CHAPTER 1

NUMERICAL ALGORITHMS AND ERRORS

SUPPLEMENT No. 1c:

FORTRAN 95 TUTORIAL

prepared for

NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS With Pseudocodes

By Zekeriya ALTAÇ

October 2024

Supplement No. 1c: The Fortran 95 Tutorial

LEARNING OBJECTIVES

The objective of this Fortran 95 programming tutorial is to
- present a short summary of the basics of Fortran 95 language;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs;
- explain how to prepare functions or subprograms.

The textbook "*Numerical Methods for Scientists and Engineers: With Pseudocodes*" focuses on implementing the methods in science and engineering applications. Supplemental course materials and resources, including C/C++, Fortran, Visual Basic, Python, Matlab®, and Mathematica®, are provided to assist the instructors in their teaching activities outside the class.

The aim of this short tutorial is to enable students to acquire the knowledge and skills to convert the pseudocodes given in the text into running Fortran 95 programming language. It is not intended to be a "complete language reference document." The author assumes that the reader is familiar with programming concepts in general and may also be familiar with the Fortran 95 programming language at the elementary level. In this regard, this tutorial illustrates the conversion and implementation of pseudocode statements (such as formatted/unformatted input/output statements, loops, accumulators, control and conditional constructs, creating and using functions, and subprograms, etc.) to the Fortran 95 programming language.

1 FORTRAN PROGRAM STRUCTURE

A fortran program consists of three sections: *declarations*, *execution*, and *termination* sections.

1.1 DECLARATION SECTION

The declaration section is located at the beginning of the main as well as subprograms. It consists of the nonexecutable statements that define the name of the program and the number and types of variables referenced in the program.

A fortran program starts with the `PROGRAM` statement, followed by the program name, which is referred to as *program header*. Fortran 95 program and variable names (identifiers) can be up to 31 characters long and contain any combination of alphabetic characters, digits, and the underscore (_) character, provided that the first character is alphabetic.

All variables must be declared before they are used. The `IMPLICIT NONE` statement is usually the next program statement following the `PROGRAM`, which is included to enforce explicit declaration of all variables to help prevent errors due to typos Common types are `INTEGER` (integers), `REAL` (floating-point numbers), `DOUBLE PRECISION` (higher-precision floating-point numbers), and `CHARACTER` (string data).

At the beginning of the program, a few comment lines are usually inserted to provide the program description. Additional *comments* can be inserted freely anywhere before, after or within the executable section of the program.

F95 Code 1.1

```
1  ! ======================================================================
2  ! Description : A Fortran program to calculate the area of a circle.
3  ! Written by  : Z. Altac
4  ! ======================================================================
5  ! Declarationsn section
6  PROGRAM example
7  REAL, PARAMETER :: PI = 3.14159 ! PI declared and defined as constant
8  REAL :: radius, area            ! Local variables declared
9  ! Execution section
10 PRINT*, "Enter radius : "       ! Prompt input instruction
11 READ*, radius                   ! Local data (radius) is supplied
12 area = PI * radius * radius     ! Calculate the area of circle
13 PRINT *, "The area of the circle is: ", area  ! Print output (area)
14 ! Termination section
15 STOP                            ! Stop execution
16 END PROGRAM                     ! End program
```

1.2 EXECUTION SECTION

This section consists of one or more executable statements describing the tasks to be carried out by the program. Any statement pertinent to reading and/or writing the data (`READ` and `WRITE` are executable statements) computing which may involve conditional, control, assignment, and so on are part of this section.

Commenting is done to allow *human-readable* descriptions detailing the purpose of some of the statement(s) and/or to create *in situ* documentation. All characters following an exclamation mark (!) except in a character string, are commentary, and are ignored by the compiler.

1.3 TERMINATION SECTION

The termination statements used in this section are the `STOP` (an executable statement) and `END PROGRAM`. The `STOP`, which is optional, directs the computer to stop running the program. The `END PROGRAM` statement tells the compiler that the end of program is reached. The compiler automatically generates a `STOP` command when `END PROGRAM` statement is reached. Therefore, in reality, it is rarely used.

1.4 AN EXAMPLE FORTRAN 95 PROGRAM

The basic features of a FORTRAN 95 program are illustrated *via* `example.f95`, presented in **F95 Code 1.1**:

Lines 1-5, 9, and 14 are reserved entirely for comments. Note that in this example, explanatory comments are dispersed throughout the program. The other lines also include comments at the end of the statements following '!' mark.

In line 5, the `PROGRAM example` statement is placed.

In line 7, the constant `PI` is declared using the `PARAMETER` attribute and assigned the numerical value 3.14159.

In line 8, the variables (`area` and `radius`) are declared as `REAL`.

In line 10, the first executable statement in this program is the `PRINT` statement, which prints out a message prompting the user to enter the radius of the circle.

In line 11, the next executable statement is the `READ` statement, which reads in the `radius` supplied by the user.

In line 12, the third executable statement instructs the computer to calculate the area from `PI*radius`^{2} and store the result in variable `area`.

In line 13, the program execution is stopped with the `STOP` statement.

In line 14, the program is terminated with the `END PROGRAM` statement.

2 VARIABLES, CONSTANTS, AND INITIALIZATION

2.1 IDENTIFIERS AND DATA TYPES

Identifiers (i.e., symbolic names for variables, functions, and so on) are represented symbolically with letters or combinations of letters and numbers (`A`, `B`, `AX`, `XY`, `A1`, `TOL`, . . .). The first character of an identifier must be a letter, which includes underscore (_). The Fortran95 language keywords (such as `INTEGER`, `DO`, `WHILE`, `PRINT`, `COMMON`, etc.) are reserved for specific tasks and cannot be used as identifiers.

