetu, AbdulAzeez, Hamza, Ismaila, Aunty Rekiyat, Salima, Fatima, Rahana and Rasheed) for being my hommies.

Above all, I must be grateful to the Almighty Allah for given me the much needed strength and wisdom to complete this project to his glory. Alhamdu Rillahi.

ABDULLAHI MUHAMMED

ABSTRACT

The Computer Aided Learning of FORTRAN (C..A.L OF FORTRAN) Programming is lesigned to teach Users and Students the basics and fundamentals of FORTRAN Programming .anguage. The C.A.L of FORTRAN is just like a book.

A Turbo BASIC computer program was used to develop the lessons contained in C.A.L of FORTRAN, which are arranged in chapters and pages. The Computer language so chosen in designing the program has the essential features of accommodating a large volume text data and operations.

TABLE OF CONTENTS

CHAPTER FOUR
FORMATTED INPUT / OUTPUT STATEMENT

CHAPTER FIVE
ARRAYS, FUNCTIONS AND SUBROUTINES

APPENDIX

REFERENCES

CHAPTER ONE
PRELIMINARIES

1.1 GENERAL INTRODUCTION

Computers have gained widespread importance in our society as problems solving tools. These tools (computers) cannot solve our problems without being fed with instructions (programs) that control the operation of the machine for the performance of any given task. But early computers were difficult to use because of the complex coding schemes required for the representation of programs (instructions) and data. Consequently, in addition to improved hardware (the term used to described the general components of the computer system), computer manufacturers began to develop collections of programs known as systems software , which makes computers easier to use. One of the most important advances in these area was the development of high-level languages , which allow users to write programs in a language similar to natural language. One of the first high-level languages to gain widespread importance was FORTRAN; an acronym for Formula Translation.

1.2 BRIEF HISTORY OF FORTRAN PROGRAMMING LANGUAGE

FORTRAN (FORmula TRANslation) was developed by John Backus and a team of thirteen other programmers at international business machine (IBM) corporation at the united states of America over a period of three years (1954 - 1957) for IBM 704 computer. This programming language has the appearance of mathematical formulas and it is characterised by ease of programming , understanding and debugging.

The first FORTRAN compiler was developed successfully in 1957 by Backus and his research team to operate exclusively on IBM 704. As computer hardware improved, the FORTRAN language also was refined and extended. By 1958 it has undergone its second revision known as FORTRAN II and to a more machine independent version called FORTRAN IV in 1962. A committee was constituted in the same year by American National Standard Institute (ANSI) to develop an American standard FORTRAN with the objective of setting up a uniform standard which will facilitate portability of FORTRAN programs. This led to the development of FORTRAN 66 in 1966 which was widely adopted all over the world.

As computer hardware and software still continue to advance rapidly , ANSI went further to revise and update FORTRAN 66 in 1969 which resulted in the adoption of FORTRAN 77 in 1978 as the fifth revision. A large number of other high-level languages have also been developed - ALGOL , BASIC, COBOL, PASCAL, PL/1, and Ada, to name but a few. Thus, FORTRAN remains the oldest problem oriented programming language designed essentially for engineering and scientific environments. In both industrial and academic cycles, FORTRAN still maintains a dominant role in these areas.

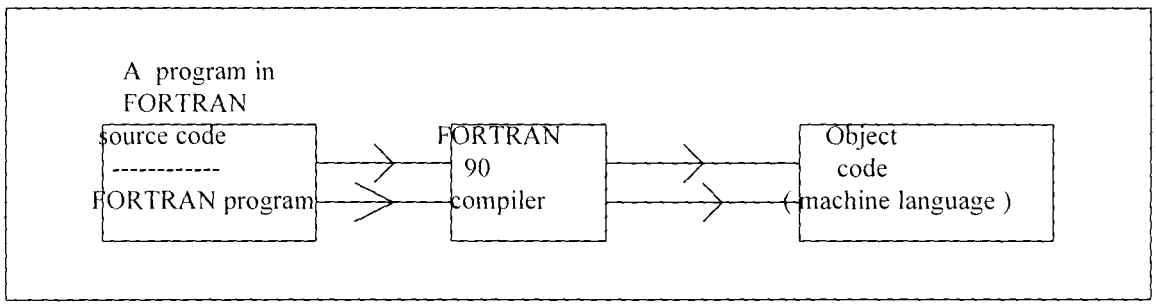
Consequently, for FORTRAN to cope with recent advances in programming concepts, ANSI naugurated a new committee known as X3J3 in 1980 to formulate a new standard for FORTRAN 90 which incorporates all of FORTRAN 77 and is largely competitive with other structured languages such as PASCAL and C language. It has further advanced well beyond them in operations on whole vectors and matrices which are especially well suited to the recently introduced super computers.

The rapid improvement that have marked FORTRAN language progression since its development in 1957 can be expected to continue in to the future.

1.3 PROGRAM TRANSLATION

A program written in a high-level language such as FORTRAN is known as a source program; (for FORTRAN language , we can say, FORTRAN source code). For most high-level languages, the instruction that makes up a source program must be translated into a machine language, that is, the only language understood by the computer, the one it was designed to recognize and obey in all its calculations and processing. This machine language is an object program. The programs that translate source programs into object programs are called compilers.

Computers, then do not understand FORTRAN. Consequently, a FORTRAN compiler is needed to translate source programs written in FORTRAN language into its machine language equivalent (or object code) before the computer can execute it. The stage of translation is known as the compilation process.



Central Memory fig 1.1

The figure above illustrates a program written in FORTRAN source code being converted into object code before execution by the computer.

1.4 PROGRAM EXECUTION

When a program is written in a FORTRAN language, it is required to perform task(s). Before the FORTRAN program performs the task, it is meant for, it has to undergo two phases. The first phase is the translation stage provided that there are no syntax errors (i.e. mistakes in the use of the language) in the FORTRAN source code and the next stage is the execution of the object code.

In FORTRAN language, there are two statements related to both these stages: the END statement and the STOP statement. Although, we may think they are similar, they perform different functions. The

END statement informs the FORTRAN compiler that there are no more instructions to translate. It must therefore appear as the very last statement in a FORTRAN program.

The STOP statement informs the computer not to execute any more instructions. In any FORTRAN program, the STOP statement will appear just before the END statement. However, it can be placed at any meaningful part in a program. Thus, the STOP statement may be optional if it immediately precedes the END statement.

1.5 THE FORTRAN CHARACTER SET

Just like every natural languages, each programming language has its own character set. When learning to write a new language the first thing that comes to mind is to learn its character set (alphabets).

The FORTRAN programming language currently has two versions popularly in use, these include FORTRAN 77 and the latest version , FORTRAN 90. The character set in FORTRAN 77 were later improved upon to give way to FORTRAN 90 character set. The following characters are admissible in FORTRAN 77

(a) Alphabetic characters

They include the 26 upper case letters.

(b) Numeric characters

The decimal digits, 0 to 9

(c) Special characters

The special characters used for arithmetic operations consists of :

(1) Arithmetic operators

There are basically five arithmetic operations that can be carried out in FORTRAN. They are:

+ (addition)

- (subtraction)

* (multiplication)

/ (division)

** (exponentiation)

The exponentiation combination '**' must be treated as a single symbol.

When typing instructions, some computers may allow lower case characters to be used since they will be converted to uppercase by the computer system. If this is not a feature of your machine, you will have to type in uppercase character.

The character set in FORTRAN 90 includes the full ASCII set (see the appendix); lower case letters are admissible.

1.6 STRUCTURE OF A FORTRAN STATEMENT

FORTRAN language was the first high-level language to be developed and as such, its structure and design reflects the days when the punched card was a major source of input to the computer. It was designed before terminals and keyboards were developed. The card has 80 columns and each column may now correspond to the "columns" on the screen.

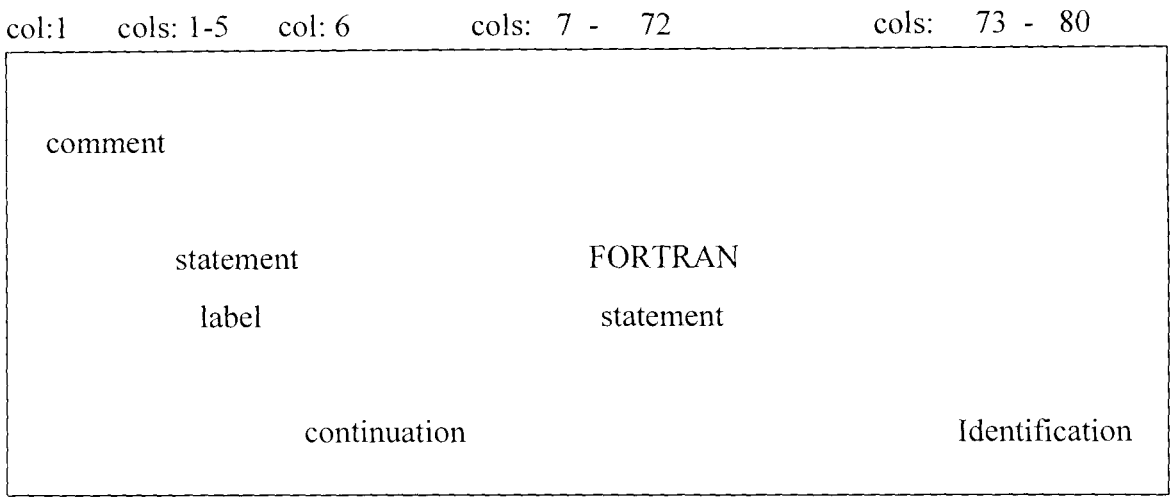


FIGURE 1.2 illustrates the use of columns in FORTRAN.

1.6.1 Statement labels

Column 1 -5 are used for statement labels in reality, these are numbers. A label is in the range 1 - 99999. Labels can be typed in the first five columns. (a one -digit label appears in column 5 two digits in columns 4 and 5, and so on.

1.6.2 Comments

Column 1 serves a special function. Typing a letter C in this column indicates to the computer that the rest of the line (2 - 80) is to be ignored during program execution.

1.6.3 FORTRAN Statement Field

The very proper FORTRAN statement may begin anywhere between column 7 - 72.

1.6.4 Continuation Code

During the process of coding FORTRAN program, an instruction may be too long to fit into one line i.e. column 7 - 72 , as a result, to identify a subsequent line as a continuation of previous line, we place a character other than zero or a blank in column 6 of the continuation line.

1.6.5 Identification Field

Column 73 - 80 is used for program identification. The part of a FORTRAN statement entered into these columns are ignored by the compiler during program execution. This field may be left blank, normally it is used to identify the program or to insert sequence numbers.

however , FORTRAN 90 , the latest version of FORTRAN contains many important features that are not available in other earlier versions like FORTRAN 77 and below. This enables FORTRAN to become more competitive with other structured languages such as C and PASCAL languages.

FORTRAN 90 allows free form source input. Since there is no restriction on a line where FORTRAN statements are to be located. Writing a trailing ampersand (&) at the end of a FORTRAN statement in a line indicates that the statement contained in the next line is a continuation of the current statement.

A comment can be placed at the end of a statement provided an exclamation mark (!) comes before the comment. Thus, anything that follows an exclamation mark on a line of a FORTRAN statement up to and including end of the line is ignored by the FORTRAN compiler during compilation.

1.7 PROGRAM COMPOSITION : The PROGRAM, STOP and END statements

Any simple FORTRAN program has the following format:

PROGRAM statement
Opening documentation
Variable declarations
Program statements
END

The PROGRAM statement has the form:

PROGRAM name

Where name is a legal FORTRAN name; that is, it consists of up to six letters or digits, the first of which must be a letter of the alphabet in FORTRAN 77. This name must be distinct from all other names used in the program. It could be chosen to indicate the purpose of the program. Thus, the first statement in the program of figure 1.3 written in FORTRAN 77 to calculate the area of a rectangle is

PROGRAM RECTAN

This is a program identifier in FORTRAN 77. The program statement is optional, but if it is used, it must appear as the first statement of the program.

Following the PROGRAM statement there should be opening documentation that explain the purpose of the program, clarifies the choice variable names and provide other pertinent information about the program. In FORTRAN 77, this documentation consists of comment lines, which are blank lines or lines having the letter C or an asterix (*) in the first position of the line. Comment lines are not considered to be program statements and may be placed any where in the program. See figure 1.3. Such comment lines can be used to clarify the purpose and structure of key parts of the program.

The variable declaration part of a program must appear next. This part consists of type statements whose purpose is to specify the type of each of the variables used in the program. For example, in figure 1.3, The statement

INTEGER LENGTH, BREADT, AREA

Specifies that LENGTH, BREADT, and AREA are integer variables in FORTRAN 77 program.

Type statements and the PROGRAM statement are called non-executable statements because they provide information that is used during compilation of the program, but they do not cause any specific action to be performed during execution. These non executable statements are followed by executable statements such as the assignment statement and input / output statements that specify the actions to be performed during the execution of the program.

The last statement of every program written in FORTRAN language must be the END statement. In FORTRAN 77, this statement indicates to the compiler the end of the program unit; it also halts execution of the program thus is an executable statement. See figure 1.3 below.

In more complex programs, it may be necessary to stop execution before the END statement is reached. In such cases, execution can be terminated with a STOP statement. This statement has the form :

STOP

or

STOP constant

Where constant is an integer constant with five or fewer digits or is a character constant. Usually the constant is displayed when execution is terminated by a STOP statement of the second form, but the precise form of the termination message depends on the compiler. Another statement called the PAUSE statement can also be used to interrupt a program rather than terminate execution and has a form similar to that of the STOP statement. That is

PAUSE

or

PAUSE constant.

Although, a program in FORTRAN 77 may have any number STOP (or PAUSE) statements, it may have only one END statement, and it may be the last statement of the program

```
      PROGRAM    RECTAN
C     This program is written in FORTRAN 77 to compute the area of a rectangle
C     of length 20 units, breadth 15 units and print the result.
      INTEGER   LENGTH, BREADT, AREA
      READ(*,10)  LENGTH, BREADT
   10 FORMAT(' ENTER THE LENGTH AND BREADTH ', 2X, I2, 3X, I2  )
      AREA = LENGTH * BREADT
      WRITE(*, 20) LENGTH, BREADT, AREA
   20 FORMAT( ' LENGTH IS ', 1X, I2, ' , ', 'BREADTH IS ', 1X, I2, 3X, 'AND',
     1 'AREA = ', IX, I3 )
      STOP
      END
```

FIGURE 1.3

The **FORTRAN 90** 's **PROGRAM COMPOSITION** takes the same format as FORTRAN 77. That is, FORTRAN 90 has the format:

PROGRAM statement
Opening documentation
Variable declarations
program statements
END

Moreover, FORTRAN 90 has some important features which are not available in FORTRAN 77. These features of FORTRAN 90 are expected to make FORTRAN compile more competitive with other highlevel language such as PASCAL, ALGOL and C language.

FORTRAN 90 allows free form source input of program since there are no restrictions where FORTRAN statements are to be located on a line. FORTRAN 90 PROGRAM name or VARIABLE name may contain up to 31 characters. The underscore '_' is allowed as part of the PROGRAM name or VARIABLE name. This greatly enhances the use of innemonic names for the both variables, files and readable FORTRAN source codes.

In FORTRAN 90 , a comment can be placed at the end of a statement provided an exclamation mark (!) comes before the comment. See figure 1.4 below. Also writing a trailing ampersand ($) at the end of a line indicates to the compiler that the next line is a continuation of the current line.

We will discuss more other features later. So let us now rewrite our program in figure 1.3 to a standard FORTRAN 90 program.

```
        PROGRAM  Rectangle   ! program identifier
!           This program is written in FORTRAN 90  to compute the area of  a rectangle
!            of length 20 units, breadth 15 units and  print the result.
        INTEGER      length , breadth, Area                  !     Type declaration
        READ(*, 10) Length, breadth                          !     entering of variables
   10   FORMAT ( 'Enter the length and breadth', 2X, I2, 3X, I2 )
        Area = Length * Breadth                              !  computation of  Area
        WRITE(*,20) Length, Breadth, Area                    !  display of result
   20   FORMAT( ' LENGTH IS ', 1X, I2, ' , ', ' BREADTH   $
        IS  ', 1X, I2, 3X,  ' AND ',  'AREA = ',  $
        1X,  I3 )
        STOP
        END
```

FIGURE 1.4

CHAPTER TWO

BASIC FORTRAN

2.1 DATA TYPES

Computer programs regardless of the language in which they are written are designed to manipulate data. Thus, we begin our discussion of FORTRAN language by considering the data types that can be processed in FORTRAN program.FORTRAN provides six data types:

1. Integer 2. Real 3. Double precision 4. Complex 5. Character 6. Logical

The first four are numeric types and are used to process different kinds of numbers. The character type is used to process data consisting of strings of characters. The logical type is used to process logical data; such data may have either .TRUE. or .FALSE. as there values. In this chapter we restrict our attention mainly to integer, real, and double precision types; logical and complex types will be discussed later.

2.2 FORTRAN CONSTANTS AND VARIABLES

2.2.1 CONSTANTS

Constants are quantities whose values do not change during program execution. They may be of numeric, character or logical type.

An Integer constant is a string of digits that does not include commas or a decimal point. A negative integer constant must be preceded by a negative sign, but a plus sign is optional for nonnegative integers. Thus,

0, 125, -5381, +1267

are valid integer constants, whereas the following are invalid for the reasons indicated

7,351 (Commas are not allowed in numeric constants)

21.3 (Integer constants may not contain decimal points.)

- -2 (Only one algebraic sign is allowed)

9 (The algebraic sign must precede the strings of digits)

A Real constants also known as single precision data is a string of digits with a decimal point and that does not include commas. They may be represented as ordinary decimal numbers or in exponential notation. In the decimal representation of real constants, a decimal point must be present , but no commas are allowed. Negative real constants must be preceded by a negative sign, but the plus sign is optional for non negative reals.

Thus,

0.0 .123 -.123 +0.123 -5. -5.00 12.531 are valid real constants, whereas the following are invalid for the reasons indicated:

21,543 (Commas are not allowed in numeric constants.)

36 (Real constants must contain a decimal point.)

Although, 2 and 2.0 are the same value, they are two different entities as far as FORTRAN is concerned. The former being an integer value, the second being a real. They are stored quite differently inside the computer.

The scientific representation of a real constant consists of an integer or decimal number, representing the mantissa or fractional part, followed by an exponent written as the letter E with an integer constant following .