FORTRAN 95 language supports the a wide variety of built-in data types: integer (`INTEGER`), real types (`REAL`, `DOUBLE PRECISION`), complex (`COMPLEX`), boolean (`LOGICAL`), and strings (`CHARACTER`) type (*see* full listing).

`CHARACTER`: The character data type is used for single characters or a string of characters enclosed in single (') or double (") quotes, e.g., `CHARACTER(LEN=12)`. The minimum number of characters (a single character) in a string is 1.

`LOGICAL`: The logical data type stores boolean values and can only contain values `TRUE` or `FALSE`.

`INTEGER`: Integers are whole numbers that can hold positive, or negative whole values; e. g., 0, -23, or 140. On most compilers, the default kind stores integers as short integers 4 bytes in size (`KIND=2`). Long integers are usually 8 bytes (`KIND=4`). Here, the `KIND` attribute specifies the precision and representation of data types, particularly for numeric types.

`REAL` or `DOUBLE PRECISION`: Real numbers or floating point numbers with a decimal point can be represented; e. g., -2.3, 0.14, 1.2e5, and so on. The default kind for real variables (`KIND=4`) is 4 bytes (32 bits). Most compilers support a double precision data type (`KIND=8`) that uses 8 bytes (64 bits).

`COMPLEX`: The complex data type is used to represent complex numbers, which consist of a real part and an imaginary part; eg., `z=(1.0, 3.0)`. Complex data types can be single (`KIND=4`) or double precision (`KIND=4`).

FORTRAN 95 provides an intrinsic function, `SELECTED_REAL_KIND(R,P)`, to choose an appropriate *kind* based on the user's desired precision. Here, `R` is the desired number of decimal digits (precision), and `P` is the desired exponent range (the largest exponent that can be represented).

FORTRAN 95 also allows a user to define a derived `TYPE` that can contain multiple fields, each of potentially different types. For example, a user-defined type `person` may have five fields: `name`, `lastname`, and `ssn` (social security number) as `CHARACTER(20)`, `age` (as `INTEGER`) and `height` (as `REAL`).

2.2 TYPE DECLARATION OR INITIALIZATION OF VARIABLES

A *variable* is a symbolic representation of the address in memory space where values are stored, accessed, and updated. A *variable declaration* is to specify its *type* of the variable (i.e., the *identifier*) as `char`, `int`, `float` and so on, but a value to the variable has yet to be assigned. The value of a variable changes during the execution of a computer program, while the value of a `constant` does not. All variables (or constants) used in a program or subprogram must be declared before they are used in any statement.

The type of every constant or variable name must be declared in the program before it is used. The type declaration syntax for variables or constants is as follows:

```
type specifier ::  identifier-1, identifier-2, ..., identifier-n ;
type specifier, PARAMETER :: identifier = value ;
```

A *variable initialization* while type declarations can be carried out by assigning a value to it, typically with the assignment operator `"="`.

In fortran, variable identifiers that begin with the letters `I` thru `N` are assumed to be of type `INTEGER` by default. Identifiers starting with other letters are assumed to be of type `REAL`. Also the variable identifiers are *not* cases sensitive. For example, `Name`, `NAME`, or `name` denote the same variable.

The `type` declaration is carried out at the beginning of the `main()` function. `IMPLICIT NONE` overrides the default implicit typing rules for the variable (identifiers) names. When `IMPLICIT NONE` is invoked, all constant and variable names in a program module must be explicitly declared.

A variable can be initialized while being declared. The following are a set of example declarations.

```
INTEGER :: iter, maxit, i, j       ! Declare integer variables
DOUBLE PRECISION :: radius, area   ! Declare float (real) variables
INTEGER, PARAMETER :: m = 99       ! Integer m is declared and initialized
REAL, PARAMETER :: PI=3.14159      ! Real pi is declared and initialized
REAL :: yd, zd=2.0*PI              ! zd is declared and initialized
CHARACTER(1) :: yes='y', no='n'    ! yes and no are declared and initialized
CHARACTER(LEN=72) :: line          ! declared as a 72-character-long string
```

In this example, `m`, `pi`, `zd`, `yes` and `no` identifiers are not only type-declared but also initialized at the same time. The order of declaration can be important. For instance, in order to initialize `zd`, `PI` must be declared and initialized beforehand. A string is declared according to its maximum length as `CHARACTER(72)` or `CHARACTER(LEN=72)`.

In Fortran 95, variables can have default initial values depending on their type and context; zero (for numbers) or blank (for strings). Local variables declared inside a `subroutine` or `function` without an explicit initialization statement do not have a defined initial value. Their initial value is undefined, often referred to as having *undefined* or *garbage* values.

Using the value of an uninitialized variable in operations may lead to unpredictable results. It's good practice to initialize local variables explicitly before use to avoid unintended behavior.

3 INPUT/OUTPUT (I/O) STATEMENTS

An inevitable element in any program is the communication of the input and output data with the main program or its sub-programs. This is done through `READ*` and `PRINT*` statements, which are used to read initial values of variables into the program from a keyboard or print (or write) the intermediate or final values of variables to a screen.

This form of input-output is called *list-directed input/output* and the exchange of I/O data is in *free format*. The term *list-directed input* implies that the types of the variables in the variable list determine the required format of the input data. The `READ*` and `PRINT*` are commonly used to read and display data from and on input and output devices, respectively.

The syntax for the `READ`, `PRINT` and `WRITE` statements are given as follows:

```
READ (unit,fmt)  variable-1, variable-2, ... , variable-n
PRINT(unit,fmt)  variable-1, variable-2, ... , variable-n
WRITE(unit,fmt)  variable-1, variable-2, ... , variable-n
```

where **unit** denotes logical I/O unit number, **fmt** (optional) denotes the format specification for the listed data (uses free format if omitted), and the **variable**'s 1 through **n** are the list of the variables to be read, printed, or written on a device. The values of **unit=5** and **unit=6** are used for reading and writing data from the terminal device, respectively.

In free format FORTRAN 95, the **PRINT** and **READ** statements can be used with asterisks (*) to allow variables to be easily read and printed without specifying detailed format specifications (list-oriented formatting). The use of asterisks in free format Fortran allows for simple and flexible I/O operations, making it easier to work with data without worrying about detailed formatting.

Consider following code segment:

```
1  INTEGER :: n1, n2;
2  PRINT*, "Enter two integer numbers: "
3  READ *, n1, n2
4  PRINT*, "You entered",n1," and",n2
5  PRINT*, "Their squares are",n1*n1," and",n2*n2
```

where the **PRINT*** or **WRITE(*,*)** statement outputs the specified text and variable values. The * indicates list-directed formatting, which handles spacing automatically. The **READ*** (or **READ(*,*)**) statement reads input and assigns it to the specified variables. The input format is flexible and does not require specific formatting strings. For the input values of **n1=3** and **n2=5**, the code generates the following output:

```
          1         2         3         4         5
12345678901234567890123456789012345678901234567890
You entered           3  and           5
Their squares are           9  and          25
```

Here, the compiler has reserved a length of 12 characters for an integer number and right-aligned them. However, Fortran does not specify a fixed number of spaces for integers in free format output; instead, it allows compilers to determine the appropriate width based on the integer values being output and the characteristics of the output device.

3.1 FORMAT SPECIFICATIONS

A *format specification* is a placeholder denoting a value to be filled in when printing. It is a string, containing a list of *edit descriptors* which specify the exact format (width, digits after decimal point, etc.) of the numbers to be displayed. As it gives the user a control over the appearance of the output, but it can make the I/O statements a bit complicated or hard to read. The edit descriptors are presented in **Table 1.1**.

The default formatting for writing real numbers when using list-directed output typically involves scientific notation (for very large or very small numbers) or fixed-point notation with a default number of digits after the decimal point (usually around six significant figures).

```
program main
    implicit none
    integer :: i=1234
    real :: r=123.456789
    print *, "set width to 6 chars"
```

Table 1.1: Fortran edit descriptors.

Data Types		Edit Descriptors	
`INTEGER` form		`Iw`	`Iw.m`
`REAL`	Decimal form		`Fw.d` or `fw.d`
	Exponential form		`Ew.d`, `ew.d`, or `Dw.d`
	Scientific form		`ESw.d` or `ESw.dEe`
	General form		`Gw.d`
`LOGICAL` form		`Lw`	
`CHARACTER` form		`A` or `Aw`	
Positioning	Horizontal	`nX` or `nx`	
	vertical	`/`	

Here, `w` refers to the width of the field, `m` is the minimum number of characters to be used, `d` is the number of digits to the right of the decimal point, and `e` is the number of digits in the exponent.

```
        write(*,'(I6)') i
        print *, "set width to 10 chars"
        write(*,'(I10)') i
        print *, "set precision to 2, 4, and 6"
        write(*,'(F10.2,4x,F10.4,4x,f10.6)') r,r,r
    end program main
```

The output of the program is given below. Note that the first two lines are not part of the actual output; they denote the column numbers in the output file and are used to illustrate data formatting.

```
          1         2         3         4
12345678901234567890123456789012345678901234567890
 set width to 6 chars
   1234
 set width to 10 chars
      1234
 set precision to 2, 4, and 6
    123.46      123.4568    123.456787
```

For example, the following code segment reserves 5 characters (`w=5`) for the integer and 8 characters (`w=8` with `p=5`) for the float variable.

```
    real :: a=14.125, b=12.033, c=2415.1357
    write (*,'(f4.1)') a
    write (*,'(f6.2,f6.3)') a, b
    write (*,'(3f10.4)') a, b, c
    write (*,'(3(2X,E10.4))') a, b, c
    write (*,'(3(2X,ES10.4))') a, b, c
    write (*,'(2X,EN10.1,2X,EN12.2,4X,EN12.4))') a, b, c
```

The output is as follows:

```
          1         2         3         4         5
12345678901234567890123456789012345678901234567890
 14.1212.033
   14.1250   12.0330 2415.1357
  0.1412E+02  0.1203E+02  0.2415E+04
  1.4125E+01  1.2033E+01  2.4151E+03
    14.1E+00     12.03E+00      2.4151E+03
```

Here, again, the first two rows serve as a table indicating the column numbers.

4 ASSIGNMENT STATEMENT AND ARITHMETIC CALCULATIONS

Arithmetic operations involve plus (`+`), minus (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`) operators. These operations can be used with integer or floating-point types. The division operator involving an integer division truncates any fractional part, e.g., `15/3`, `16/3`, and `17/3` all yield 5.

Assignment operations are used to assign values or computed results of an expression to variables. A simple *assignment* denoted by ← (a left-arrow) in the pseudocode notation is replaced by the "`=`" sign, syntax as shown below:

```
variable_name = value ;      or
variable_name = expression ;
```

The *expression* on the right-hand side of the "=" sign is evaluated first, and its value is placed at the allocated memory location of the *variable* (i.e., `variable_name`) on the left-hand side. Any recently computed value of a *variable* replaces its previous value in memory.