For example, the real constants 337.456 may also be written as 3.37456E2 which means 3.37456 x 10 , or it may be written in a variety of other forms,

.337456E3

337.456E0

33745.6E-2

337456E-3

Character Constants also known as strings are sequence of symbols chosen from the FORTRAN character set. The sequence of character that comprise a character constants must be enclosed with apostrophes (single quotes), and the number of such characters is the Length of the constant. For example, '**ADEBOYE123-KR**' is a character constants of length 14; '**ABDULLAHI M.**' is a character of length 12, because blanks and decimal point are characters and are thus included in the constant count. If an apostrophe is to be one of the characters of a constant, it must be entered as a pair of the apostrophes;

'**DON"T**' is thus a character constant consisting of the five characters D, O, N,', and T.

2.2.2 VARIABLES in FORTRAN 77 and its predecessors can be implicitly specified if its first letter is any of the letters I, J, K, M, N while a real variable can also be also be implicitly specified if its first character is a letter from among A to H and O to Z. FORTRAN 90 eliminates this classification as every variable name has to be declared in a TYPE statement. It is advisable to use this approach in programming for FORTRAN 77.

The TYPE statements used for real variables and integer variables are of the form

REAL list

INTEGER listwhere;

list is a list of variable names separated by commas, whose TYPES are being declared

real or integer respectively. Thus, the statements

```
REAL   mass, veloc
INTEGER   const, factor, sum
```

declare mass and veloc to be of real types, and const, factor, and sum to be of integer type.

2.2 RULES FOR CREATING VARIABLE NAMES

The following guidelines should be strictly observed in assigning names to FORTRAN 77 variables:

(i) The first character must be a letter of the alphabets.

(ii) Subsequent digits must be a combination of letters and numerics

(iii) A variable name may not exceed six characters.

(iv) An integer variable name will normally have its first character from letters I to N while a real variable name must have its first character from letters A to H or O to Z, unless such a variable name is explicitly specified in the TYPE statement.

(v) Special names reserved by the FORTRAN 77 compiler cannot be used as FORTRAN variable names (see appendix for reserved words in FORTRAN 77)

(vii) The use of blank spaces are not allowed in formation of variable names.

(viii) The first appearance of variable name in a program must be assigned a numeric value using either an assignment statement , a READ statement, a PARAMETER statement, or in a previous ARITHMETIC statement.

However, all the rules spelt out above are so in FORTRAN 90 except that

(a) FORTRAN 90 accepts variable names or PROGRAM name up to 31 digits.

(b) All variable names used in the program must be declared in a TYPE statement.

(c) FORTRAN 90 permits the use of underscore in creation of variable names or PROGRAM name.

2.2.4 DOUBLE PRECISION

Real data are commonly called SINGLE PRECISION data, because each real constant is usually stored in a single memory location (i.e. it occupies 4 bytes of memory). In a machine that has 32 - bit words, for example, this provides approximately seven significant digits for each real value. As such, only the first six decimal digits are significant and the corresponding TYPE statement is

```
REAL
```

or

```
REAL*4
```

and its range of values is -3.402823E+38 to -1.1754944E-38; the number 0; and positive numbers between +1.1754944E-38 to 3.40282335E+38.

In many computations, particularly those involving iteration or long sequence of calculations, single precision is not adequate to adequate to express the precision required. To overcome this

limitation, FORTRAN generally allow the use of DOUBLE PRECISION data. Each double precision value is normally stored in two consecutive memory locations (that is, it occupies 8 bytes of memory). Thus, providing approximately twice as many significant digits as does single precision .

For example, it provide precision greater than fourteen to fifteen decimal digits and only the first fifteen digits are significant and range of values include approximately -1.7976931348623162316D+308 to -2.2250738858507201D-308; the number 0; +2.2250738858507201D-308 to 1.7976931348623162316D+308. We must note that the letter E is used for the exponent of a single precision real variable while the letter D is used for the exponent of the double precision real variable.

The names of variables, arrays, or functions that are double precision may be any legal FORTRAN names, but there types must be declared using the **Double precision type statement**, whose format are:

DOUBLE PRECISION list of variables separate by commas

or

REAL*8 list of real variables separated by comma

and

INTEGER*4 list of integer variables

The table 2.1 below represents the characteristics of REAL DATA type statements:

DATA TYPE	PRECISION	BYTES	ACCURATE DIGITS
REAL	SINGLE	4	6 TO 7
REAL*4	SINGLE	4	6 TO 7
REAL*8	DOUBLE	8	14 TO 15

We now give a simple example to illustrate the importance of using double precision arithmetic.

```
      PROGRAM   PRECIS
C     This program shows the accuracy  of using the double precision arithmetic
C     in FORTRAN 77 program
      REAL U, V, W
      DOUBLE PRECISION X, Y, Z
      U = 123456712
      X =  123456712
      V =  0.0000012345
      Y =   0.0000012345
      W =  1234.56789
      Z =  1234.56789
      WRITE(*,*) 'U = ', U, 'X = ', X
      WRITE(*,*) 'V = ', V, 'Y =', Y
      WRITE(*, *) 'W = ', W, 'Z = ', Z
      STOP
      END
```

FIGURE 2.1

The FORTRAN 90 equivalent of the above program is given by figure 2.2

```
PROGRAM    Precision_usage_in_fortran_90
! This program shows the accuracy of using double precision arithmetic in
!  FORTRAN 90 program.
REAL U, V, W          ! variables in single precision form
DOUBLE PRECISION   X, Y, Z  !  variables in double  precision  declared
  U= 123456712
X= 123456712
V= 0.0000012345
Y= 0.0000012345
W= 1234.56789
Z=  1234.56789
   !  printing
WRITE(*,*) ' U =',U, ' X =', X
WRITE(*,*) ' V =',V, ' Y =', Y
WRITE(*,*) 'W =',W, ' Z =', Z
STOP
END
```

FIGURE 2.2

2.3 ARITHMETIC OPERATIONS AND EXPRESSIONS

In industrial, academic, scientific and engineering cycles, FORTRAN still maintains a dominant role in these areas and as such, most of the statements in a FORTRAN program will involve mathematical expressions involving additions, multiplications, division, etc. as well as, standard mathematical functions like sine, cosine, absolute value, square root etc. which are available as intrinsic functions in FORTRAN compilers.

In FORTRAN, addition and subtraction are denoted by the usual plus (+) and minus (-) signs. Multiplication is denoted by an (*). Division is denoted by a slash (/) and exponentiation is denoted by a pair of asterix (**). For example, the quantity B - 4AC would be written as B**2 - 4*A*C in a FORTRAN program.

An expression containing these operations will be evaluated in accordance with the following priority rules:

1. All exponentiation are performed first; consecutive exponentiations are performed from left to right.
2. All multiplication and division are performed next, in the order in which they appear from left to right.
3. The additions and subtractions are performed last, in the order in which they appear form left to right.

Example:

2+4**2/2 = 2+16/2 = 2+8 = 10

The standard order of evaluation can be modified by using parentheses to enclose subexpressions with an expression. These subexpressions are first evaluated in the standard manner, and the result are then combined to evaluate the computed expression. If the parentheses are "nested" that is, if one of the parentheses is contained within one another, the computation in the inner most parentheses are performed first. Example:

Consider the expression (5*(11-5)**2)*4+9

The subexpression 11-5 is evaluated first, producing

(5*6**2)*4+9

Next, the subexpression 5*6**2 is evaluated to get

180*4+9

Now the multiplication is performed, giving

720+9

It is advisable that expressions containing two or more operations must be written carefully to ensure that they are evaluated in the order intended.

2.4 THE ASSIGNMENT STATEMENT

The assignment statement is used to assign values to variables and has the form

Variable = expression

Where,

The assignment symbol is "=" and expression may be a constant, another variable to which a value has previously been assigned or the result an arithmetic expression.

Example:

A = 5.0

B = A**2

The first assignment statement assigns the real constant 5.0 to the real memory variable A, and the second assigns the result of the computed expression to the real variable B.

A	5.0
B	25.0

2.5 USER FRIENDLY COMMENTS

FORTRAN language allows programmers to document their programs. This documentation provide useful information that explains what the program does, how it works, what variables it uses and how to use the program itself. This documentation enables the program to be user-friendly since the user can understand, setup, use and maintain the program.

In FORTRAN 77, typing a letter C in the column 1 of the structure of FORTRAN statement indicates to the compiler that the rest (2 - 80) is to be ignored during program execution. In this case, comment could be made into the program after typing a letter C into this column. See figure 2.1

FORTRAN 90 allows comments to be placed at the of a statement provided an exclamation mark (!) comes before the comment. Thus, anything that follows an exclamation mark on a line of a FORTRAN statement upto and including end of the line is ignored by the FORTRAN 90 compiler during compilation. See figure 2.2 above.

2.6 LIST - DIRECTED INPUT / OUTPUT STATEMENT

Computer as a useful tool for performing calculations and processing of data, it becomes necessary to inform it on how to obtain the data meant for calculations and processing, and to display its results of calculation and processing in a form easily readable and understood by man. This can be accomplished by the use of Input / output statement. The input statements enables the computer to transfer data or values from an input device such as keyboard, cardreader, tape, disk etc. to a variable names during program execution. The output statement enables the computer to display or print a constants or the contents of a variable name to an output device such as the screen of a monitor, disk, tape, card etc.

In FORTRAN language, the input / output statement are in the form of using INPUT / OUTPUT command known as READ and WRITE statements. The list directed READ statement allows data to be entered into the computer using free format READ statement. The general form of it is:

```
        READ*, list
OR
        READ(*,*) list
```

Where

READ is the key word and

list is the list of variable names or string constants separated by commas.

The asterisk in the first READ statement coincides with the second asterisk of the second READ statement and is an indication of a free formatted input. The first asterisk of the second READ statement specifies that the data are to be entered through the keyboard or an input peripheral attached to the computer. for example,

```
        READ*, LENGTH, BREADT
    OR
        READ(*,*) LENGTH, BREADT
```

are legal READ statements in FORTRAN77. During program execution, a pause will occur to allow the user to input values for the two variables length and breadt and then followed by depressing the ENTER KEY. It must be noted that length is an integer variable name and breadt is a real variable name unless a TYPE statement has been invoked at the beginning of a program to alter the normal convention.

For a user friendly program, it is necessary to display on the screen of a monitor what the computer expects the user to input at the pause stage during runtime. This can be achieved by having a print statement prior to the READ statement. For example,

```
PRINT*, 'ENTER LENGTH, BREADTH'
READ(*,*) LENGTH, BREADT
```

OR

```
WRITE(*,*) 'ENTER LENGTH, BREADTH'
READ(*,*) LENGTH, BREADT
```

The computer responds by simply displaying the string

```
ENTER LENGTH, BREADTH
```

on the visual display unit before the pause.

It must be noted that in the PRINT / WRITE statement above, the expression within the single quotes is taken as a string. To print the values of variables LENGTH, BREADT on the screen, the equivalent PRINT / WRITE statements are:

```
PRINT*,   LENGTH, BREADT
WRITE(*,*)  LENGTH, BREADT
```

OR

```
WRITE(6,*)  LENGTH,  BREADT
```

Where,

6 is the unit earlier assigned to the output peripheral device like the screen of the monitor, printer or diskette drive or the hard disk; while the * indicates that the FORTRAN compiler dictates how the output will be displayed.

CHAPTER THREE

CONTROL STRUCTURE

3.1 INTRODUCTION

Programmers should learn to write programs that are easy understand and whose logical flow is easy to follow. Such programs are more likely to be correct when first written than are poorly structured programs; and if they are wrong, the errors are easier to discover and remove. Such programs are also easier to modify, especially if the modification has to be carried out by the non- original programmer.

There are three basic control structures that govern the logical flow in a structured program. They are sequential arrangement , selection and repetition.

3.1.1 SEQUENTIAL STRUCTURE

This refers to the sequential execution of the program statements in the order in which they occur in the program. The sample programs figure 2.1 and figure 2.2 are straight -line programs in which the only control used is sequential.

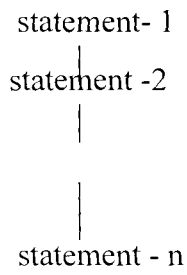


FIGURE 3.1 illustrates sequential control structure program.

3.1.2 SELECTION

This allows the selection and execution of one of a number of alternative blocks of statement it enables the programmer to introduce decision points in a program. That is, points at which a decision is made during program execution to follow one of the several course of action.

3.1.3 REPETITION

This allows the construction of a loop, that is , the controlled repetition of a block of statement. This block may be executed repeatedly for a predetermined number of times, or the repetition may be controlled by some logical expressions.

3.2 LOGICAL EXPRESSIONS

Logical expressions are used extensively in selection and repetition structures . It makes it possible to construct non numeric expression in which non numeric quantities (character and logical) are combined using appropriate operations and functions to give non numeric results.

logical expressions may be either simple or compound. The most common simple logical expression are **relational expression** of the form

expression - 1 relational - operator expression - 2

where,

expression - 1 and expression - 2 are numeric expressions, and the relational-operator in FORTRAN 77 may be any of the following :

Relational Operator	Meaning
.LT.	Is less than
.GT.	Is greater than
.EQ.	Is equal to
.LE.	Is less than
.GE.	Is greater than or egual to
.NE.	Is not equal to

The periods must be part of these relational symbols, because they help the compiler to distinguish a logical expression such as A .LE. B form the variable name or program name ALEB.

Examples of simple logical expressions:

A .GT. 3.0

AREA .EQ. 23.5

B**2 .GE. 4*A*C

If A has the value 2.8, the logical expression A .GT. 3.0 is false. If AREA has the value 23.2, theological expression AREA .EQ. 23.5 is false. If B**2 has the value 23.27 and 4*A*C has the value 10.21, then the logical expression B**2 .GT. 4*A*C is true.

Compound logical expressions are formed by combining logical expressions by using the **logical operators**

.NOT.

.AND.

.OR.

.EQV.

.NEQV.

These operators are defined as follows:

Logical Operator	Logical Expression	Definition
NOT.	.NOT. X	.NOT. X is true if X is false.
.AND.	X .AND. Y	Conjunction of X and Y: X .AND. Y is true if both X and Y are true; it is false otherwise
.OR.	X .OR. Y	Disjunction of X and Y: X .OR. Y is true if X or Y or both are true; it is false otherwise.
.EQV.	X .EQV. Y	Equivalence of X and Y: X .EQV .Y is true if both X and Y are true or both are false; it is false otherwise
.NEQV.	X .NEQV. Y	Non equivalence of X and Y: X .NEQV. Y is the negation of X .NQEV. Y; it is true if one of X or Y is true and the other is false; it is false otherwise.

Logical expression may contain more than operator, either relational or logical. For example, if A has the value 1.4, the logical expression

A+1.2 .GT. 1.1 .AND. .NOT. A .LT. 0.2 is true.

We can also insert parentheses to improve readability,

(A+ 1.2 .GT. 1.1) .AND. .NOT. (A .LT. 0.2) is true.

We note that the logical operators operate only on logical expressions. If a logical expression contains several operators, the order in which they are performed is

1. Relational operators
   (.GT. , .GE. , .EQ. , .NE. , .LT. , .LE.)
2. .NOT.
3. .AND.
4. .OR.
5. .EQV. and .NEQV.

Parentheses may be used in the usual way to modify this order.

Example : (X .AND. Y) .OR. Z

The term in the parentheses is evaluated first. Thus , this true if Z is true or both X and Y are true.

However, FORTRAN 90, the latest version of FORTRAN accepts all the logical and relational operators spelt out in FORTRAN 77. In addition, an alternative style of relational operators has been introduced into FORTRAN 90.

These include:

NEW	OLD
<	.LT.
<=	.LE.
==	.EQ.
>	.GT.
>=	.GE.
/=	.NE.

FORTRAN 90 accepts both the old and the new relational operators.

3.3 SELECTION STRUCTURE

A selection structure makes possible the selection of one of several alternative actions, depending on the value of a logical expression. The simplest selection structure is illustrated by figure 3.2. In this structure, a single statement is executed or by passed depending on whether the value of a given logical expression is true or false.

This selection structure is implemented in FORTRAN by using a **Logical IF Statement** of the form

IF (logical - expression) **statement**

If the logical expression is true, the designated statement is executed; otherwise, it is by passed. The statement must be an acceptable statement. It cannot be another Logical IF Statement, an END statement, a DO statement, or any statement that is part of the block IF structure. We note that the logical expression in a Logical IF Statement must be enclosed in parentheses.

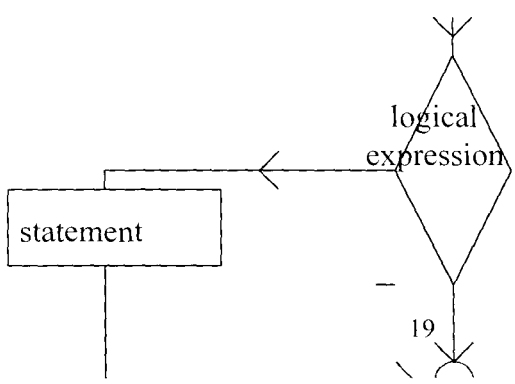


FIGURE 3.2

Example

Consider the following statements where DISCRI and Are real variables:

```
IF (DISCRI .GE. 0)  DISCRI = SQRT (DISCRI)
IF  (1.7 .LE. A .AND. A .LT. 3.2 )     PRINT*,  X
```

In the first example, the value of DISCR is compared with 0 to determine the truth or falsity of the logical expression DISCRI .GE. 0. If this logical expression is true, the assignment statement

```
DISCRI = SQRT (DISCRI)
```

is executed; otherwise, it is by passed. In the second example, IF 1.7 < A < 3.2, the value of X is printed; otherwise, the PRINT statement is by passed.

For each of these Logical IF Statements, execution continues with the next statement of the program, regardless of whether the logical expression is true or false.