Declaring a variable does not automatically assign a value to it; the variable is an *unassigned variable* until a value is assigned to it. Variables can be initialized by specified values in the same way, i.e., `pi = 3.1416`, `X=0`, etc. Following examples illustrate initialization of variables with different data types:

```
INTEGER             :: i = 5, j = 100
REAL                :: x, y = 1.0E5
DOUBLE PRECISION    :: u=2.350d0, large=9.99D+33
CHARACTER(LEN=8)    :: lname = 'Williams'
LOGICAL             :: yes = .TRUE., no = .FALSE.
```

Symbolic constants, also referred to as *parameters* in FORTRAN 95, is set up in a declaration statement containing the `PARAMETER` attribute:

```
INTEGER, PARAMETER  :: n = 10, m = 10, nm = n * m
CHARACTER(LEN=13)   :: warn1 = 'Illegal Input'
REAL, PARAMETER     :: pi = 3.14159
```

In programming, we generally employ arithmetic summations of the form `sums = sums + x` or `sums = sums - x`, which is *illegal* in normal algebra. But it makes sense in computer programs in that the values of the expressions are stored in memory. In this regard, the value of `x` is added to the current value stored in variable `sums` and the result is stored back into the memory location of `sums`.

Consider the fortran code segment below:

```
1    integer :: X = 0  ! X is assigned the value zero, i.e., initialized
2    X = X + 1   ! increment X by 1, X becomes 1
3    X = X + 2   ! increment X by 2, X becomes 3
4    X = X + 4;  ! increment X by 4, X becomes 7
5    X = X - 1   ! decrement X by 1, X becomes 6
6    X = X - 3   ! Equivalent to X = X -3, decrement X by 3, X becomes 3
```

In line 1, `X` is declared as an integer and initialized at the same time. The value of `X` in memory becomes zero. In line 2, the assignment operation adds "1" to `X`. In line 3, the value of `X` in memory is increased by 2, leading to 3, which is placed in the memory location of `X`. In line 4, when the memory value of `X` is substituted into the rhs and evaluated in place, the rhs becomes 7, and the value of `X` is updated with the current value. When this procedure is applied in line 5 and subsequently in line 6, the value of `X` leads to 6 and 3, respectively.

In fortran, the *exponentiation operation* x^y is carried out as `x**y`. While using $\exp(y \ln x)$ algebraic formula will certainly work, but it takes longer to carry out the operation and is less accurate than an ordinary series of multiplications. Thus, if possible, we should try to raise real numbers to integer powers instead of real powers.

5 SEQUENTIAL STATEMENTS

Frequently, a sequence of statements (with no imposed conditions) are used to perform a specific task. These statements will be executed in the order they are specified in the program. The statements can begin in any column. By using a semicolon (;) as a separator, multiple expressions can be placed in a single line. An exclamation mark "!" in any column is the beginning of a comment and thus the rest of the line is ignored by the compiler. A statement can be continued on the following line by appending a "&" sign at the end of the expression on the current line. For instance, the following fortran code segment illustrates how the first and second statements are presented on a single line with the use of a semicolon.

```
1       a = b * b + c * c ;  d = sqrt(a) ! Statement-1 and Statement-2
2       x1 = d / ( b + c )               ! Statement-3
3       x2 = d / ( b - c )               ! Statement-4
4       y = x1 + x2 + x3    &            ! Statement-5
5           + x4 + x5                    ! continuation of statement4
```

In line 1, statements 1 and 2 are separated with a semicolon. Single arithmetic statements occupy lines 2 and 3. In lines 4 and 5, the statement 5 extending over two lines is connected with the "&" sign.

6 RELATIONAL AND LOGICAL OPERATORS

Branching in a computer program causes a computer to execute a different block of instructions, deviating from its default behavior of executing instructions sequentially. **Table 1.2** summarizes the arithmetic, relational, and logical operators with illustrative examples.

The *relational operators* are commonly used in conditional and control constructions to control the flow of a program based on comparisons. The relational operators (`<, >, <=, >=, ==, /=`) can be used with various data types, including integers, floats, and characters. However, comparing different types may lead to implicit type conversion. While relational operators do not short-circuit like logical operators, they can be combined in logical expressions for more complex conditions.

The *logical operators* ((`.AND., .OR., .NOT., .EQV.`)) are used to combine or negate boolean ex-

Table 1.2: Arithmetic, equality/relational, logical and assignment operators in C.

Operator	Description	Example
ARITHMETIC OPERATORS		
`+, -`	Addition and subtrations	`a ± b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`**`	Exponentiation	`a ** b`
RELATIONAL OPERATORS		
`== (.EQ.)`	compares the operands to determine equality	`a == b`
`/= (.NE.)`	compares the operands to determine unequality	`a /= b`
`> (.GT.)`	determines if first operand greater	`a > b`
`< (.LT.)`	determines if first operand smaller	`a > b`
`<= (.LE.)`	determines if first operand smaller than or equal to	`a > b`
`>= (.GE.)`	determines if first operand greater and equal to	`a > b`
LOGICAL OPERATORS		
`.AND.`	Logical AND operator	`a.AND.b`
`.OR.`	Logical OR operator	`a.OR.b`
`.NOT.`	Logical NOT operator	`.NOT.a`
`.EQV.`	Logical equivalence EQV operator	`a.EQV.b`

pressions. They are essential in controlling the flow of programs, especially in conditional statements and loops. These operators are frequently used in conditional constructions, such as `IF` constructs, to evaluate multiple conditions or control the flow of loops based on complex logical expressions. The logical `AND` and the logical `OR` operators are both binary in nature (requiring two operands). The logical `NOT` operator negates the value of a Boolean operand, and it is a unary operator.

Branching structures are controlled by *logical variables* and *logical operations*. *Logical_expression* can be a combination of logical constants, logical variables, and logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result, i.e., `TRUE` or `FALSE`). Logical operators are typically used with Boolean operands.