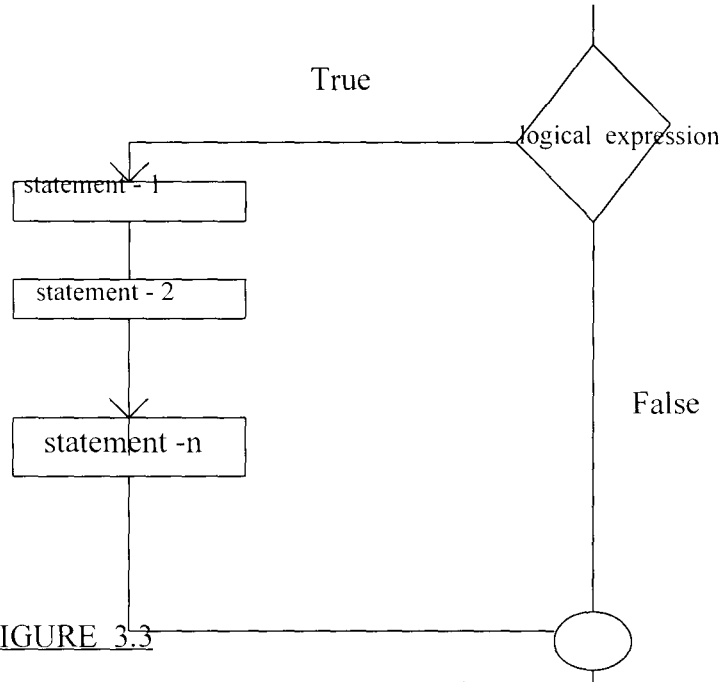
3.4 SELECTION STRUCTURE : The BLOCK IF Statement

The logical IF Statement allows the programmer to specify only a single statement to be executed if a given logical expression is true. The **Block IF statement** is more powerful because it allows the programmer to specify a block of statements to be executed if a logical expression is true. It may even be used to specify one block of statements to be executed if the logical expression is true and a different block to be executed if it is false.

The form of a block IF statement used to specify a set of statements to be executed in a given logical expression is true is

```
IF  (Logical expression )  THEN
        statement - 1  ┐
            :          │
            :          ├─ block
            v          │
        statement - n  ┘
END IF
```

If the logical expression is true, the entire block of statement between THEN and the END IF statement is executed; otherwise it is bypassed. In either case, execution continues with the next executable following the END IF statement, unless some statement within the block transfers control to some other points of the program or halts execution. This is illustrated by figure 3.3 below

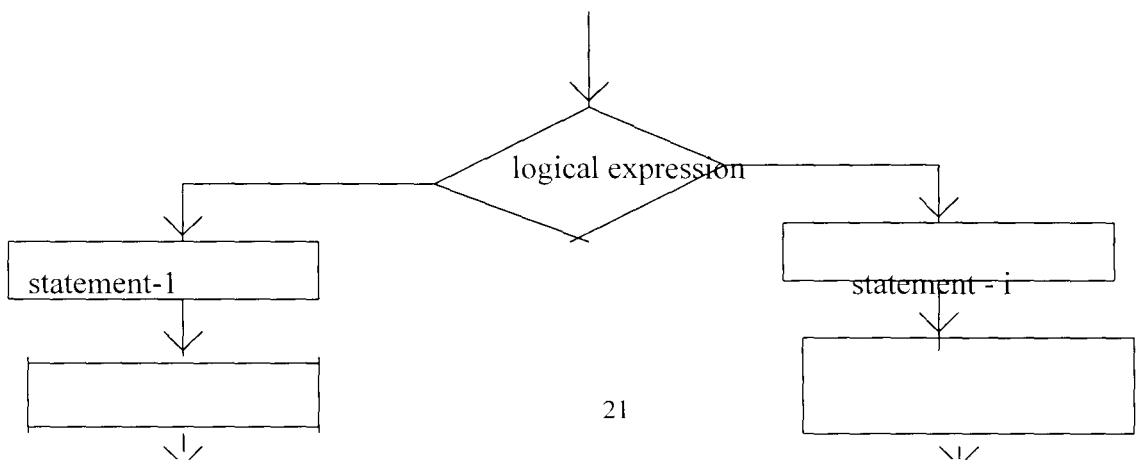


F FIGURE 3.3

The selection structure in figure 3.4 below specifies one block of statement to be executed when the logical expression is true but a different block when the logical expression is false. The form of the block IF statement that implements this selection structure is

```
IF  (logical-expression )  THEN
        statement - 1
                                    block - 1
        statement - n
ELSE
        statement - i
                                    block - 2
        statement - m
END IF
```

If the logical expression is true, the statements in block - are executed, and the statements in block -2 are bypassed; otherwise the statements in block-1 are bypassed, and the statements in block two are executed. In either case execution continues with the next executable statement following the END IF statement. (unless, of course, some statement within one of the blocks transfers control to some other point in the program or stops
execution).



statement-1 statement-ii

statement-n statement-m

FIGURE 3.4

3.5 MULTIALTERNATIVE SELECTION STRUCTURE : The IF - ELSE IF Construct

The selection structures illustrated in the preceding sections involve selecting one of the two alternatives. It is also possible to use the block IF statement to design selection structures that contain more than two alternatives. For example, consider the piecewise continuous function defined by

$$f(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } 0 < x < 1 \\ 1 & \text{if } x > 1 \end{cases}$$

This definition really consists of three alternatives and was implemented in the preceding section using a nested block IF statement of the form

```
IF (logical-expression-1) THEN
        block-1
ELSE
        IF (logical-expression-2)  THEN
          block-2
        ELSE
        block-3
END IF
END IF
```

But such compound IF statements to implement selection structures with many alternatives can become quite complex, and the correspondence among the IF , ELSE , and END IF may not be clear if indentation is not used properly.

An alternative method of implementing a multialternative selection structure is to use an IF - ELSE IF construct of the form

```
IF (logical-expression-1) THEN
        block-1
ELSE IF (logical -expression-2) THEN
        block-3
ELSE IF (logical-expression -3) THEN
        block-3
```

```
            :
            :
            :
ELSE
        block -n
END IF
```

The logical expressions are evaluated to determine the first true logical expression; the associated block of statements is executed, and execution then continues with the next executable statement following the END IF statement. The block IF statement thus implement an n-way selection structure in which exactly one of block-1, block-2, ----------, block-n is executed.

The ELSE statement and its corresponding block of statements may be omitted in this structure. In this case, if non of the logical expressions is true, execution continues with the next executable statement following the END IF.

As an example of an IF- ELSE construct, the three part definition of the preceding functions f(x) could be evaluated by

```
IF (X .LE. 0) THEN
        FVAL = -X
ELSE IF (X .LT. 1.0) THEN
        FVAL = X**2
ELSE
        FVAL = 1.0
END IF
```

We now give a simple program to illustrate the use of IF-ELSE IF construction in FORTRAN 77.

```
        PROGRAM   QUADRA
C       This  program solves a  quadratic equation using quadratic formula.
C       Variables used are : A, B, C  :- The coefficients of the quadratic equation
C       DISCRI : The  discriminant, B**2 - 4*A*C
C       ROOT1, ROOT2: The two roots of the equation.
        REAL  A, B, C, DISCRI, ROOT1, ROOT2
        WRITE(*,*) 'ENTER THE COEFFICIENTS OF THE QUADRATIC
         1  EQUATION'
        READ A, B, C
        DISCRI = B**2 - 4*A*C
        IF  (DISCRI .LT. 0)  THEN
                WRITE(*,*)' DISCRIMINANT IS' , DISCRI
                WRITE(*,*)'THERE ARE NO REAL ROOTS'
        ELSE IF (DISCRI .EQ. 0) THEN
                ROOT1 = -B/(2*A)
                WRITE (*,*)'REPEATED ROOT IS ', ROOT1
        ELSE IF
                DISCRI = SQRT(DISCRI)
                ROOT1 = (-B+DISCRI) / (2*A)
                ROOT2 = (-B-DISCRI) / (2*A)
                 WRITE(*,*) 'THE ROOTS ARE' , ROOT1, ROOT2
        END IF
        END
```

FIGURE 3.5

3.6 REPETITION STRUCTURE: The DO and CONTINUE statement

3.6.1 INTRODUCTION

A repetition structure or **LOOP** makes possible the repeated execution of one or more statements. This repetition must be controlled so that these statements are executed only a finite number of times. There are two basic types of repetition structures, which use different control mechanisms:

1. Loops controlled by some counters in which the body of the loop is executed once for each value of some control variables in a specified range of values.
2. Loops controlled by some logical expressions in which the decision to continue or to terminate repetition is determined by the truth or falsity of some logical expression.

3.6.2 The DO and CONTINUE statement

In FORTRAN 77, a repetition of the first type is called a **DO loop** and is implemented using the DO and CONTINUE statement. This structure has one of the forms

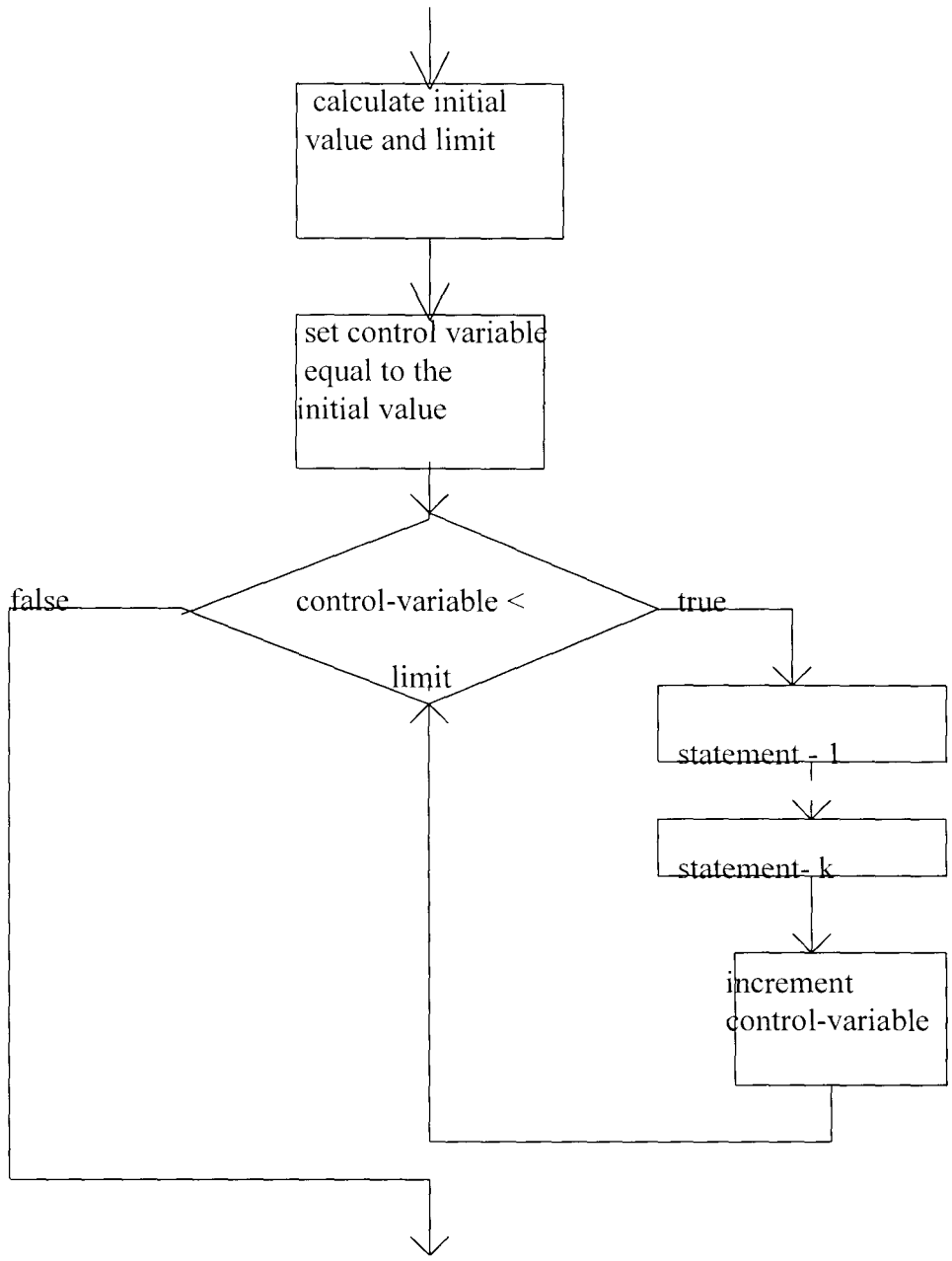
```
DO  n,   control-variable = initial-value, limit
         statement-1
                        ┐
                        ├──   body
         statement - k  ┘
n  CONTINUE
DO  n, control-variable = initial-value, limit, step-size
         statement - 1  ┐
                        ├──   body
         statement - k  ┘

n  CONTINUE
```

Here n is a statement number that is a positive integer of up to five digits; initial value, limit, and step-size are integers or real (or double precision) expressions; and step-size must be nonzero.
The comma following the statement number is optional, and we omit it in our example of this structure. The first form of a DO loop is equivalent to a DO loop of the second form in which the value of step-size is 1.

A DO loop in which the step-size is positive implements the repetition structure shown in figure 3.6 below. As the flowchart shows, when a DO loop is executed, the control variable is assigned the initial value, and the body of the loop is executed unless the initial value is greater than the limit. After the body of the loop has been executed, the control variable is incremented by the step size, and if this new value does not exceed the limit, the body of the DO loop is executed again. Execution of the DO

loop terminates when the value of the control variable exceeds the limit. Note that if the initial value is greater than the

limit, the body of the loop is never executed.



We now have an example to illustrate a DO loop whose step-size is a positive number

```
      PROGRAM
C     program written in FORTRAN 77 to illustrate a DO loop whose
C     step-size is positive
      INTEGER   NUMBER
      WRITE(*,*) 'NUMBER SQUARE VALUE'
      DO  10  NUMBER = 1, 21, 2
            PRINT*, NUMBER, NUMBER**2
 10   CONTINUE
      STOP
      END
```

FIGURE 3.7

Where NUMBER is of an integer type.

In this example, NUMBER is the control variable, the initial value is 1, the limit is 21, and the step size is 2. When this DO loop is executed, the initial one is assigned to NUMBER, and the PRINT statement is executed. The value of NUMBER is then increased by 2, and because this new value 3 is less than the limit 21, the PRINT statement is executed again. This repetition continues as long as the value of the control variable NUMBER is less than or equal to the final value 21. Thus, the output produced by this DO loop is :

NUMBER	SQUARE VALUE
1	1
3	9
5	25
7	49
9	81
11	121
13	169
15	225
17	289
19	361
21	441

If the step size in a DO loop is negative the control variable is decremented rather than incremented , and repetition continues as long as the value of control variable is greater than or equal to limit. This is illustrated in figure 3.8 below. Note that if the initial value is less than the limit, the body of the loop is never executed.

We can now have an example of a DO loop program whose step size is negative .

```
      PROGRAM   DOLOP2
C       program written in FORTRAN 77 to illustrate a DO
C     loop whose step size is negative
```

```
      INTEGER  NUMBER
      WRITE(*,*) 'NUMBER              SQUARE VALUE'
      DO  15          NUMBER = 21,  1,  -2
            PRINT*,  NUMBER,   NUMBER**2
15    CONTINUE
      STOP
      END
```

FIGURE 3.8

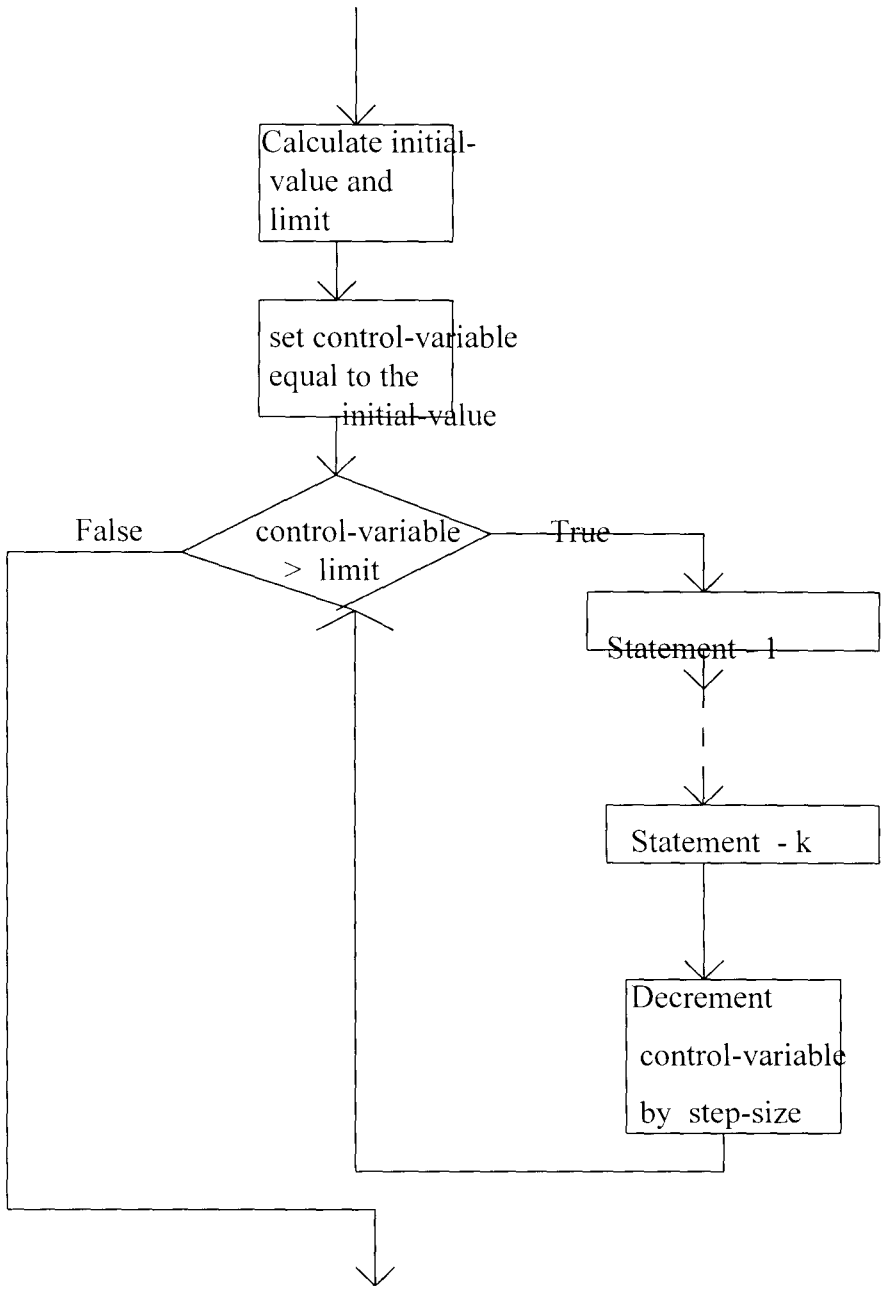


FIGURE 3.9

The body of a DO Loop may contain another DO Loop. In this case, the second DO Loop is said to be nested within the first DO Loop. As an example, consider the program in Figure 3.10 below that

calculates and displays products of the form I*J for I ranging 1 through LAST I and J ranging from 1 through LAST J for integers I, J, LAST I, LAST J and PROD. The table of product is generated by the DO Loop.

```
      DO 20 I=1, LAST I
                DO 10  J=1, LAST J
                 PROD = I*J
                 PRINT *, I,J, PROD
10               CONTINUE
20  CONTINUE
```