Logical operators are used in a program together with relational operators to control the flow of the program. The `AND` and `OR` operators connect pairs of conditional expressions. Let L_1 and L_2 be two logical prepositions. In order for L_1`.AND.`L_2 to be `TRUE`, both L_1 and L_2 must be `TRUE`. In order for L_1`.OR.`L_2 to be `TRUE`, it is sufficient to have either L_1 or L_2 to be `TRUE`. When using the unary `NOT` operator in any logical statement, the logic value is changed to `TRUE` when it is `TRUE` or changed to `TRUE` when it is `TRUE`. These operators can be used to combine multiple expressions. For given `x=5`, `y=9`, `a=18`, and `b=3`, we can construct following logical expressions:

```
1  (x < y).AND.(y < a).AND.(a > x)                ! TRUE
2  (x < y).AND.(y > a).AND.(a >= b)               ! FALSE
3  ((x < y).AND.(y < a)).OR.(a < b)               ! TRUE
4  ((x > y).OR.(y > a)).OR.(a < b)                ! FALSE
5  (.NOT.(x > 6).AND..NOT.(y < 5)).AND.(a > x)    ! TRUE
```

The order of evaluation of `.AND.` and `.OR.` is from left to right.

7 CONDITIONAL CONSTRUCTIONS

In fortran, branching control and conditional structures allow the execution flow to jump to a different part of the program. *Conditional* statements create branches in the execution path based on the evaluation of a condition. When a control statement is reached, the condition is evaluated, and a path is selected according to the result of the condition.

In Fortran 95, we use `IF-THEN`, `IF-THEN-ELSE`, and `SELECT CASE` constructions, which allow one to check a condition and execute certain parts of the code if the *condition* (logical expression) is `TRUE`.

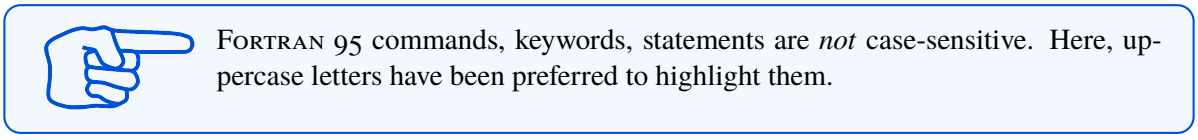
7.1 IF-THEN CONSTRUCTION

The `IF-THEN` construction whose syntax is given below is the most common and simplest form of the conditional constructs. It corresponds to the **If-Then** construct in the pseudocode. It executes a block of statements *if and only if* a *condition* (i.e., logical expression) is `TRUE`.

```
IF (condition) THEN
    STATEMENT(s)       ! if condition is TRUE
ENDIF
```

Note that the *condition* is enclosed with round brackets. The `IF` construct can also command multiple statements. A block of statements is syntactically equivalent to a single statement. Following are examples of this construction:

```
IF (num >= 0) THEN    ! simple compasion, INTEGER :: num
    PRINT*, 'num is a positive integer'
ENDIF
    ...
IF ((x + y >= z**2).AND.(z < 5.0)) THEN  ! compex conditions
    code block        ! REAL :: x, y, z
ENDIF
```



Fortran 95 commands, keywords, statements are *not* case-sensitive. Here, uppercase letters have been preferred to highlight them.

7.2 THE ELSE AND ELSE IF CLAUSES

Often we are required to execute a set of statements if one condition is true, and a different set of statements if another condition is true. In reality, there may be several options to consider. The `ELSE` and `ELSE IF` clauses can be used to establish **If-Then-Else** constructs in the pseudocode to cover all possible options.

The simplest `IF-THEN-ELSE` construct (presented below) contains another block for the statements to be executed in the case the *condition* is `FALSE`.

```
IF (condition) THEN
    STATEMENTS-t       ! if condition is TRUE
ELSE
    STATEMENTS-f       ! if condition is FALSE
ENDIF
```

One may chain multiple conditions using `ELSE IF` to test multiple possibilities. A more complicated `IF-THEN-ELSE` construct can be devised as follows:

```
IF (condition_1) THEN
    STATEMENT(s)       ! if condition_1 is TRUE
ELSE IF (condition_2) THEN
    STATEMENT(s)       ! if condition_2 is TRUE
ELSE
    STATEMENT(s)       ! if condition_2 is FALSE
END IF
```

This chain can be continued indefinitely by repeating the last statement another **ELSE IF** statement. Also, nesting **IF** statements within other **IF** statements is allowed. Following are examples of **IF-THEN-ELSE** constructions:

```
INTEGER :: i, j, k, num
   ...
IF (num > 0) THEN
    PRINT*, 'num is a positive integer'
ELSE
    PRINT*, 'num is zero or a negative integer'
END IF
   ...
IF (i < j .AND. i < k) THEN
   num = i                      ! gives the minimum of 3 numbers
ELSE IF (j < i .AND. j < k) THEN
   num = j
ELSE
   num = k
END IF
```

7.3 LOGICAL IF CONSTRUCTION

An alternative form of a block **IF** construct described above is the logical **IF** construct corresponding to the logical **If** construct in the pseudocodes. The syntax is

```
IF (logical_expression) STATEMENT
```

where **STATEMENT** is a single executable FORTRAN 95 statement. If the *logical expression* is **TRUE**, the program executes the single statement on the same line with it. Otherwise, the program skips to the next executable statement in the program.

```
      INTEGER :: p, maxit
      REAL    :: a, b, c, x, y, z
100   CONTINUE
       ...  (a code block)
      IF (p <= maxit) GOTO 100  !  a Repeat-Until Loop
       ...
      IF (a*b+c <= 99) ans = .TRUE.
       ...
      IF (x /= 0.0 .AND. y /= 0.0) z = 1.0/( x + y )
       ...
```

All control constructs can be both *named* and *nested*:

7.4 THE SELECT CASE CONSTRUCTION

The **SELECT CASE** construction is an alternative to the **IF-ELSE-IF** ladder. A **SELECT CASE** construction allows a multi-decision case to be executed based on the value of an integer selector variable.