In the sample run, LAST I and LAST J both are assigned the value 4. The control variable I is assigned its Initial value 1, and the DO LOOP

```
      DO  10  I = 1, LAST I
              PROD =I*J
              PRINT*, I, J, PROD
10    CONTINUE
```

is executed. This calculates and displays the first four products, I*1, I*2, I*3, and I*4. The value of I is then incremented by 1, and the preceding DO Loop is executed again. This calculates and displays the next four products, 2*1, 2*2, 2*3, 2*4. The control variable I is then incremented to 3, producing the next four products, 3*1, 3*2, 3*3, and 3*4. Finally, I is incremented to 4, giving the last four products , 4*1, 4*2, 4*3, 4*4. We can now give a comprehensive program in FORTRAN 77 to illustrate this example.

```
      PROGRAM MULTIP
C     Program written in FORTRAN 77 to calculate and display a
C     list of products of two numbers.
C     Variables used are: I,J: The two numbers being multiplied
C     PROD: Their product
C     LASTI, LASTJ: The last values of I and J
      INTEGER I,J, LASTI, LASTJ, PROD
      WRITE(*,*) 'ENTER THE LAST VALUES OF THE TWO VARIABLES'
      READ(*,*)  LAST1, LASTJ
      WRITE(*'*) '   I      J        I*J    '
      DO 20 I=1, LASTJ
            DO 10 J=1, LASTJ
                  PROD = I*J
                 PRINT*' I,J, PROD
10          CONTINUE
20    CONTINUE
                                                          STOP
        END
```

FIGURE 3.10

More so, FORTRAN 90, the latest version of FORTRAN programming language accepts these method of DO Loop construction as used in FORTRAN77. in addition, a new DO Loop construct has been introduced into FORTRAN 90 which eliminates the use of statement label. The new format is

DO control variable = Initial -value, Limit, Step-Size

```
Statement -1 ┐
     |       │
     |       │   body
     |       │
Statement - k ┘
ENDDO
```

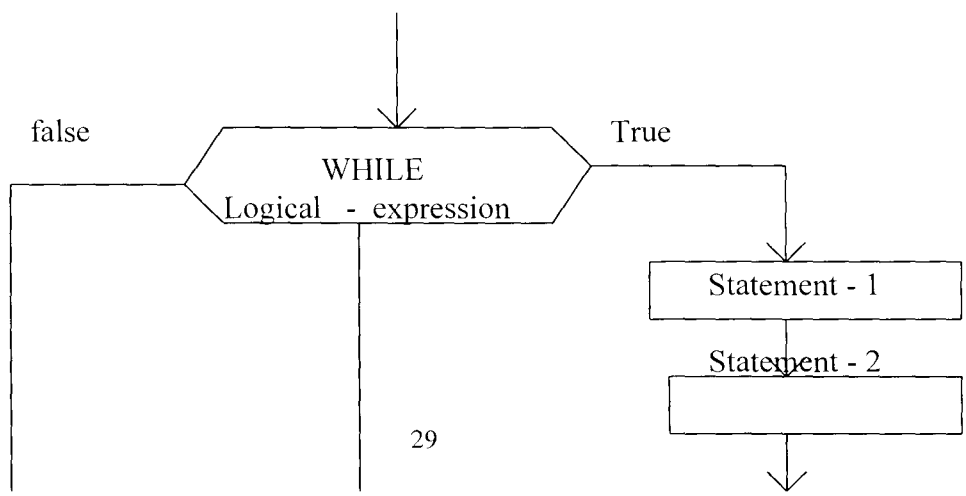
Thus, we shall now use the repetition structure DO and ENDDO Construct to modify our program. Figure 3.10 above written in FORTRAN 77 to program Figure 3.11 in FORTRAN 90 below.

```
PROGRAM  MULTIPLICATION-OF-TWO-NUMBERS
 !   Program written in FORTRAN 90 to calculate and display a list of  products
 !   of two numbers. variables used are:
 !   I, J, :  The two numbers being multiplied
 !   PRODUCT : Their product
 !   LAST-I, LAST-J : The last values of  I and J
 INTEGER I, J, LAST-I, LAST-J, PRODUCT
 PRINT*, 'Enter the last values of the two variables'
 Read(*,*) LAST-I, LAST-J
 Write(*,*) '  I          J          I*J  '
 DO     I = 1,  LAST-I
    DO     J = 1,  LAST-J
           PRODUCT = I*J
           PRINT*,    I , J  ,  PRODUCT
    ENDDO
 ENDDO
 STOP
 END
```

FIGURE 3.11

3.7 The WHILE Repetition Structure

Repetition structure in which the number of iterations is known or determined before the Loop is executed can be implemented using a DO Loop. There can be cases, a repetition structure is required for which the number of iteration is not known in advance but in which repetition continues while some logical expressions remain true. Such a repetition structure is called a WHILE LOOP and can be seen in figure 3.12 below:



Statement - k

FIGURE 3.1.2

The syntax of WHILE LOOP for FORTRAN 77 takes the form:

WHILE (Logical Expression) DO
statement - 1
———
———
statement - 2
ENDWHILE

When this statement is executed, the logical expression is evaluated; if it is true, the body of the WHILE loop consisting of statement -1, statement-2, ----------, statement -n is executed. The logical expression is then reevaluated, and if it is still true, these statements are executed again. This process of evaluating the logical expression and executing the specified statements is repeated as long as the logical expression is true. When it becomes false, repetition is terminated. This means that execution of the statements within the WHILE statement must eventually cause the logical expression to become false, since otherwise an **infinite loop** would result. Because the logical expression in a while statement is evaluated before the repetition begins, the statements that comprise the body of the while loop are not executed if this expression is initially false.

We can now have an example to illustrate the use of WHILE-DO and ENDWHILE statement in FORTRAN 77 program.

```
      PROGRAM EVEN
C  program written in FORTRAN 77 to display even numbers between 1 and 10
      INTEGER  NUMBER
      NUMBER = 2
      WHILE (NUMBER .LE. 10)  DO
            WRITE(*,*) NUMBER
            NUMBER =  NUMBER + 2
      ENDWHILE
      STOP
      END
```

FIGURE 3.13

However, the FORTRAN 90 introduces a new alternative method for while loop construct which takes the form:

WHILE (Logical Expression) DO

```
statement - 1 ──┐
_______        │   body
_______        │
satement - m ──┘
ENDDO
```

Thus, we can have an example to illustrate the use of WHILE-DO and ENDDO statement in FORTRAN 90 program.

```
PROGRAM  ODD_NUMBERS
!   program written in FORTRAN 90 to dispay odd
 !   numbers between 1 and 10
INTEGER  NUMBER
NUMBER = 1
WHILE (NUMBER .LE. 10)  DO
WRITE(*,*)  NUMBER
NUMBER = NUMBER + 2
ENDDO
STOP
END
```

FIGURE 3.14

3.8 THE TRANSFER OF CONTROL STRUCTURES

3.8.1 INTRODUCTION

Computer carries out execution of program statements in the order in which the statements are originally written from the begining to the end. However, in almost any non-trivial program, we may wish to alter this normally serquence of execution and execute a FORTAN statement that is not the next statement in sequence. This can be carried out by using a **transfer of control statement**.

Control can only be transferedto statements which are executable within the program. Thus, the transfer of control can be basically classified into two viz: Conditional and unconditional transfer of control. A conditional transfer of control is carried only if some conditions is true otherwise an unconditional transfer of control is done without testing any condition.

3.8.2 The GOTO Statement

This an unconditional transfer of control to any executable statement contained in the program.

The general form is

GOTO line-number

Where ,

GOTO is the key word

line-number is a statement label of an executable statement.

It must be an unsigned positive integer constant.

The statement label line- number may come before or after the current statement having the control.

Example

```
      PROGRAM
C  program written in FORTRAN 77 to display odd numbers
      INTEGER  N
      N = 1
   10 WRITE(*,*) N
      N = N+2
      GOTO 10
      STOP
      END
```

FIGURE 1.15

This PROGRAM ODDNUM will display odd integers 1, 3, 5, 7, 9, ---------- infinitely because the loop will be repeated indefinitely. A **conditional** statement will be required to terminate such an endless loop.

3.8.3 The IF (Logical Expression) GOTO Statement

This is a conditional transfer of control to any executable statement which is a part of the body of the main program. It provides means for getting out of a loop, and for choosing one of the several alternative steps in a process. The general form is

IF (Logical-expression) GO TO N

where,

IF is thekeyword. The logical expression enclosed in parentheses is to be tested whethe it is true or not. If this logical expression is true, control will be transfered by the command GO TO the line number or label number of the statement number N. This n must be an unsigned or positive integer constant. Example,

```
      PROGRAM    PRIME
C      program written in FORTRAN 77 to display prime numbers between
C     3 and 1000 . Variables used are: NUMBER, NNUMD, MNUMB, I
      NUMBER = 3
      NNUMB = NUMBER - 1
   20   MNUMB = 2
   30 I = (NUMBER / MNUMBER) * MNUMB
        IF (NUMBER .EQ. I)  GO TO 170
      IF ( MNUMB .EQ. NNUMB ) GO TO 60
            MNUMB = MNUMB + 1
            GO TO 30
   60 WRITE(*,*) NUMBER
  170  IF (NUMBER .EQ. 1000) GO TO 180
      NUMBER = NUMBER + 1
      GO TO 20
  180  STOP
      END
```

FIGURE 3.1.6

CHAPTER FOUR

FORMATED INPUT / OUTPUT STATEMENT

4.1 INTRODUCTION

The list directed input and output statements explained in chapter two are very easy to use , as the format for the input and out of data is automatically supplies by the FORTRAN compiler. It does not, however, permit the user to control the precise format of the data. For example, uing list directed output, one cannot specify that real values are to display with two digits to the right of the decimal point, even though this might be appropriate in some applications. The precise form of the output can be specified however, using the formatted output statement.

Sometimes input data has a predetermined form, and the programmer must design the program to read this data. This can be accomplished by using the formatted input satement.

4.2 FORMATTED OUTPUT

FORTRAN has two output statements: The **PRINT** statement and the **WRITE** statement. The PRINT statement has the form

PRINT format-identifier, output-list

The output-list is a single expresssions seperated by commas; it may also be empty, in which case the comma preceding the list is omitted. The format-identifier specifies the format in which in which values of the expressions in the output list are to be displayed.

A format identifier may be :

1. An asterisk (*)
2. The label of a FORMAT statement (or a variable to which such a label has been assigned by an ASSGN

statement).

3. A character expression or array whose value specifies the format for the output.

As we saw in chapter two, an asterisk indicates List directed output whose format is determined by the type of expressions in the output list. This is adequate when the precise form of the output is not important. However, for reports and other kinds of output in which results must appear in a precise form, list directed formatting is not adequate and format identifiers of type 2 or 3 must be used.

In the second type of format identifier, the formatting information is supplied by a **FORMAT Statement** whose label is specified. The statement has the form

FORMAT (list of format descriptors)

Each of the format descriptors that appears in the list of either the second or third type of format identifier specifies precisely the format in which to display the items in the output list. For example, some output statements that could be used to display the value 18 of the integer variable NUMBER and the value 12.24 of the real variable LENGTH are the following:

```
PRINT*, NUMBER, LENGTH
PRINT 20, NUMBER, LENGTH
```

Where,

statement 20 is the label number of the format statement.

```
20  FORMAT (1X, I5, F8.2)
```

In the FORMAT statement, 1x, I5, and F8.2 are format descriptors that specify the format in which the values of NUMBER and LENGTH are to be displayed.

As we know, list-directed output like that in the first statement is compiler dependent but might appear as follows:

```
 18   12.34
I-----------------------------------------------------------------------
```

The output produced by the second form is not compiler dependent and appears as follows:

```
 18   12.34
I-----------------------------------------------------------------
```

There are many format descriptors that may be used in format identifiers. The table 4.1 below gives a list of some of these descriptors.

FORMAT	DESCRIPTOR	USE
Iw	Iw.m	Integer data.
Fw.d		Real data in decimal.
Ew.d	Ew.dEe	Real data in scientific notation.
Dw.d		Double precision data.
Gw.d		F or E input / output, depending on the value of item

A	Aw	Character data
'x--------x'	nHx------x	Character strings
nX		Horizontal spacing
/		Vertical spacing
Tc	Tln Trn	Tab descriptors

where,

w: positive integer constant specifying the field width

d: nonnegative integer constant specifying the number of digits to the right of the decimal point.

e: nonnegative integer constant specifying the number of digits in an exponent.

x: character

c: positive integer constant separating a column number

n: positive integer constant specifying the number of columns.

FIGURE 4.1

4.2.1 INTEGER OUTPUT - The I Descriptor:

This is used to describe the format in which integer data is to be displayed. It has the form

rIw or **rIw.m**

where,

I denotes integer data

w is an integer constant indicating the width of the field in which the data is to be displayed, that is, the number of spaces to be used in displaying it.

r is an integer constant called a repetition indicator, indicating the number of such fields; for example, 2I3 is the same as I3, I3; if there is only one such field, the number one need to be given.

m is the minimum number of digits to be printed.

Integer values are right justified in the fields of the specified sizes; that is, each value is printed so that it last digit appears in the right most position of the field. For example, if the values of the integer variables I, J, and K are

```
I = 2
J = 1234
K = -13472
```

The statements

```
      PRINT 10, I, I-2, J, K
   10 FORMAT (3X, 2I5, I7, I10)
```

Produces the following output;

```
  2   0    -13472
I----------------------------------------------------------------------------
```

If an integer value (including minus sign if the number is negative) requires more spaces than are allowed by the field width specified by a descriptor, the field is field with asterisks. Thus, the statement

```
      PRINT 20, I,  I-2, J, K
   20 FORMAT (1X,  4I3)
```

will produce the following output

```
  2  0******
I-----------------------------------------------------------------------------
```

4.2.2 REAL OUTPUT --- The F Descriptor

One of the descriptors used to describe real or floating point data. It has the form:

rFw.d

where,

F denotes real (floating point) data

w is an integer constant indicating the total width of the field in which the data is to be displayed.

d is an integer constant indicating the number of digits to the right of the decimal point.

r is the repetition indicator, an integer constant indicating the number such fields, again, if there is to be only one such field, the number 1 need not be used.

Real values are printed right justified in the specified fields. For a descriptor Fw.d, if the corresponding real value has more than d digits to the right of the decimal point, it is rounded to d digits. If it has fewer than d digits, the remaining positions are field with zeros.

In most systems, values less than 1 in magnitude are displayed with a zero to the left of the decimal point (for example, 0.123 rather than .123).

For example, to display the values of the integer variables IN and OUT and the values the real variables X, Y, and Z given by

```
IN = 123
OUT = -75
X = 6.2
Y = .567
Z = 345.678
```

We could use the statement

```
      PRINT 30,  IN,  OUT,  X,  Y,  Z
   30 FORMAT(2X,  2I4, 2F6.3,  F8.3)
```

The resulting output would be

```
  123  -75   6.200  0.567 345.678
I-----------------------------------------------------------------------
```

To provide more space between the numbers and to round each of the real values to two decimal places, we use

```
40 FORMAT(2X, 2I10, 3F10.2)
```

This would display the numbers right justified in fields containing ten spaces, as follows:

```
  123       -75      6.200      0.57   345.678
I----------------------------------------------------------------------------------------------
```

If the real numbers being output requires more spaces than are allowed by the filed width specified in the descriptor, the entire field is filled with asterisks. for example,

```
      REAL ALFA
      ALFA = -567.89
      PRINT 70, 123.4
      PRINT 70, ALFA
   70 FORMAT(2X, F5.2)
```

Produces

```
        *****
      I-------------------------------------
        *****
      I-------------------------------------
```

It should be noted that for a descriptor **Fw.d**, one should have

$$w > d+3$$

to allow space for the sign of the number, the first digit, and the decimal point.

4.2.3 REAL OUTPUT - The E Descriptor

Real data may also be output in scientific notation using a descriptor of the form

rEw.d or **rEw.dEe**

where,

E indicates that the data is to be output in scientific notation.

w in an integer constant that indicates the total width of the field in which the data is to be displayed.

d is an integer constant indicating the number of decimal digits to be displayed.

r is the repetition indicator, an integer constant indicating the number of such field; it need not be used if there is only one field.

e is the number of positions to used in displaying the exponent.

Although some detail of the output are compiler dependents, real values are usually printed in normalised form - a minus sign, if necessary, followed by one leading zero, then the decimal point followed by d significant digits, and then E with an appropriate exponent in the next four spaces for the first form or e spaces for the second form.

For example, if values of real variables A, B, C and D are given by

```
REAL A, B, C, D
A = .12345.E8
B = .0237
C = 4.6E-12
D = -76.1684E12
```

The statements

```
      PRINT 60, A, B, C, D
60    FORMAT( 1X, 2E15.5, 2E15.4 )
```

produces output like the following :

```
 0.12345E+08 0.23700E-01 0.4600E-11 -0.7617E14
I-----------------------------------------------------------------------------------
```

As with the F descriptor, a field is asterisk filled if it is not large enough for the value to be permitted. It should also be noted that for a descriptor Ew.d one should have

w > d +7

or for the second form,

w > d + e +5

to allow space for the sign of the number, a leading zero, a decimal point, and E with the exponent.

4.2.4 CHARACTER OUTPUT - A Descriptor

Character constants may also be displayed by including them in the list of descriptors of a format identifier. For example, if C and D have the values .2 and 6.8 respectively,

The statements

```
      PRINT 70, A, B
70    FORMAT (1X, 'A =', F6.2, 'Y =', F6.2)
```

produces as output

```
 C=0.20   y=6.80
```

I----------

Character data may also be displayed by using an A format descriptor, of the form

rA or **rAw**

where

A denotes character data

w (if specified) is an integer constant specifying the field width.

r is the repetition indicator, an integer constant indicating the number of such fields; it may be omitted if there is only one field.