The general form of the **SELECT CASE** statement is as follows:

```
SELECT CASE (expression)
    CASE (case selector_1)
        STATEMENTS-1
    CASE (case selector_2)
        STATEMENTS-2
    ....
    CASE default
        STATEMENTS-n
END SELECT
```

where *case selector*$_1$, *case selector*$_2$, and so on are integers. Upon evaluating the case *expression*, if and when it matches one of the available cases, the construction branches to the matching case and executes the block statements (**STATEMENTS-1** and so on) from that point onward. Otherwise, it branches to the *optional* **default**, where the statements corresponding to unmatched cases are executed.

Following are examples of this construction:

```
INTEGER :: year=3, num = 0     ! year and num are initialized
 SELECT CASE (year)
     CASE (1)
         PRINT*, "Freshman"
     CASE (2)
         PRINT*, "Sophomore"
     CASE (3)
         PRINT*, "Junior"      ! Output is 'Junior'
     CASE (4)
         PRINT*, "Senior"
     CASE default
        PRINT*, "Graduated"
END SELECT
  ...
SELECT CASE (num)              ! Output is  zero
     CASE (:-1)                ! case of num<=-1
         val = -1
     CASE (1:)                 ! case of num>= 1
         val =  1
     CASE default              ! case of num = 0
         val =  0
 END SELECT
```

The first code segment uses **year** to execute **SELECT CASE** construction. For the case of **year=1**, **Freshman** is displayed; for **year=2**, **year=3**, and **year=4**, **Sophomore**, **Junior**, and **senior** are displayed, respectively. If **year** corresponds to none of the above, the message **Graduated** is displayed. The second example uses **num** to perform the **SELECT CASE** construction. The **(:-1)** case value corresponds to **num<0**, **(1:)** matches to **num>0**, and **default** mathes to **num=0**.

It is good practice to include a **DEFAULT CASE** clause in the **SELECT CASE** construct to catch logical errors or illegal inputs that may occur in a program.

8 CONTROL CONSTRUCTIONS

Control (loop) constructions are used when a program needs to execute a block of instructions repeatedly until a *condition*is met, at which time the loop is terminated. There are three control statements in most programming languages that behave in the same way: **DO WHILE-**, **DO-ENDDO** (equivalent to **Repeat-Until** loop), and **for-**constructs.

8.1 DO WHILE CONSTRUCTION

A DO WHILE construct (equivalent of **While**-loop in the pseudocode) has the following form:

```
DO WHILE (condition)
    STATEMENT(s)       ! if condition is true
ENDDO
```

In the **DO WHILE** construct, the *condition* is evaluated before the statement block. If the *condition* is **TRUE**, the block of statement(s) will be executed. If the *condition* is initially **FALSE**, the statement block will be skipped.

Consider the following **DO WHILE** construction:

```
1   INTEGER :: i
2   i= 0
3   DO WHILE (i < 10)          ! as long as i<=9 execute the block
4      i = i + 1               ! increment i
5      PRINT*, i, i*i, i*i*i   ! print i, i^2, and i^3
6   END DO
```

This code creates a table that gives integer values from 1 to 10 along with their squares and cubes. The code block consists of lines 2 and 3.

8.2 DO- CONSTRUCTION

A **DO** construct is used when the number of iterations to be performed is known beforehand. The **DO** construction, also referred to as an iterative **DO** loop or a *counting loop*, is used when a block of STATEMENT(S) is to be executed a specified number of times. The **DO** construct, which is equivalent of **For**-loop in the pseudocode, has the form

```
DO  loop_variable = start, stop, step
     STATEMENT(s)          ! block statements are executed
ENDDO
```

where **loop_variable** is an integer loop *control variable* or *loop index*, the *start* and *stop* are the initial and the final values of the loop variable, respectively, and the *step* denotes the increment (if $step > 0$ or decrement $step < 0$) in the loop variable. If *step* is omitted, then the default value of +1 is applied. The total number of iterations can be determined by $(stop - start + step)/step$.

In the following examples, the **DO** loop over variable-**i** is executed for every **i** from **2** to **9** with increments of **+1**, while the loop over variable-**j** is executed for every **j** from **1** to **8** with increments of **+2**. On the other hand, the **DO** loop with loop variable-**k** is executed for every **k** from **5** to **3** with decrements of **-1**, while

the loop over the variable-`m` is executed for every `m` from `6` to `1` with decrements of `-2`. The last `DO` loop is supposed to iterate from `20` to `3` with the increments of `2`; however, the index value of `3` is not within the reach with increments of `2`. In order for this loop to run, the upper bound of the loop should be greater than 20.

```
INTEGER :: i, j, k, m
DO i = 2, 9
   STATEMENT(s)    ! The loop-block is executed for i = 2, 3,..., 9
ENDDO
DO j = 1, 8, 2
   STATEMENT(s)    ! The loop-block is executed for i = 1, 3, 5, 7
ENDDO
DO k = 5, 3, -1
   STATEMENT(s)    ! The loop-block is executed for k = 5, 4, 3
ENDDO
DO m = 6, 1, -2
   STATEMENT(s)    ! The block is executed for k = 6, 4, 2
ENDDO
DO j = 20, 3, 2
   STATEMENT(s)    ! The block is skipped since 2 > 0 and 3 < 20
ENDDO
DO j = 5, 20, 5
   STATEMENT(s)    ! The block is executed for i=5, 10, 15, 20
ENDDO
```

It is easy to use in nested loop settings (with arrays) due to having clearly identified loop indexes. But do not modify a `DO` loop index inside the loop block because the loop index is intended to be controlled by the `DO` statement itself. Changing it manually can disrupt the normal flow of the loop.