In the first form, the field width is determined by the length of the character value being displayed. In the second form, if the field width exceed the length of the character value, that value is **right justified** in the field. In contrast with numeric output, however, if the length of character value exceeds the specified field width, the output consists of the leftmost w characters. For example, the preceding output would also be produced if the labels were included in the output lists as follows:

```
      PRINT 75, 'C = ', C, 'D = ', D
75    FORMAT( 1X, A, F6.2, A, F6.2 )
```

This latter method is perhaps preferable to the first, as the format identifier can be used to print other labels and values as in

```
      PRINT 75, ' MEAN IS ', XMEAN, 'WITH
     1  STANDARD DEVIATION' , STDEV
```

4.2.3 POSITIONAL DESCRIPTORS - X and T Descriptor

There are two format descriptors that can be used to provide spacing in an output line. An X - descriptor can be used to insert blanks in an output line. It has the form

nX

where,

n is a positive integer constants that specifies the number of blanks to be inserted.

For example, if the value of N is 25 and we want to print the value of N, then the statements

```
      INTEGER N
      N = 50
      PRINT 20, N
20      FORMAT( 2X, I2 )
```

produces an output as follows:

```
 50
```

I----------

Thus, the compiler skips two column spaces and then print the value of N as 50.

The T descriptor has the form:

Tc

where,

c is an integer constant denoting the number of the space on a line at which a field is to begin. This descriptor functions much like a tab key on a type writer causes the next output field to begin at the specified position on the current line. One difference is that the value of C may be less than the current position; that is, "tabbing backward" is possible.

As an illustration of these descriptors, consider the output statement

```
PRINT 78, 'DR. K.R. ADEBOYE', 'AND', 'A. MOHAMMED'
```

together with either of the following FORMAT statements:

```
78   FORMAT( 1X, A16, 3X, A3, 3X, A11 )
```

or

```
78   FORMAT( 1X, A16, T20, A3, T23, A11 )
```

would produced an output

```
 DR. K.R. ADEBOYE   AND   A. MOHAMMED
```

I----------

4.2.5 The Slash (/) Descriptor

A single output statement can be used to display values on more than one line, with different formats, by using a slash (/) descriptor. The slash causes the output to begin on a new line. It can also be used repeatedly to skip several lines. It is also necessary to use a comma to separate a slash descriptor from other descriptors.

For example, the statements

```
      PRINT *, ' VALUES '
      PRINT*
      PRINT*
      PRINT 80, I, R, J, S
      PRINT  82, /, Y, Z
80    FORMAT( 1X, 2(I10, F10.2) )
82    FORMAT( 1X, 2E15.7)
```

could be combined in the pair of statements

```
      PRINT 85, 'VALUES', I, R, J, S, Y, Z
85    FORMAT(1X, A /// 1X, 2( I10, F10.2 ) / 1X, 2E15.7 )
```

(Note that the descriptors 1x following the slashes to indicate the control characters for the new output lines.) If the values of I, R, J, S, Y and Z are given by

```
      I = 1234
```

R = 218.1
J = 3613
S = 19.26
Y = 47.666
Z = 7.1814

Then in both cases the resulting output is

```
 VALUES
I-----------------------------------------------------------------------------------------
I-----------------------------------------------------------------------------------------
  1234   218.10  361319.26
I-----------------------------------------------------------------------------------------
  0.4766600E+02   0.7181400E+01
I-----------------------------------------------------------------------------------------
```

4.2.7 The H - Descriptor

We have seen that character constants may be displayed by including them in the list of descriptors of a format identifier; for example

```
        PRINT 30;
   30   FORMAT( 3X, 'DR. K.R. ADEBOYE  AND  A. MOHAMMED )
```

strings may also be displayed by using a Hollierith descriptor of the form

nHstring

where n is the number of characters in string. Thus, the preceding format identifier could also be written

```
   30   FORMAT( 3X, 36HDR. K.R. ADEBOYE  AND  A. MOHAMMED )
```

4.3 FORMATTED INPUT

We have seen that input is accomplished in FORTRAN by a READ statement. This statement has two forms, the simpler of which is

READ format-identifier, input-list

The input-list is a single variable or a list of variables separated by commas. The format-identifier specifies the format in which the values for the items in the input list are to be entered. As in the case of output, the format identifier may be

1. An asterisk (*)
2. The label of a FORMAT statement (or a variable to which such a label has been assigned by an ASSIGN statement).
3. A character expression or array whose values specifies the format for the input.

The most commonly used form of the READ statement is that in which the format identifier is an asterisk. As we can saw in chapter two, this form indicates list-directed input in which the format is

determined by the type of variables in which the data the data have a specific predetermined form, it may also be necessary to use a format identifier of type 2 or 3 to read these data. As in the case of output, the format identifier may be the label of a FORMAT statement of the form

FORMAT(list of format descriptors)

The format descriptors are essentially the same as those discussed for output in the preceding section. Character constants, however, may not appear in the list of format descriptors, and the colon separator is not relevant to input.

4.3.1 INTEGER INPUT

Integer data can be read using the I descriptor of the form

rIw

where,

w indicates the width of the field, that is, the number of columns to be read, and

r is the repetition indicator specifying the number of such fields.

To illustrate, consider the following example:

```
      INTEGER L, M, N
      READ 5, L, M, N
5     FORMAT( I6, I4, I7 )
```

For the values of L, M, and N to be read correctly, the numbers should be entered as follows:

The value for L is the first six columns, and the values for N is the next seven columns, with each value right justified within its field. Thus, if the values to be read are

L : -425

M : 79

N : 5378

The data may be entered as follows:

```
 -425  79   5378
I-------------------------------------------------------
```

If the format statement were changed to

```
5     FORMAT( I4, I2, I4 )
```

The data should be entered as

```
-425795378
I-------------------------------------------------------
```

with no intervening blanks. Here the first four columns are read for L the next two columns for M, and the next columns for N.

Blanks within a field read with an I descriptor can be interpreted as zeros, or they can be ignored. If they are interpreted as zeros, integer values must be right justified within their fields as in the first example above if they are to be read correctly.

4.3.2 REAL INPUT

One of the descriptors used to input real data is the F descriptor of the form

rFw.d

where,

w indicates the width of the field to be read

d is the number of digits to the right of the decimal point, and

r is the repetition counter.

There are two ways that real data may be entered:

1. The numbers may be entered with no decimal point.
2. The decimal point may be entered as part of the input value.

U : 7.35

V : -1.8

W: 65.0

X : .283

Y: 725.237

We could use the statements

```
      READ 20, U, V, W, X, Y
20    FORMAT( F3.2, 2F3.1, F3.3, F6.3 )
```

and enter the data in the following form:

7518650283725237

---

of course, we could use wider fields, for example,

```
20    FORMAT ( F4.2, 2F4.1, 2F8.3 )
```

and enter the data in the form

735 -18 650 283 725237

---

with the values right justified within their fields.

In the second method for entering real data, the position of the decimal point in the value entered overrides the position determined by the descriptor. Thus, if the number to be read is 6435.79, an appropriate descriptor would be F6.2 if the number is entered without a decimal point and F7.2, or F7.1,

or F7.0, and so on, if the number is entered with a decimal point. For example, the preceding values for U, V, W, X and Y Could be read using the statements

```
      READ 30, U, V, W, X, Y
30    FORMAT( 4F5.0, F8.0 )
```

with the data entered in the following form:

```
735 -18  65   .283  725.237
```

It should be noted that each field width must be large enough to accommodate the number entered, including the decimal point and the sign.

Real values entered in E notation can also be read using an F descriptor. Thus, for the FORMAT statement

```
30    FORMAT( 5F10.0 )
```

The data of the preceding example could also have been entered as

```
.735E1    -1.8    65.0  28.3E-2 7.25237E2
```

In this case, the E need not be entered if the exponent is preceded by a sign.

The following would therefore be an alternative method for entering the preceding data:

```
.735+1    -1.8    654.0   28.3-2 7.25237+2
```

The E descriptor may also be used in a manner similar to that for the F descriptor.

4.3.4 SKIPPING COLUMNS OF INPUT

The positional descriptors X and T may be used in the format identifier of a READ statement to skip over certain columns of data. For example, if we wish to assign the following values to the integer variables L, M and N

L : 5

M : 37

N : 224

by entering data in the form

```
L = 5  M = 37  N = 224
```

The following statements may be used:

```
      READ 40, L, M, N
40    FORMAT( 3X, I2, 6X, I3, 5X, I4 )
```

or

```
40      FORMAT( T4, I2, T12, I3, T20, I4 )
```

Columns of data are also skipped if the end of the input list is encountered before the end of the data line has been reached. To illustrate this statements

```
        READ 23, I, A
        READ 23, J, B
23      FORMAT( I5, F7.0 )
```

are used to read values for the integer variables I and J and real variables A and B from the following data lines

```
 13  2.68  47  15.2  8125 730660 2188 49
```

The values assigned to I and A are

I : 13

A : 2.68

and the values assigned to J and B are

J : 8125

B : 730660

All other information's on these two lines are ignored.

4.3.5 MULTIPLE INPUT LINES

We can recall that a new line of data is required each time a READ statement is executed. A new line of data is also required whenever a slash (/) is encountered in the format identifier for a READ statement. This may be used in case one wishes to separate some of the data entries by blank lines, remarks and the like which are to be skipped over by the READ statement. For example, the following data

```
AMOUNT TO BE PRODUCED
-----------------------------------------------------------------------------
635.00
-----------------------------------------------------------------------------
REACTION RATE
-----------------------------------------------------------------------------
(THIS ASSUMES CONSTANT TEMPERATURE
-----------------------------------------------------------------------------
6.45
-----------------------------------------------------------------------------
```

could be read by a single READ statement, and the values 635.00 and 6.45 assigned to AMOUNT and RATE, respectively in the following manner:

```
        REAL AMOUNT, RATE
        READ 40, AMOUNT, RATE
```

```
40      FORMAT( / F10.0 /// F5.0 )
```

The first slash causes the first line to be skipped so that the value 635.00 is read for AMOUNT; the three slashes then cause an advance of three lines, so that 6.45 is read for RATE.

A new line of data is also required if all descriptors have been used and there are still variables remaining in the input list for which values must be read. in this case, the format identifier is rescanned, as in the case of the output.

Thus, the statements,

```
        INTEGER I, J, K, L, M
        READ 55, I, J, K, L, M
55      FORMAT( 3I8 )
```

require two lines of input, the first containing values of I, J and K and the second, the values of L and M.

4.4 THE GENERAL READ AND THE WRITE STATEMENT

The READ and PRINT statements used thus far are simple FORTRAN input / output statements. We shall now consider more general input / output statements, the WRITE statement and the general form of the READ statement.

The WRITE statement:- This has a more completed syntax than does the PRINT statement, but it is a more general output statement. It has the form

WRITE(control- list) output- list

where,

output-list has the same syntax as in the PRINT statement and control-list may include items selected from the following:

1. A unit specifier indicating the output device.
2. A format specifier
3. Other items that are especially useful in file processing.

The control list must include a unit specifier and a format specifier as well. The unit specifier is an integer expression whose value designates the output device, or it may be an asterisk, indicating the standard output device (usually a terminal or printer).

The unit specifier may be given in the form

UNIT = unit-specifier

or simply

unit-specifier

if the UNIT = clause is not used, the unit specifier must be the first item in the control list.

The format specifier has the form:

FMT = format-identifier

or simply

Format-identifier

where format-identifier may be any of the forms allowed in the PRINT statement. If the format specifier without the FMT = clause is used, then it must be the second item in the control list, and the UNIT = clause must also be omitted for the unit specifier.

To illustrate the WRITE statement, suppose that the values of GRAV and WEIGHT are to be displayed on an output device having unit number 6. The statement

```
WRITE( 6, * ) GRAV, WEIGHT
```

0r any of the following equivalent forms

```
WRITE( 6, FMT = * ) GRAV, WEIGHT
WRITE( UNIT = 6, FMT = *) GRAV, WEIGHT
WRITE( NOUT, * ) GRAV, WEIGHT
WRITE( UNIT=NOUT, FMT=* ) GRAV, WEIGHT
```

Where,

Nout is an integer variable with value six, produce list-directed output to this device. If this device is the system standard output device, the unit number 6 may be replaced by an asterisk in any of the preceding statements;

for example,

```
WRITE(*,*) GRAV, WEIGHT
```

and each of these is equivalent to the short form

```
PRINT*, GRAV, WEIGHT
```

Formatted output of these values could be produced by statements like the following:

```
      WRITE( 6, 20 ) GRAV, WEIGHT
20    FORMAT( 1X, 2F10.2 )
      WRITE( UNIT=6, FMT=20 ) GRAV, WEIGHT
20    FORMAT( 1X, 2F10.2 )
```

The FORTRAN 77 program figure 4.1 below displays tables of numbers together with their squares, cubes and square roots.

```
      PROGRAM  TABLE
C     FORTRAN 77 program demonstrating the use of the formatted output to print a table of
values of N, square          and cube of N and the square root of N for N=1, 2, ----------,LAST where
the value of LAST is red          during execution.
      INTEGER  N, LAST
      WRITE(*, *) 'ENTER LAST NUMBER TO BE USED'
      READ*, LAST
```

```
C       printing the headings
        WRITE(*, 10) 'NUMBER', 'SQUARE', 'CUBE' , 'SQ. ROOT'
   10   FORMAT( //, 1X, A8, T11, A8, T31,  A10 /1X, 40 ('=') )
C       print table
        DO  30  N=1,  LAST
                WRITE(*, 20) N,  N**2,  N**3,  SQRT( REAL(N) )
   20   FORMAT( 1X, I6, 2I10, F10.4 )
   30    CONTINUE
        END
```

FIGURE 4.1

Sample Run

```
ENTER LAST NUMBER TO BE USED
10

 NUMBER       SQUARE      CUBE       SQ. ROOT
==========================================
1             1           1          1.0000
2             4           8          1.4142
3             9           27         1.7321
4             16          64         2.0000
5             25          125        2.2361
6             36          216        2.4495
7             49          343        2.6458
8             64          512        2.8284
9             81          729        3.0000
10            100         1000       3.1623
```

The General READ Statement

The general form of the READ statement is

READ (Control-list) input-list

Where,

control -list may include items selected from the following:

1. A unit specifier indicating the unit device
2. A format specifier
3. An END = clause giving the number of a statement to be executed when the end of data occurs.
4. Other items that are particularly useful in processing files.

As an illustration of the general for of the READ statement, suppose that values for CODE, TIME and RATE are to be reading using the input device 5. The statement

```
READ( 5, *) CODE, TIME, RATE
```

or any of the following equivalent forms

```
READ( 5, FMT=* ) CODE, TIME, RATE
READ( UNIT=5, FMT=* ) CODE, TME, RATE
READ( IN, *) CODE, TIME, RATE
READ( UNIT=IN, FMT=* ) CODE, TIME, RATE
```

where,

IN has the value 5, can be used. If this device is the system standard input device, an asterisk may be used in place of the device number in any of the preceding unit specifications; for example,

```
READ(*,*) CODE, TIME, RATE
```

Formatted input is also possible with the general READ statement; for example

```
READ( UNIT=5, FMT=10 ) CODE, TIME, RATE
```

or

```
READ(5, 10) CODE, TIME, RATE
```

where,

10 is the number of the following FORMAT statement;

```
10  FORMAT( I6,  2F6.2)
```

4.5 INTRODUCTION TO FILE PROCESSING

The programs we have written up to this point have involved relatively small amounts of input / output data. We have assumed that the input data were read from a terminal or cards and that the output was displayed either at a terminal or at a printer. This is adequate if the volume of data involved is not

large. However, applications involving large data set may be processed more conveniently if the data is stored on magnetic tape or magnetic disk or some other form of external (secondary) storage.

Magnetic tape is a plastic tape coated with a substance that can be magnetized. Such a tape stores " sound information". Information can be written onto or read from a tape using a device called a **tape drive**. A standard tape drive can record 1600 bytes per inch of tape. Therefore, a 2400 foot reel of tape can store approximately 46 million characters.

A **magnetic disk** is also coated with a substance that can be magnetized. Information's is stored on such disks in **tracks** arranged in concentric circles and is written onto or read from a disk using a **disk drive**. This device transfers information's by means of a movable read / write head, which is positioned over one of the tracks of the rotating disk. Some **disk packs** consisting of several such disks can store more than a million characters. Information stored on such auxiliary devices that is to be processed by a FORTRAN program is usually arranged in structures called **files**, and each line of data in the file is called a **record.**

Each record of a file to be used as an input file must have the entries arranged in a form suitable for reading by a READ statement. These record are read during program execution just as a card of data is read by a card reader or a line of data is read from a terminal.

For example, if the variables CODE, TEMP and PRESS are declared by

```
INTEGER CODE
REAL TEMP, PRESS
```

and the values of these variables are to be read from a file using a list-directed READ statement, this data file might have the following form:

```
37, 77.5, 30.39
22, 85.3, 30.72
1,  100.0, 29.95
78, 99.5, 29.01
      :
      :
      :
```

If values is to be read using the format statement

```
10    FORMAT( I3, 2F8.0 )
```

The file might have the form

```
37  77.5  30.39
22  85.3  30.72
```

```
1    100.0  29.95
78   99.5   29.01
       :
       :
```

whereas the format statement

```
10    FORMAT( I2, F4.1, F4.2 )
```

would be appropriate for the file

```
37 7753039
22 8533072
  110002995
789952901
       :
       :
```

<u>OPENING FILES</u>:- Before a file can be used for input or output in a FORTRAN program, it must be "opened".

This can be accomplished by using an **OPEN statement** of the form

OPEN (open-list)

Where open-list includes

1. A unit specifier indicating a unit connected to the file being opened.
2. **A FILE** = Clause giving the name of the file being opened.
3. A **STATUS** = Clause specifier whether the file is a new or an old file.

The unit specifier has the form

UNIT = integer- expression

where the value of integer- expression is a nonnegative number that designates the unit number to be connected to this file. Reference to this file by a READ or WRITE statement will be by means of this unit number.

The FILE= clause has the form

FILE = character - expression

where the value of character expression (ignoring trailing blanks) is the name of the file to be connected to the specified unit number.

The STATUS = clause has the form

STATUS = character-expression

where the value of character-expression (ignoring trailing blanks) is

'OLD'

or

'NEW'

OLD means that the file already exists in the system. NEW means that the file does not yet exist and is being created by the program: execution of the OPEN statement creates an empty file with the specified name and changes its status to OLD.

CLOSING FILES:- The **CLOSE statement** has function opposite to that of the OPEN statement and may be used to disconnect a file from its units number. This statement is of the form

CLOSE (close-list)

where close list must include a unit specifier. After a CLOSE statement is executed, the closed file may be reopened by means of an OPEN statement; the same unit number may be connected to it, or a different one may be used. All files that are not explicitly closed by means of a CLOSE statement are automatically closed when a STOP or END statement is executed.

FILE INPUT / OUTPUT

Once a file has been connected to a unit number, data can be read from or written to that file using the general forms of the READ and WRITE statements in which the unit number appearing the control list is the same as the unit number connected to the file. For example, to read values for CODE, TEMP, and PRESS from a file named INFOR, the statement

```
OPEN ( UNIT=12, FILE= 'INFOR', STATUS='OLD' )
```

open the file, and the statement

```
READ(12, *) CODE, TEMP, PRESS
```

reads the values.

Similarly, a file named REPORT to which values of CODE, TEMP, and PRESS are to be written could be created by

```
OPEN ( UNIT=13, FILE='REPORT', STATUS='NEW', )
        WRITE( 13, 30) CODE, TEMP, PRESS
   30   FORMAT( 1X, I3, F7.0, F10.2 )
```

Each execution of a READ statement caused an entire record to be read and then positions the file so that the .

next execution of a READ (WRITE) statement causes values to be read from (written to) the next record of the file. Similarly, execution of a WRITE statement writes an entire record in to the file and then

position the file so that the next execution of a WRITE (READ) statement produces output to (input from) the next record of the file.

The END = CLAUSE :- We have already noted in the preceding section that the control list of a general READ statement may contain an **END=Clause** to transfer control automatically when there are no more data values. This clause has the form

END=statement-number

where statement number is the number of an executable statement that is the next statement to be executed when the end of data is encountered. For example, the statement

```
READ( 12, *,  END=50) CODE, TEMP, PRESS
```

could be used within a loop to read values for CODE, TEMP, and PRESS form a file. When the end of the file is reached, control transfers to statement 50 which might calculate the mean temperature:

```
50    TMEAN = TSUM / COUNT
```

FILE - POSITIONING STATEMENTS

There are several FORTRAN statements that may be used to position a file. Two of these statement are

REWIND unit

and

BACKSPACE unit

where unit is the unit number connected to the file.

The **REWIND statement** positions the file at its initial point, that is, at the beginning of the first record of the file. The **BACKSPACE** statement causes the file to be positioned at the beginning of the preceding record. If the file is at its initial point, these statement have no effect.

At this juncture, we shall now design a FORTRAN 77 program to illustrate the use of file processing as shown in figure 4.2 below:

```
      PROGRAM TEMVOL
C     program to read temperatures and volumes from a file containing time, temperature,
pressure and      C     volumes readings made by some monitoring device. The temperature and
volume measurement are     C     displayed in tabular form, and the equation of the least square line
y=mx + b ( x= temperature,
C     y=volume ) is calculated. Variables used are:
C     TEMP        : temperature recorded
C     VOLUME      : volume recorded
C      COUNT            : count of (TEMP, VOLUME) pairs
C     SUMT        : sum of temperatures
```

```
C       SUMT2                   : sum of squares of temperatures
C       SUMV            : sum of volumes
C       SUMTV                   : sum of the products TEMP*VOLUME
C       TMEAN           : mean  temperature
C       VMEAN           : mean volume
C       SLOPE                   : slope of the least squares line
C       YINT            : Y - intercept of the line
        INTEGER   COUNT
        REAL TEMP, VOLUME, SUMT, SUMT2, SUMV, SUMTV, TMEAN, SLOPE, YINT
C       open the file as unit 15, set up the input and output formats, print the table heading and
initialize      C       counter and the sums to 0.
        OPEN ( UNIT=15, FILE='TEMP-VOL-FILE', STATUS='OLD')
   10   FORMAT( 4X, F4.1, T13, F4.1 )
   20   FORMAT( 1X, A11,  A10 )
   21   FORMAT(1X, F8.1,  F12.1)
        PRINT 20, 'TEMPERATURE',   'VOLUME'
        PRINT 20,  '-----------------------', '-------------'
        COUNT = 0
        SUMT = 0
        SUMT2 = 0
        SUMV = 0
        SUMTV = 0
C       while there are more data, read temperature and volumes display each in the table and
calculate the   C       necessary sums
   30   READ( UNIT=15, FMT=10, END=40 ) TEMP, VOLUME
        PRINT 21, TEMP, VOLUME
        COUNT = COUNT + 1
        SUMT = SUMT + TEMP
        SUMT2 = SUMT2 + TEMP**2
        SUMV = SUMV + VOLUME
        SUMTV = SUNTV + TEMP* VOLUME
        GO TO 30
C       finding the equation of least squares line
   40   TMEAN = SUMT / COUNT
        VMEAN = SUMV / COUNT
```

```
      SLOPE = (SUMTV - SUMT*VMEAN ) / (SUMT2-SUMT*TMEAN)
      YINT = VMEAN- SLOPE*TMEAN
      PRINT 50, SLOPE, YINT
50    FORMAT ( // 1X, 'EQUATION OF LEAST SQUARES LINE IS ', 11X, ' Y=',
     1        F5.1, ' X+ ', F5.1, 11X, 'WHERE X IS TEMPERATURE AND Y IS VOLUME
')
      CLOSE (15)
```

FIGURE 4.2

Listing of 'TEMP-VOL-FILE':

1200034203221015

1300038803221121

1400044803241425

1500051303201520

1600055503181665

1700061303191865

1800067503232080

1900072103282262

2000076803252564

2100083503272869

2200088903303186

Sample run:

TEMPERATURE	VOLUME
34.2	101.5
38.8	112.1
44.8	142.5
51.3	152.0
55.5	166.5
61.3	186.5
67.5	208.0
72.1	226.2

76.8	256.4
83.5	286.9
88.9	318.6

EQUATION OF LEAST SQUARES LINE IS

$$Y=3.8X + 39.8$$

WHERE X IS TEMPRATURE AND Y IS VOLUME

CHAPTER FIVE

ARRAYS, FUNCTIONS AND SUBROUTINES

.1 INTRODUCTIONS TO ARRAYS AND SUBSCRIPTED VARIABLES

The variables we have considered so far in this project are simbolic addresses of single memory ›cations that are used to store only one value at a tme. Such variables are usually called SIMPLE 'ARIABLES . There are situations, however when it is necessary to process a collection of values that re related in some way; for example a list of test score, a set of measurement resulting from some xperiment, or a matrix. Because processing such collection using only simple variable is extremely ;umbersome, most high-level languages includes special features for structring such data. Once such data ;tructure provided in almost every highlevel language is an ARRAY in which a fixed number of data ıtems, all of the same type are organised in a sequence and in which direct access to each item is possible by specifying its position in this sequence.

In FORTRAN, we can refer to an entire array using array variable and can access each individual element or component of the array by means of a subscripted (or indexed) variable formed by appending a subscript(or index)
enclosed in parentheses to the array variable. Thus, if A is an array variable the subscripted variables A(1), A(2), A(3), A(4) and A(5) refer to the first, second, third, fourth and fifth element repectively inthis array. This corresponds to the subscript notations A1, A2, A3, A4 and A5 commonly used in mathematics to refer to a specified element in a sequence.

For example, if 6 Age readings of the people at a polling station to cast their vote are to be processed in a programed, we might use an array to store these values. The computer must first be instructed to reserve a sequence of 6 memory locations for them. The DIMENSION STATEMENT can be used for this purpose.
For example the statement

```
DIMENSION AGE(1:6)
INTEGER AGE
```

or

```
DIMENSION AGE(6)
INTEGER AGE
```

instructs the compiler to establish an array withe the name AGE, consisting of six memory locations in which integer values are to be stored and it associate the subscripted variables

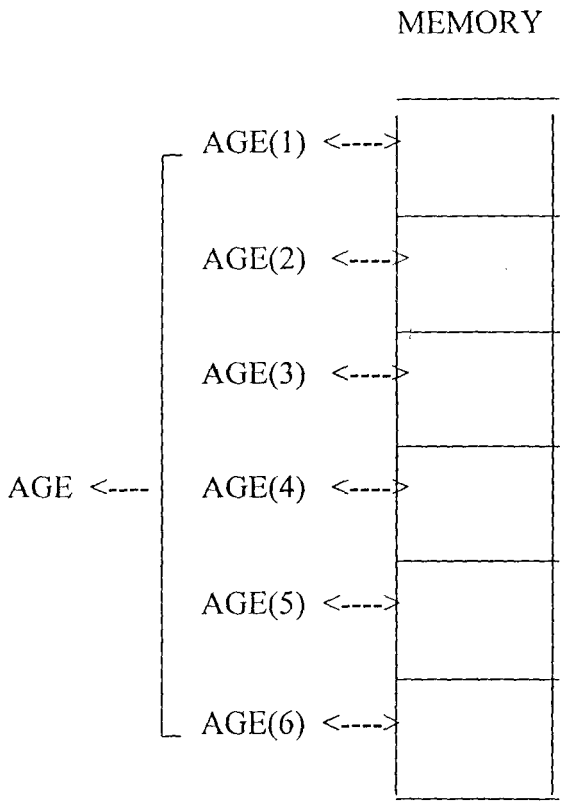
```
AGE(1)
AGE(2)
```

:

:

AGE (6)

vith these locations given below



This same array could be declared by including the dimension informatiom in the type statement itself,

```
INTEGER   AGE(1:6)
```

or

```
INTEGER   AGE( 6 )
```

Each subscripted variable AGE(1), AGE(2), AGE(3), AGE(4), AGE(5), ------- AGE(6) names an individual memory locations and hence can be use in much the same way as asimple variable can. For example, the assignment statement

```
AGE(5) = 36
```

stores the value 36 in the fifth location of the array AGE. The type of an array may be any of the FORTRAN data types. For example, an array TEXT declared by the statements

```
DIMENSION  TEXT(1:60)
CHARACTER*80  TEXT
```

or

```
DIMENSION  TEXT (60)
CHARACTER*80   TEXT
```

or simply

CHARACTER*80 TEXT

Consists of sixty character strings, each of which has length 80. Such an array could be used to store the individual line of a page of Text. TEXT(1) would refer to the first line, TEXT(2) to the second line, and in general TEXT(N) to the Nth line of text on the page.

Arrays such as AGE and TEXT involve only a single subscript and are commonly called One dimensional arrays. The name and range of subscripts of each one dimensional array in a dimension statement of the form

DIMENSION list

where 'list' is a list of array declarations of the form

array-name (l :u)

separated by commas. The pair l:u must be a pair of integer constants (or parameters) specifying the range of values for the subscript to be form from the lower limit l through the upper limit u, for example, the pair 1:7 declares that a certain subscript may be any of the integers 1, 2, 3, 4, 5, 6, 7.

Thus, as we have noted earlier, the integer array AGE that has a subscript ranging from 1 through 6 may be declared by the statements

```
DIMENSION AGE(1:6)
INTEGER AGE
```

or

```
DIMENSION AGE( 6)
INTEGER AGE
```

declares AGE to be a one dimensional integer array with a subscript ranging from 1 through 6. A single DIMENSION statement could be use to declare this array.

DIMENSION are non executable statement as they provide instructions to the compiler to reserve locations in the memory for the items in the arrays being declared. They must be placed at the beginning of the program before all executable statements.

5.1.1 INPUT / OUTPUT OF ARRAYS

The entries of a one dimensional array can be read or displayed by using any of the following three methods:

a. A DO loop containing an input / output statement

b. The array name is an input / output statement

c. An implied DO loop in an input / output statement

Each of these methods are described below:

INPUT / OUTPUT Using a DO Loop

To read or display the element of an array, one can simply put an input or output statement containing an array reference with a variable subscript within a DO loop. For example, if LENGTH is a one dimensional array and we wish to read 4 values into this array, the following statements might be used:

```
      REAL LENGTH(4)
      INTEGER I
      DO 10 I=1, 4
            READ(*,*) LENGTH(I)
10    CONTINUE
```

The DO loop containing the READ statement is equivalent to the following sequence of 4 READ statements:

```
READ(*, *) LENGTH(1)
READ(*, *) LENGTH(2)
READ(*, *) LENGTH(3)
READ(*, *) LENGTH(4)
```

Recall that each execution of a READ statement requires a new line of input data. Consequently, the 4 values to be read into the array LENGTH must be entered on 4 separate lines, one per line.

If we wish to declare a larger array and use only part of it, the statement

```
      REAL LENGTH(30)
      INTEGER  NUMLEN, I
      WRITE(*, *) 'ENTER NUMBER OF LENGTHS'
      READ(*, *) NUMLEN
      DO  10  I =1, NUMLEN
            READ(*, *)  LENGTH( I )
10    CONTINUE
```

might be used. The DO loop has the same effects as the sequence of statements

```
      READ(*, *)  LENGTH( 1 )
      READ(*, *)  LENGTH( 2 )
            :
            :
      READ(*, *)  LENGTH( NUMLEN)
```

Arrays can be displayed in a similar manner using by using a print statement within a DO loop. Thus, the first four element of the array LENGTH can be displayed with the statements

```
      DO  20  I=1, 4
            WRITE(*, *) LENGTH( I )
20    CONTINUE
```

This is equivalent to the following sequence of 4 WRITE statements below:

```
      WRITE(*, *)  LENGTH(1)
      WRITE(*, *)  LENGTH(2)
      WRITE(*, *)  LENGTH(3)
      WRITE(*, *)  LENGTH(4)
```

Because each execution of a WRITE statement causes output to begin on a new line, the 4 element of the array LENGTH are printed on four lines, one value per line. The requirement that data values must be entered on separate lines and are printed on separate lines is one of the disadvantages of the DO loop for input / output of lists.

The FORTRAN 77 program figure 5.1 below illustrate the use of DO loop for input / output of one dimensional array.

```
            PROGRAM  LEN1
      C  This program illustrate the use of  DO loops to read and displays a list of lengths.
      C NUMLEN is the number of values reads into the array length.
            INTEGER NUMLEN,  N
```

```
      REAL  LENGTH(30)
C  Entering the list of lengths
      WRITE(*, *) ' ENTER THE NUMBER OF LENGTHS '
      READ(*, *) NUMLEN
      WRITE(*, *) 'ENTER THE LENGTH VALUES, ONE PER LINE
      DO  10  N=1,  NUMLEN
            READ(*, *) LENGTH (N)
   10 CONTINUE
C  printing the list of lengths
      PRINT 20
   20 FORMAT( / 1X, 'LIST OF MEASURED LENGTHS ' / 1X,  27('=') )
      DO  40  N=1,  NUMLEN
            PRINT 30, N, LENGTH(N)
   30 FORMAT( 1X,  I3, ':', F10.1)
   40 CONTINUE
      STOP
      END
```

FIGURE 5.1

Sample of program execution:

```
ENTER THE NUMBER OF LENGTHS
4
ENTER THE LENGTH VALUES ONE PER LINE
84.6
28.0
95.3
54.1

LIST   OF   MEASURED   LENGTHS
==============================
1  :          84.6
2  :          28.0
3  :          95.3
4  :          54.1
```

INPUT / OUTPUT Using the Array Name

An input or output statement containing one array name without a subscripts is an alternative method of reading or displaying an array. The effect is the same as listing all of the array elements in the input / output statement. For example, if the array LENGTH is declared by

```
      REAL  LENGTH(4)
```

the statement

```
      READ(*, *)  LENGTH
```

is equivalent to

```
      READ(*, *)  LENGTH(1), LENGTH(2), LENGTH(3), LENGTH(4)
```

Because the READ statement is executed only once, the entries for LENGTH need not be read from separate lines. All of the entries can be on one lines or two; may be on the first line with two on the next line, or one entry ; may be on each of four lines and so on. This method can also be used with a formatted READ statement. The number of values to be read from each line of input is then determined by the corresponding format identifier.

For, the statements

```
      REAL LENGTH(4)
      READ 30,  LENGTH
30    FORMAT ( 4F6.1)
```

read the values for LENGTH(1),----------, LENGTH(4) from the first line of data. An array can be displayed in a similar manner. For example, the statements

```
      WRITE(*, 40) LENGTH
40    FORMAT( 1X, 4F8.2)
```

are equivalent to

```
      WRITE(*, 40) LENGTH(1), LENGTH(2), LENGTH(3), LENGTH(4)
40    FORMAT (1X, 4F8.2)
```

This method of reading and displaying the elements of an array is illustrated by the program labeled figure 5.2 below

```
      PROGRAM  LEN2
C  this program illustrate the use of the array name for reading and displaying list of lengths.
      REAL  LENGTH(4)
      PRINT*, 'ENTER THE LENGTH VALUES'
      READ(*, *)  LENGTH
```

```
C  displaying list of lengths
        write(*, 10)
   10   FORMAT( 1X, 'LIST OF LENGTHS: ' / 1X, 17('=') / )
        WRITE(*, 20) LENGTH
   20   FORMAT( 1X, 4F8.2)
        STOP
        END
```

FIGURE 5.2

Sample run:

```
ENTER THE LENGTH VALUES
35.2  88.5  92.1  50.7
LIST  OF  LENGTHS
================
        35.3  88.5  92.1  50.7
```

INPUT / OUTPUT Using implied DO loops

An implied DO loop in an input / output statement provide the most flexible method for reading or displaying the elements of the array. It allows the programmer to specify that only a portion of the array be transmitted and to specify the arrangement of the values to be read or displayed.

An IMPLIED DO LOOP has the form

(i/o-list, control-variable = initial-value, limit)

or

(i/o-list, control-variable = initial-value, limit, step-size)

The effect of an implied DO loop is exactly that of a DO loop - as if the left parenthesis were a DO, with indexing information immediately before the match right parenthesis and the i/o - list consisting the body of the DO loop. The control-variable, the initial-value, the limit and the step-size are as in a DO statement. The i/o-list may, in general, be a list of variables (subscripted or simple), constants, arithmetic's expressions, or other implied DO loops, separated by commas with a comma at the end of the list.