8.3 DO LOOP AS A Repeat-Until CONSTRUCTION

A conditional loop with a test *condition* at the bottom (i.e., equivalent to Repeat-Until in the pseudocodes) can be constructed with the `DO` loop. The `DO` construction is used along with `EXIT` to execute a block of code statements repeatedly until a condition is satisfied. The general syntax is given as follows:

```
DO
    STATEMENT(s)      ! if condition is false
    IF ( condition ) EXIT
ENDDO
```

Note that the code block up to the `IF` statement is executed at least once. Such loops may contain more than one `EXIT` statement. This construction is similar to `DO WHILE` construct; however, the code statements are executed as long as the *condition* is `FALSE`. When the <*condition*> becomes `TRUE`, the `EXIT` statement is executed and control jumps to the first statement after the `END DO`.

The following code segment is a **Repeat-Until** loop constructed using a `DO-END DO` loop with an `EXIT` statement. The code sums up the even numbers up to n. The loop will be executed until the <*condition*> ($i \geqslant n$) becomes `TRUE`.

```
INTEGER :: i, sums, n
i = 0; sums = 0; n = 50
DO
   i = i + 2
   sums = sums + i
   IF (i >= n) EXIT
ENDDO
PRINT*, "Sum of the even numbers up to",n,"is",sums
```

This structure can also be used to construct a **While**-loop by placing the conditional `IF` immediately after the `DO` statement.

8.4 THE EXIT AND CYCLE STATEMENTS

Sometimes it may be necessary to exit a loop except for a *condition* specified at the top or bottom. The `EXIT` statement also provides an early exit from an indexed `DO` loop construction. In other words, execution of the `EXIT` statement results in transferring the control to the next executable statement after the `END DO` statement to which it refers. In nested loops, it terminates execution of the innermost `DO` construct it is contained within.

Consider the following fortran code segment with two loops, where the `EXIT` statement is inside the innermost loop.

```
INTEGER :: i, d
DO d = 2, 3
    DO i = d, 6
        PRINT *, "d=", d, " i=", i
        IF (i == 3) EXIT
    ENDDO
ENDDO
```

The code output is

```
d=           2  i=           2
d=           2  i=           3
d=           3  i=           3
```

Note that the `PRINT*` statement is before the logical `IF` construct. When `i` becomes 3, the loop for `i=4, 5` and `6` is skipped, and the loop is transferred to the next loop (over `d`-variable), which continues with `d=3`. The innermost loop over `i`-variable spans from 3 to 6, but the iterations are carried out only for `i=3`, i.e., (until) the `EXIT` statement is executed.

When a `CYCLE` statement is executed inside a `DO` loop, the statements after the point at which CYCLE is invoked in the loop are not executed, and the loop control is returned to the top of the loop. Then the loop index is incremented, and iteration continues until the index reaches the *stop*-value. An example of the `CYCLE` statement in a `DO` loop is presented below:

```
1  INTEGER :: i
2  DO i=1, 8
3      IF ( i == 3 .OR. i==5 ) CYCLE
4      PRINT *, i
5  END DO
6  PRINT *, "End of loop!"
```

Here, the `DO` loop spans from `1` to `8`, but when the loop variable becomes `i=3` or `i=5`, the statements following the logical `IF` statement are skipped. In the code output below, it is noted that the loop statements following line 3 are skipped.

```
           1
           2
           4
           6
           7
           8
 End of loop!
```

8.5 NAMED CONDITIONAL AND CONTROL CONSTRUCTIONS

FORTRAN 95 allows a *name* to be assigned to a conditional or control construction. When a large number of `DO`, `IF-THEN`, `IF-THEN-ELSE`, or `SELECT CASE` or nested complex constructions are implemented in a program, it may sometimes be helpful to *name* these constructions. Following are examples of named constructions:

```
INTEGER :: i, d
loop1 : DO d = 2, 3
      loop2 :  DO i = d, 6
                  PRINT *, "d=", d, " i=", i
                  IF (i == 4) EXIT  loop2
               ENDDO  loop2
        ENDDO loop1
        ...
  if1 : IF (d>0) THEN
           PRINT*, "d=",d
            ...
           if2 : IF ( expression ) THEN
                    ...
                 ELSE
                    ...
                 END IF if2
         ENDIF if1
         ...
```

The `EXIT` or `CYCLE` statements can refer to a particular loop, as demonstrated by the following code segment:

```
1  loop0: DO
2      loopI: DO
3               IF (a > b) EXIT loop0       ! jump to line 9
4               IF (a == b) CYCLE loop0     ! jump to line 1
5               IF (c > d) EXIT loopI       ! jump to line 8
6               IF (c == a) CYCLE           ! jump to line 2
7            END DO loopI
8        END DO loop0
9        ...
```

9 ARRAYS

An array is a special case of a variable representing a set of data (variables) under one group name. Arrays can have one, two, or more dimensions. The subscripts are always *integers*.

9.1 DECLARING ARRAYS

Arrays must always be explicitly declared at the beginning of each program because the range and length of an array are critical in programming. The `DIMENSION` attribute in the type declaration statement declares the size of the array being defined.