An implied DO loop may be used in a READ, PRINT, or WRITE statement (or in a Data statement, as described in the next section). For example, if the array LENGTH is declared by

```
REAL  LENGTH(10)
```

and the first four entries are to be read, we could use the statement

```
READ(*,*) ( LENGTH(N), N=1, 4)
```

which is equivalent to

```
READ(*, *) LENGTH(1), LENGTH(2), LENGTH(3), LENGTH(4)
```

In a similar manner, we can display the entries:

```
WRITE(*,*) LENGTH(1), LENGTH(2), LENGTH(3), LENGTH(4)
```

Let us now have an example to illustrate the use of implied DO loops to input and output the elements of a list as shown in figure 5.3 below

```
      PROGRAM  LEN3
C  The use of implied Doloop for input and output of an array.
C     INTEGER N, I
      REAL  LENGTH(10)
C  Entering the list of measured lengths
      PRINT*, 'ENTER THE NUMBER OF LENGTHS'
      READ(*,*) N
      PRINT*, 'ENTER THE VALUES AS MANY  PER LINE
      READ *, ( LENGTH(I), I=1, N)
C  Displaying the list of lengths
      PRINT 10, N
 10   FORMAT( 1X, 'LIST OF', I3, 'LENGTHS:' /1X,  18('=') )
      PRINT 20, ( LENGTH(I),  I=1, N)
 20   FORMAT( 1X, F5.1)
      STOP
      END
```

FIGURE 5.3

5.1.2 INTRODUCTION TO MULTIPLE DIMENSIONAL ARRAYS AND MULTIPLE SUBSCRIPTED VARIABLES

In the preceding section, we discuss one dimensional arrays and used them to process lists of data. FORTRAN allows arrays of more than one dimension and that two dimension can be arranged in rows and columns. Similarly, a three dimensional array becomes appropriate when the data can be arranged in rows, columns and ranks. When there are several characteristics associated with the data, still higher dimension corresponding to one of those characteristics.

The general form of **array declaration** is

array-name (I1:J1, I2:J2,----------,IN,JN)

Where the number N of dimensions is at most seven, and each pair Ii:Ji must be a pair of integer constants or parameters specifying the range of values for the ith subscript to be from Ii through Ji. There must be one such array declaration for each array used in a program, and these declarations may appear in DIMENSION OR TYPE statements.

For example, consider the temperature readings that can be arranged in a table having four rows and three columns:

TIME	SITE / LOCATION		
	A	B	C
1	62.8	67.3	61.4
2	65.3	68.4	66.2
3	69.8	71.0	68.5
4	66.5	69.8	67.6

In this table, the three temperature readings at time 1 are in the front row, the three temperatures at time 2 are in the second row, and so on. These 12 data items can be conveniently stored in a two-dimensional array. the array declaration

```
DIMENSION TEMP( 1:4, 1:3)
REAL TEMP
```

or

```
DIMENSION TEMP( 4, 3 )
REAL TEMP
```

reserves 12 memory locations for these data items. The dimensioning information can also be included in the type statement:

```
REAL TEMP( 1:4, 1:3 )
```

or

```
REAL TEMP( 4, 3)
```

The doubly subscripted variable TEMP (3.2) then refers to the entry in the third row and second column of the table, that is, to the temperature 71.0 recorded at time 3 SITE B.

In array processing, a one dimensional array is usually processed in their natural order in which the array elements occur in sequence. Two dimensional arrays organised as a table consisting rows and columns leads to two natural orders for processing the entries in row wise and columnwise form. Rowwise processing means that the array elements in the first row are processed first, then those in the second rows, and so on. In columnwise processing, the entries in the first column are processed first, then those in the second column and so on.

In section 5.1.1, we discussed the three methods for input and output of one dimensional arrays:

a. Use an input / output statement within a DO loop.

b. Use the array name in an input / output statement.

c. Use an implied DO loop in an input / output statement.

Each of these three techniques can also be used for the input and output of multidimensional arrays.

INPUT / OUTPUT Using a DO loop

Here, the input or output statement is placed within a set of nested DO loops, each of whose indices controls one of the subscripts of the array. For example, to read the temperature values in the 4x3 real arrays TEMP declared by

```
REAL TEMP(4, 3)
```

so that it has the value

```
62.8   67.3   61.4
65.3   68.4   66.2
69.8   71.0   68.5
66.5   69.8   67.6
```

We use the statements

```
      DO 20 TIME=1, 4
            DO 10 SITE=1, 3
                  READ*, TEMP(TIME, SITE)
10          CONTINUE
20    CONTINUE
```

The data will have to be entered on 12 separate lines one per line 6.28, 65.3, ..., 68.5, 67.6.

Because the data values must appear on separate lines, one value per line, this method is cumbersome for large arrays. A similar problem also occurs with output. Since each execution of a print or write statement within nested DO loops such as the one below causes output to begin on a new line.

```
      DO 20 TIME=1, 4
            DO 10 SITE=1, 3
                  PRINT*, TEMP(TIME, SITE )
10          CONTINUE
20          CONTINUE
```

INPUT / OUTPUT Using the ARRAY NAME

Here, the total number of entries as specified in the array declaration must be read or displayed. Therefore, it is not possible to read or display only part of an array using this method. For example, the statement:

```
INTEGER MAT(3, 4)
READ(*, *) MAT
```

causes values to be read into the Array MAT columnwise. Thus, for input data

```
15, 23, 18, 75, 23, 16
28, 72, 81, 42, 35, 20
```

the value assigned to MAT IS

```
15  75  28  42
23  23  72  35
18  16  81  20
```

the output statement

```
PRINT '(1X, 4I5/ )', MAT
```

displays the elements in columnwise produces the output

```
  15  23   18  75
I----------------------------------------
  23  16   28  72
I----------------------------------------
  81  42   35  20
I----------------------------------------
```

<u>INPUT / OUTPUT Using Implied DO loops</u>

An implied DO loop already discussed in section 5.1.1, has the form

(i/o-list, control-variable = initial-value, limit)

or

(i/o-list, control-variable = initial-value, limit, step-size)

The fact that the input / output list may contain other implied DO loops makes it possible to use implied DO loops to read or display multidimensional arrays.

For example, the statement

```
READ*, ( (MAT(ROW,COL), COL=1, 4), ROW=1,3)
```

is equivalent to the statement

```
READ*, (  MAT(ROW, 1), MAT(ROW, 2), MAT(ROW, 2),  MAT(ROW, 3),
1              MAT(ROW, 4) , ROW=1, 3  )
```

which has the same effect as

```
READ*, MAT(1, 1), MAT(1, 2), MAT(1, 3), MAT(1, 4)
```

1 MAT(2,1), MAT(2,2), MAT(2, 3), MAT(2, 4)

MAT(3,1), MAT(3, 2), MAT(3,3), MAT(3,4)

and thus reads the entries of the array MAT in rowwise order. Note that because the READ statement is encountered only once, the data values to be read can be entered all on the same line, or with four values on each of three lines or with seven values on one line, four on the next, and one on another line, and so on.

5.2 LIBRARY FUNCTIONS AND STATEMENT FUNCTIONS

The FORTRAN language provides many **intrinsic or library functions.** These library functions include not only the numeric functions but as well as character and logical functions. Table 5.1 below gives a complete list of the standard FORTRAN library functions.

FORTRAN FUNCTION	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE
ABS(X)	Absolute value of X	I, R, DP, C	Same as argument
ACOS(X)	Arccosine (in radians) of x	R, DP	Same as argument
AIMAG(Z)	Imaginary part of Z	C	R
AINT(X)	Value resulting from truncation of fractional part of X	R, DP	same as argument
ANINT(X)	X rounded to the nearest integer INT(X+5) if x > 0 INT(X-5) if x < 0	R, DP	same as argument
ASIN(X)	Arcsine (in radians) of X	R, DP	same as argument
ATAN(X)	Arctangent (in radians) of X	R, DP	same as argument
ATAN2(X,Y)	Arctangent (in radians) of X /Y	R, DP	same as argument
CHAR(I)	Character in ith position of the collating sequence	I	character
CMPLX(X,Y)	The complex number (X, Y)	I, R, DP	C
CMPLX(X)	The complex number (X,0) if X is type I, R, or DP; the complex number X if X is type C	I, R, DP, C	C
CONJG(Z)	Conjugate of Z	C	C
COS(X)	Cosine of X (X in radian)	R, DP, C	same as argument
COSH(X)	Hyperbolic cosine of X	R, DP	same as argument
DBLE(X)	conversion of X to double precision	I, R, DP, C	DP
DIM(X,Y)	X-Y if X > Y 0 if X < y	I, R, DP	same as argument
DPROD(X,Y)	Double precision product of X and Y	R	DP
EXP(X)	Exponential function of X	R, DP, C	same as argument
ICHAR(C)	Position of C in the collating sequence	Character	I
INDEX(C1, C2)	Location of substring C2 in string C1	Character	I
INT(X)	Conversion of X to integer	I, R, DP, C	I

	type; sign of X or real part of X times the greatest integer < ABS(X)		
LEN(C)	Length of character string C	Character	I
LGE(C1,C2) LGT(C1,C2) LLE(C1,C2) LLT(C1,C2)	Value is true if and only if, C1 is lexically greater than or egual to C2, greater than C2, less than or equal to C2, less than C2, respectively	Character	Logical
LOG(X)	Natural logarithm of X	R, DP, C	same as argument
LOG10(X)	Common (base10) logarithm of X	R, DP	same as argument
MAX(X1,--Xn)	Maximun of X1,---,Xn	I, R, DP	same as argument
MIN(X1,--,Xn)	Minimum of X1,--,Xn	I, R, DP	same as argument
MOD(X, Y)	X (MOD Y); X - INT(X/Y)*Y	I, R, DP	same as arguments
NINT(X)	X rounded to the nearest integer	R, DP	I
REAL(X)	Conversion of X to real type	I, R, DP, C	R
SIGN(X,Y)	Transfer of sign: ABS(X) if Y>O -ABS(X) if y<0	C, R, DP	same as arguments
SIN(X)	Sine of X (in radians)	R, DP, C	same as arguments
SINH(X)	Hyperbolic sine of X	R, DP	same as argument
SQRT(X)	Square root of X	R, DP, C	same as argument
TAN(X)	Tangent of X (in radians)	R, DP	same as argument
TANH(X)	Hyperbolic tangent of X	R, DP	same as argument

Note that
I = integer, R = real, DP = double precision, C = complex.
Types of arguments in a given function reference must be the same.

As we have seen, any of these functions may be used to calculate some values in an expression by giving its name followed by the actual arguments to which it is to be applied, enclosed in parentheses. For example, if ALPHA, NUM1, NUM2, SMALL, BETA and X are declared by

```
INTEGER ALPHA, NUM1, NUM2, SMALL
REAL BETA, X
```

then the statements

```
PRINT*, ABS(X)
ALPHA=NINT( 100.0*BETA ) / 100
SMALL=MIN( 0, NUM1,     NUM2)
```

displays the absolute value of X, assign to ALPHA the value of BETA rounded to the nearest hundredth, and assign to SMALL the smallest of the three integers 0, NUM1 AND NUM2.

In some programs it may be convenient for user to define an additional functions. Such user-defined functions are possible in FORTRAN, and once defined, they are used in some way as library functions. The simplest user-defined functions are the **statement functions.**

A statement function must be defined by a single statement of the form

name (argument-list) = expression

where

argument-list is a list (possibly empty) of variables separated by commas. The expression may contain constants, variables, formulas or references to library functions, to previously defined statement functions, or to functions defined by subprograms, but not references to the function being defined. Such statements must appear

in the program unit in which the functions are referenced, and they must be placed after the specification statements and before all executable statements.

In a statement defining a function, the function name may be any legal FORTRAN name. It must differ from other function and variable names in the same program unit. The type of the value of the function is determined by the type of its name. The variables in the argument list are called **formal arguments** and indicate the number, order and type of arguments of the function.

For example,

```
REAL A, B, HYPO
HYPO(A,B)=SQRT(A**2+B**2)
```

define a real valued function of two real arguments. The arguments in a function reference are called **actual_ arguments.** When a function is referenced, the values of these actual arguments becomes the values of the corresponding formal arguments and are used in computing the value of the function. For example, if X, Y and Z have been declared to be real variables and the values of X and Y are 3.0 and 4.0, respectively, then in the statement

```
Z= HYPO(X,Y)
```

The values of the actual arguments X and Y becomes the values of the formal arguments A and B respectively. the value of the function

3.0 +4.0 = 5.0

is then computed and assigned to Z.

Because of this association between actual and formal arguments, the number and type of the actual arguments must agree with the number and type of the formal arguments.

To illustrate the use of the statement functions, we shall solve the problem of approximating the integral of a function f(x) = x + 1 over the interval [A, B] using the rectangle method and a statement function to define a function to be integrated.

```
      PROGRAM   AREA
C     This program is written in FORTRAN 77 to illustrate the use of statement function to
```

```
C  to approximate the integral of a function over the interval [A, B]  using the rectangle method
C with altitude chosen at the mid points of the subintervals. variables used are :
C  A, B: the endpoints of the interval of integration
C  N    : the number of subintervals used.
C  I    :  counter
C  DELX  :  The length of the subintervals.
C  X  :  The midpoint of one of the intervals.
C  F  :  The function being integrated
C  SUM   : The approximating sum
      REAL   F, A, B, X, DELX, SUM
      INTEGER  N, I
      F(X) = X**2 + 1
      PRINT*, 'ENTER THE INTER ENDPOINTS AND THE NUMBER
     1 OF SUBINTERVALS'
      READ*, A, B, N
      DELX =( B-A ) / N
C  Initialize the approximating sum and set X equal to the midpoint of the first subinterval
      SUM = 0
      X = A + DELX / 2
C  Now  compute and display the sum
      DO  10  I=1,  N
              SUM = SUM + F(X)
              X = X + DELX
   10 CONTINUE
      SUM = DELX * SUM
      PRINT*, 'APPROXIMATE  VALUE USING' , N, 'SUBINTERVALS IS', SUM
      STOP
      END
```

FIGURE 5. 4

Sample of program execution

ENTER THE INTERVAL ENDPOINTS AND THE NUMBER OF SUBINTERVALS
0,1,10
APPROXIMATE VALUE USING 10 SUBINTERVALS IS 1.33250

ENTER THE INTERVAL ENDPOINTS AND THE NUMBER OF SUBINTERVAL
0,1, 100
APPROXIMATE VALUE USING 100 SUBINTERVALS IS 1.33332

5.3 FUNCTIONS SUBPROGRAMS

A statement function consists of a single statement and thus can be used only to define a function whose definition can be given by a single formula . Also, a statement function must be defined within the program unit in which it is referenced. In contrast, a **function subprogram** consists of several statements and thus makes possible the definition of a function whose value cannot be specified by a simple expression. Moreover, subprograms are separate program unit. Consequently, a subprogram can be prepared and saved in a user's library, it may be used in any program, by simply attaching it to that program.

The syntax of a function subprogram is similar to that of FORTRAN (main) program:

FUNCTION statement

Declaration part

subprogram statements

END

The first statement must be **FUNCTION statement** of the form

FUNCTION name (argument-list)

Here, **name** is the name of the function and must follow the usual naming rules, with the type of the function name determining the type of the value of this function; **argument-list** is a list (possibly empty) of variables separated by commas. These variables are **formal arguments** and indicate and indicate the number, order and type of the arguments of the function like the main program, a subprogram should also include opening documentation to describe briefly what the subprogram does, what its argument and other variables represents, and other information's that explains the subprograms. These opening documentation or comments and the rest of the subprograms must conform to the usual rules governing FORTRAN programs. For example, DIMENSION, type and DATA statements must precede all of the executable statements in the subprograms.

At least one of the executable statements should assign a value to the function. Normally, this is done with an assignment statement of the form

name = expression

The expression may be an expression involving constants, the formal argument of the function, other variables already assigned values in this subprogram, as well as references to other functions.

The last statement of the of the subprogram must be

END

The value of the function is returned to the program unit that references the function when the END statement or a RETURN statement of the form

RETURN

is executed.

As an example, suppose we wish to use the function

$$f(x, y) = \begin{cases} x + 1 & \text{if } x < y \\ x + y & \text{if } x > y \end{cases}$$

The following function subprogram defines this function

```
FUNCTION F(X, Y, N)
REAL F, X, Y
INTEGER N
IF ( X .LT. Y ) THEN
        F = X + 1
ELSE
        F = X**N + Y**N
END IF
END
```

This function F can then be referenced by such statements as

```
W = F( A, B+3.0, 2 )

Z = F(TOP(I), SIN(A), K )

IF ( 1.1, BETA, 2) .LT. EPS ) DONE = .TRUE.
```

provided the types of actual argument s used in these statements match those of the formal arguments X, Y and N.

5.4 SUBROUTINE SUBPROGRAMS

Subroutine subprograms, like function subprogram are program units designed to perform a particular task. They differ from function subprogram, however, in the following respects:

1. Function are designed to return a single value to the program unit that references then. Subroutine often return more than one value, or they may return no value at all but simply perform some tasks such as displaying a list of instructions to the user.
2. Functions return values via function names; subroutine return values via arguments.
3. A function referenced by using its name in an expression, whereas a subroutine is referenced by **CALL** statement.

The syntax of subroutine subprogram is similar to that of function subprograms and thus to that of FORTRAN (main) programs:

SUBROUTINE statement

Declaration part

Subprogram statement

END

Subroutine subprograms must begin with a SUBROUTINE statement of the form

SUBROUTINE name (arguments-list)

Here, name represent the name given to the subroutine and may be any legal FORTRAN name, but no type is associated with a the name of a subroutine; argument-list is a list (possibly empty) of variables separated by commas.

These variables are the **formal arguments** and indicate the number, order, and type of values transferred to and returned from the subroutine. If there are no formal arguments, the parentheses in the SUBROUTINE statement may be omitted.

A subroutine is referenced by a CALL statement of the form

CALL name (argument-list)

Here, **name** is the name of the subroutine being called, and argument-list contains variables, constants, or expressions that are the **actual arguments.** The number of actual argument must equal the number of formal argument, and each actual argument must agree in type with the corresponding formal argument. If there are no actual argument, the parentheses in the CALL statement may be omitted.

A simple illustration, suppose we wish to develop a subroutine that accept from the main program a month number, a day number, and a year number, and displays them in the form

MM / DD / YY

For example, the values 8, 14, 1941 as 08 / 14 / 41

and the values 9, 3, 1905 as 09 / 03 / 05

This subroutine must have three formal arguments, each of the integer type representing the number of the month, day and year respectively. Thus, an appropriate SUBROUTINE statement is

SUBROUTINE DATE (MONTH, DAY, YEAR)

where MONTH, DAY and YEAR must be declared of type INTEGER in the declaration part of this subroutine.

Only the last two digits of the year are to be displayed, and these can be obtained using the statement

YEAR = MOD (YEAR, 100)

For example, if the value passed to YEAR is 1941, this statement assigns the value 41 to YEAR, which can then be displayed. If the value passed to YEAR is 1905, this statement assigns the value 5 to YEAR, which we wish to display as 05. Similarly, when the month and day numbers are single digits, we wish to display them with a leading ZERO. Formatting the output using a format descriptor I2.2 achieves the desired result:

This subprogram is referenced in the program in figure 5.5 below by the CALL statement

CALL DATE(BMONTH, BDAY, BYEAR)

This statement causes the values of the actual arguments BMONTHS, BDAY and BYEAR to be passed to the formal parameters MONTH, DAY and YEAR, respectively, and initiate execution of the

subroutine. When the end of the subroutine is reached, execution resume with the statement following this CALL statement in the main program.

```
      PROGRAM  DATE
C  program demonstrating the use of a subroutine subprogram  DATE
C  to display a given date in the form MM / DD / YY variables
C  used are :
C BMONTH :  birth month
C  BDAY : birth day
C  BYEAR  :  birth year
      INTEGER BMONTH, BDAY, BYEAR
      PRINT*, ' ENTER BIRTH MONTH, DAY AND YEAR
     1    (ALL O"S TO STOP) '
      READ(*, *) BMONTH, BDAY, BYEAR
C  While there are more data, do the following:
   10 IF (BMONTH .GT.0  )  THEN
            CALL  DATE ( BMONTH, BDAY, BYEAR )
            PRINT*,
            PRINT*, 'ENTER BIRTH MONTH,  DAY, AND
     1            YEAR   ( ALL 0"s  TO  STOP )
            READ(*, *)  BMONTH, BDAY, BYEAR
            GO TO 10
      END IF
      STOP
      END


C DATE
C  subroutine to displaying a data in the form MM / DD / YY.
C  The MONTH, DAY and  YEAR number are passed as arguments.
      SUBROUTINE  DATE ( MONTH, DAY, YEAR )
      INTEGER  MONTH, DAY, YEAR
      YEAR = MOD ( YEAR, 100 )
      WRITE(*, 10) MONTH, DAY, YEAR
   10 FORMAT( 1X, 2( I2.2, '/', I2.2 )
      END
```

FIGURE 5.5

Sample of program dates execution

ENTER BIRTH MONTH, DAY AND YEAR (ALL O's TO STOP)

8 14 1941

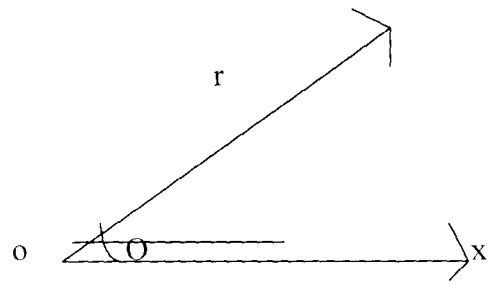
08 / 14 / 41

ENTER BIRTH MONTH, DAY AND YEAR (ALL O's TO STOP)

9 3 1905

09/O3/ 05

In this example, the subroutine DATE does not calculate and return new values to the main program; it only displayed the information passed to it. As an illustration of a subroutine that does return values, consider the problem of converting polar coordinates (r, O) of a point p to a rectangular coordinates (x, y) . The first polar coordinate r is the distance r from the origin to p, and the second polar coordinate O is the angle from the positive x axis to the ray joining the origin with p.



The formula that relate the polar coordinates to the rectangular coordinate for a point are

x = r cosO

y = r sin O

Because subprogram that performs this conversion must return two values, it is natural to use a subroutine subprogram like the following to accomplish this:

```
C  CONVER
C  subroutine to convert polar coordinates (R, THETA)
C  to rectangular coordinates (x, y)
      SUBROUTINE CONVER ( R, THETA, X, Y)
      REAL R, THETA, X, Y
      X = R * COS (THETA)
      Y = R * SIN (THETA)
      END
```

This subroutine can be referenced by the CALL statement

```
      CALL CONVER ( RCOORD, TCOORD, XCOORD, YCOORD )
```

where RCOORD, TCOORD, XCOORD and YCOORD are real variables. When this CALL statements is executed, the actual arguments RCOORD, TCOORD, XCOORD and YCOORD are associated with the final arguments R, THETA, X and Y respectively, so that corresponding arguments have the same values.

Actual parameters	memory locations	formal parameters
RCOORD	1.0	R
TCOORD	1.57	THETA

XCOORD	?	X
YCOORD	?	Y

These values are used to calculate the rectangular r coordinates X and Y, and these values are then the values of the corresponding actual arguments XCOORD and YCOORD.

Actual parameters	Memory locations	Formal parameters
RCOORD	1.0	R
TCOORD	1.57	THETA
XCOORD	7.967636E-04	X
YCOORD	1.00000	Y

The program in figure 5.6 below reads values for RCOORD and TCOORD, calls the subroutine CONVER to calculate the corresponding rectangular coordinates, and then displays these coordinates as shown below.

```
      PROGRAM   POLAR
C This program accepts the polar coordinates of a point and displays the corresponding
C rectangular coordinates. The subroutine CONVER is used to effect the conversion .
C  Variables used here are as follows:
C  RCOORD, TCOORD: polar coordinates of a point
C  XCOORD, YCOORD : rectangular coordinates of a point
      REAL  RCOORD, TCOORD, XCOORD, YCOORD
C while there is more data do the following
   10       WRITE( *,*) 'ENTER POLAR COORDINATES IN RADIANS'
      READ (*,*,  END=20) RCOORD, TCOORD
      CALL CONVER ( RCOORD, TCOORD, XCOORD, YCOORD )
      PRINT*,  'RECTANGULAR COORDINATES:'
      PRINT*, XCOORD, YCOORD
      PRINT*
      GO TO 10
20    END
C  CONVER
```

```
C  Subroutine to convert polar coordinates (R, THETA)
C  to rectangular coordinates (X, Y)
      SUBROUTINE CONVER (R, THETA, X, Y )
      REAL R, THETA, X, Y
      X = R* COS(THETA)
      Y= R* SIN(THETA)
      END
```

FIGURE 5.6

Sample of program execution

```
ENTER POLAR COORDINATES IN RADIANS
1.0,  1.57
RECTANGULAR COORDINATES:
      7.967636E-04          1.00000
ENTER POLAR COORDINATES IN RADIANS
4.0,  3.14159
RECTANGULAR COORDINATES:
      -4.00000              1.498028E-05

                <----------------------(end of data signaled)
```

5.5 PROCEDURE FOR USING WATFOR77 COMPILER

5.5.1 INTRODUCTION

The WATFOR77 FORTRAN compiler is a popular version of FORTRAN compiler for personal computers specially designed for executing programs written in standard FORTRAN 77 language. However, it is also capable of executing programs written in other earlier versions of FORTRAN language on or before FORTRAN 77.

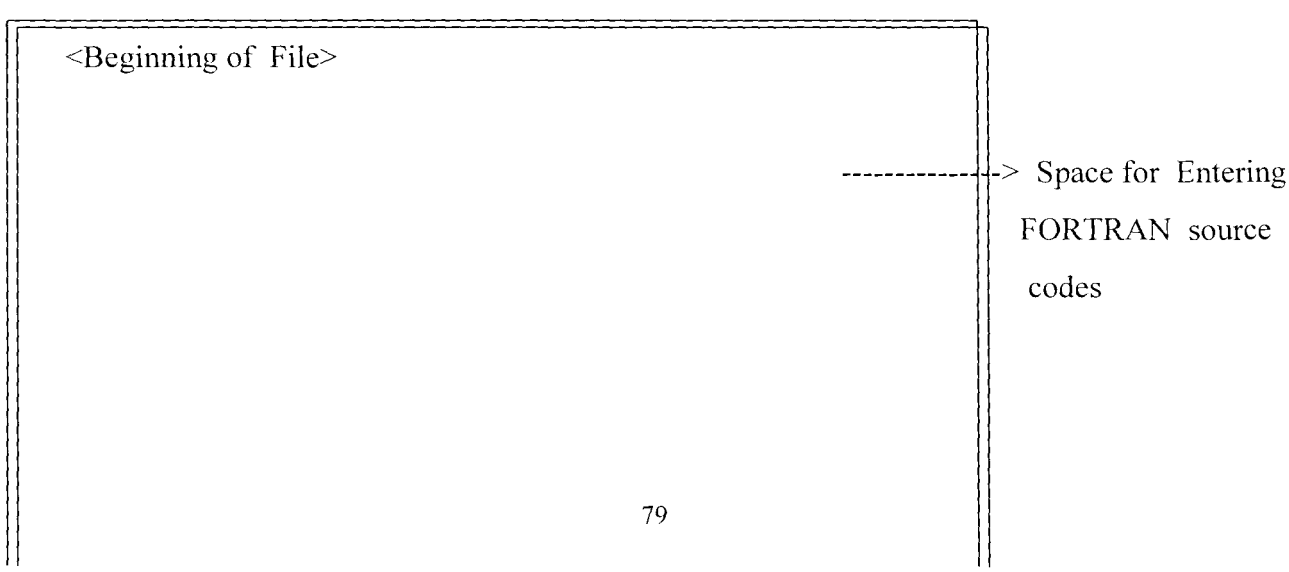
WATFOR77 fully integrates the compiler, the editor as well as the execution debugger which is made interactive to the users. The compile time is also optimized since the source code is compiled directly into the memory, immediately linked and executed, thus, eliminating the time consuming independent step of compilation, linkage and execution. This is evident for the fact that, one does not have to leave the WATFOR77 environment to edit a FORTRAN source code. It also provides excellent diagnostic, clear, concise and readable error messages in plain English rather than numeric error codes given by other FORTRAN compilers.

The WATFOR77 compiler appears to be more efficient and easier to use especially for the first time programmers in FORTRAN language. This is because of its unparalleled comprehensive diagnostic messages and thorough error checking capabilities. The interactive debugger provides options like program trace, break-point, display of variable contents to facilitate debugging activity. It also flags FORTRAN extensions which are not FORTRAN standards.

In order to LOAD WATFOR77 compiler into our system, we must have to be in the directory that contains WATFOR77 compiler. While in this directory, we must type

WATFOR77 < ENTER >

This automatically starts the WATFOR77 compiler. The WATFOR77.EXE is the executable file in this directory.



<End of File>

WATFOR77 Version 3.0 copyright WATCOM SYSTEM INC.

Space for WATFOR77

----------> system commands

DISPLAY OF MENU COMMANDS

FIGURE 5.7 THE STRUCTURE OF WATFOR77 SCREEN IN A DISPLAY OF PERSONAL COMPUTER SCREEN

.5.2 WATFOR77 SYSTEM COMMANDS

These are the **commands** that help us in executing our FORTRAN programs but they do not take part directly during the program execution. These commands send instructions directly into the WATFOR77 compiler. They are normally entered at the **WATFOR77 COMMAND LINE** through the KEY BOARD of the personal computer.These commands include the followings:

1. **EDIT**:- This command enables us to create a **new file** for starting a new FORTRAN program. It also enables us to LOAD and run a FORTRAN program stored in any directory on the disk of the micro

computer. EDIT command is the first command to be entered at the WATFOR77 command line immediately the compiler Is LOADED. The structure of these command is

EDIT filename <RETURN>

where,

EDIT is the command

filename is the name of the program file. All program written in FORTRAN to be run by this computer must have **.FOR** as i t extension name.

<RETURN> is the ENTER KEY to be pressed at the end of typing this command.

2. **RUN**:- This command when entered at the command line enables us to COMPILE, LINK, PARSE and EXECUTE our FORTRAN program. The structure of this command is

RUN < RETURN>

where,

RUN is the keyword and

RETURN is the ENTER KEY to be pressed at the end of typing this command.

While this is done, the program listing will be STORED in the present directory with an extension file name **.LST** . We can access our filename **.LST** to see the nature and type of errors before

EDITING the source code FILENAME.FOR in this directory. The listing in file with an extension name **.LST** include the storage used, execution and compile time.

3. **PUT**:-This command is used to save our FORTRAN program in the display of the personal computer screen. It also enable us to write filename in to our program source code, rename the program filename and automatically save such program in to the disk. The structure of this command is

PUT filename <RETURN>

where,

PUT is the keyword.

filename is the name of the file to be written into the program on the screen and to be saved with such filename.

<RETURN> is the ENTER KEY to be pressed after typing this command at the WATFOR77 command line.

4. **EXIT** :-This command enable us to exit or leave the present program filename we are working with in order to give way to enter and run new programs. The structure of this command is

EDIT <RETURN>

Where,

EXIT is the keyword and

<RETURN> is the ENTER KEY to be pressed immediately after typing this command.

5. **QUIT**:-This command enables us to save our existing FORTRAN program file in the display of personal computer screen as well immediately takes us out of the compiler to the DOS (Disk Operating System) prompt. The structure of this command is

QUIT <RETURN>

where,

QUIT is the key word and <RETURN> is the ENTER KEY to be pressed immediately after typing this command.

5.5.3 WATFOR77 MENU COMMANDS

The menu commands usually displayed at the bottom of the screen where n the WATFOR77 compiler is loaded are the commands allocated to the **PROGRAMMABLE FUNCTION KEYS** and **SHIFT + FUNCTION KEYS** by the WATFOR77 compiler and this command is only recognisable **within** this WAFOR77 compiler environment only.

Thus, pressing any of these FUNCTION KEYS or SHIFT +FUNCTION KEYS enables the command associated with this key by the WATFOR77 compiler to be executed immediately. These keys along with thier associated functions are outlined below:

A. **F1 KEY** :- Pressing the F1 key in the WATFOR77 environment will take us **one page of the screen up** and subsequent pressing of f1 key will also do the same thing.

B. **F2 KEY**:- Pressing the F2 key in the WATFOR77 environment will takes us one page of the screen down and subsequent pressing of this key will perform this same command.

C. **F3 KEY**:- This key also tagged with line up is meant to take us **one line of the screen immediately upward** from the current line whenever it is pressed.

D. **F4 KEY**:- This key when pressed takes us **one line of the screen down** immediately after the current line. Subsequent pressing will also do the same.

E. **F5 KEY**:- This key when invoked **insert an empty line space** between the current line and the line immediately after the current line. The key is particularly important because line must be inserted between the <Beginning of the file> and <End of the file> before entering our FORTRAN program on the screen through the key board of the personal computer.

F. **F6 KEY**:- This key when pressed **delete the current line** where the cursor is located.

G. **F7 KEY**:- This key when pressed enable us to **select / deselect** a **line** or some line with the help of the down arrow key (!) complete with F8 KEY.

H. **F8 KEY**:- This key is used to complete line the **select / deselect**. Otherwise it is used to cut part of the program instruction line.

I. **F9 KEY**:- This key is called **screen command** key. Pressing this key when we are in FORTRAN program source code file takes us to the COMMAND LINE. Also pressing this key when at the command line takes to the program source code area of the screen of the personal computer.

J. **F10 KEY**:- Called the help key of the WATFOR77 compiler. Pressing this key **display menu commands** fully on the screen. We have to press the **<RETURN>** key to take us out of the HELP screen mode.

K. **SHIFT + F1**:- Pressing these keys takes one **half page up the screen.**

L. **SHIFT + F2**:- Pressing the SHIFT KEY and simultaneously pressing the F2 key take one **half of the page down**

the screen.

M. **SHIFT + F3**:- The SHIFT KEY together with the F3 key enables one to change the background color of the screen of the monitor.

N. **SHIFT + F4**:- The SHIFT KEY and simultaneously pressing the F4 key enable one to change the foreground color of the screen of the monitor.

O. **SHIFT + F5**:- This key is associated with the command to perform **line split**. Whenever this key is pressed, the current line is split into two starting from where the cursor is situated.

P. **SHIFT + F6**:- This key perform the command to **undelete a line.** When the F6 key is pressed, the current line is automatically deleted. So to undelete this line one can simply press the SHIFT key together with the F6 key to perform this function.

Q. **SHIFT + F7**:- This key performs the command to **join line** together on the screen.

R. **SHIFT + F8**:- This key performs the command to **paste** the selected source code on the location it is to be situated.

S. **SHIFT + F9**:- This key is used to **fill** unwanted spaces that may dominate the current line of the program source code.

T. **SHIFT + F10**:- This key is used to **edit** the FORTRAN program displayed on the screen of the personal computer.

THE HELP SCREEN DISPLAY

F1 page up	F6 line up	F11 1/2 page up	F16 line undelete
F2 page down	F7 select/deselect	F12 1/2 page down	F17 line join
F3 line up	F8 cut	F13 fore ground	F18 paste
F4 line down	F9 screen command	F14 back ground	F19 fill
F5 line insert	F10 help	F15 line split	F20 edit
FUNCTION KEY		SHIFT FUNCTION KEY	

FIGURE 5.8

REFERENCES

1. Microsoft Quick BASIC Manual Version 4.5 Yr of Publication
2. Sommerville, I, (1982) Software Engineering (International Computer Science series)

REFERENCES

1. Microsoft Quick BASIC Manual Version 4.5
2. Sommerville, I, (1982) Software Engineering (International Computer Science series) Addison - Wesley Publishers Limited, London.
3. Groft G.M., (1983) Computer Studies a practical Approach, Prentice- Hall Inc., London.
4. Hammond R. H.; Rogers W.B. and Critten J.B., (1987), Introduction to FORTRAN 77 and the Personal Computer, McGRAW-HILL Book company.
5. Fatunla S.O., (1993), Fundamentals of FORTRAN Programming , ADA+JANE press Nigeria LTD.
6. Lipschutz S. and Poe A., (1982), Shaum's Outline of theory and problems of PROGRAMMING WITH FORTRAN, MicGraw-Hill Book Co., Singapore.
7. Mynatt B.T., (1990), Software Engineering with Student project Guidance, Prentice- Hill Inc., London.