Consider the following FORTRAN 95 code segment, which includes one-, two-, and three-dimensional array declarations. In this example, the integer-type variable `no` is declared as a one-dimensional array of length 16. The real-type array variables `arra`, `arrb`, and `arrc` are declared as one-dimensional arrays of length 100. The 14-character-long string variables `name` and `last_name` variables are declared as one-dimensional arrays of length 999. The double precision-type `arr2d` and integer-type `arr3d` variables are defined as two- and three-dimensional arrays, respectively.

```
INTEGER, DIMENSION(16) :: no
REAL, DIMENSION(0:99)  :: arra, arrb, arrc
CHARACTER(len=14), DIMENSION(999) :: name, last_name
DOUBLE PRECISION, DIMENSION(10,10)  :: arr2d
INTEGER, DIMENSION(5,2,2)  :: arr3d
```

Most compilers also allow a mixture of declarations. For example, in the following declarations, arrays (involving different sizes) and non-array variables of the same type are declared with a single statement.

```
INTEGER:: i, j, no(16), arr3d(5,2,2)
REAL   :: arra(0:99), arr3d(5,2,2), pi
CHARACTER(len=14) :: name(99), last_name(99), chrindx
```

In this case, the first statement defines integer variables `i` and `j` and one- and three-dimensional array variables `no` and `arr3d`. Besides variable `pi`, the second statement creates two real array variables, namely, `arra` and `arr3d`. The character variables of 14 characters long are declared as arrays of size 99.

9.2 CHANGING THE SUBSCRIPT RANGE

The elements of a one- or multi-dimensional array are subscripted, by default, starting from 1. However, in some algorithms, it is more preferable to index the array elements with subscripts that start with zero or some other value. FORTRAN 95 allows the declaration and use of user-defined subscript ranges, as shown below:

```
INTEGER, DIMENSION(60) :: no    ! no(0), no(1), ...., no(60)
REAL, DIMENSION(-2:9)  :: a     !  a(-2), a(-1), a(0), ..., a(9)
INTEGER, DIMENSION(3,7):: indx  ! indx(3), indx(4), ..., indx(7)
```

9.3 DECLARING VARIABLE SIZE ARRAYS

In many programs, the size of the arrays and memory requirements are subjected to the size of the input arrays. The Fortran language allows declaration of the array sizes by using named constants, which make it easy to resize the arrays in the program. For example,

```
INTEGER, PARAMETER :: MAKS=99, ROW=5, COL=5  ! no(0), no(1), ...., no(60)
INTEGER, DIMENSION(MAKS):: indx     ! a variable size array declaration
REAL, DIMENSION(M, N)   :: matrixA  ! a variable size matrix declaration
```

9.4 INITIALIZING ARRAYS

Initial values may be assigned to an array using *assignment statements* by either element by element in a `DO` loop or all at once with an array constructor.

Initialization With Assignment: The following fortran code segment initializes the elements of integer arrays. In line 1, all elements of `narr` are set to 3. In line 2, all elements of `arr2d` are set to 0. The `DO` construct in lines 4-6 allows an initialization process that can be formulated as in line 5. Lastly, in line 7, the elements of an array can be initialized as a list.

```
INTEGER :: i, narr(100), arr(5), arr2d(5,5), arr1d(5)
narr = 3   ! yields narr(1) = narr(2) = ... narr(100) = 3
arr2d = 0  ! yields arr2d(1,1) = arr2d(1,2) = ... =arr2d(5,5) = 0
DO i = 1, 5
   arr(i) = i  ! yields arr(1) = 1, arr(2) = 2, ..., arr(5) = 5
ENDDO
arr1d = (/ 1, 2, 3, 4, 5 /)
```

Initialization With Type Declaration: Initial values of an array can also be specified at compilation time while declaring its values in a type declaration statement using an array constructor. This method is suited well to initialize only small-size arrays. For example, the following example declares a five-element integer array arrA and a three-element real array arrB.

```
INTEGER, DIMENSION(5) :: arrA = (/ 5, 11, -33, 45, 999 /)
REAL, DIMENSION(3)    :: arrB = (/ 1.2, 9.8, 0.59 /)
```

where initialization gives `arrA(1)=5`, `arrA(2)=11`, ..., `arrA(5)=999`, `arrB(1)=1.2`, `arrB(2)=9.8`, and `arrB(3)=0.59`. Note that the number of elements in the data must match the number of elements in the array being initialized.

Initializing large arrays can be carried out using implied `DO` loop. An implied `DO` loop has the general form

```
(arg-1, arg-2, ..., index = start, stop, step)
```

where `arg-1`, `arg-2`, ... are values evaluated each time the loop is executed, and `index`, *start*, *stop*, and *step* behave in the same way as they do for ordinary counting `DO` loops.

This method is well-suited if the arrays whose initial values can be formulated. Afterwards, no matter how large the array length is, the initialization can be carried out simply using the implied `DO` loop. For example, the following arrays can be initialized as `arr(i)=i` and `x(i)=0.1*i`.

```
1 INTEGER, DIMENSION(5) :: arr = (/ i, i = 1, 5 /)
2 REAL, DIMENSION(10) :: x = (/ ( 0.1*i, i = 1, 10) /)
```

Array constructors always generate one-dimensional arrays. In order to initialize two-dimensional arrays using array constructors, FORTRAN 95 provides `RESHAPE`, an intrinsic function, to change the shape of an array without changing the data in it.

The syntax of the `RESHAPE` function is

```
output = RESHAPE( array1, array2 )
```

where **array1** contains the data to reshape, and **array2** is a rank-1 array describing the new shape.

Consider the following example, where a one-dimensional array is initialized as a two-dimensional array of three rows and four columns.

```
INTEGER :: i, j, int2d(3,4)
int2d = RESHAPE ( (/ 1,1,1,2,2,2,3,3,3,4,4,4 /), (/3,4/) )  ! Resize
DO i=1,3
  PRINT 10, (int2d(i,j), j=1,4)    ! Generate output in matrix form
ENDDO
10 FORMAT(1x,5(1x,i2))
```

The code output is three rows of "1 2 3 4".

The initialization for arrays may also be carried out at compilation time during the type declaration. In Fortran, arrays are allocated in the column order; thus, the values listed in the type declaration statement must also be in the column order. That is, all elements of the array must be listed in column-by-column order, starting from the first column.

The type declaration and initialization of array **int2d** in the prior example can be combined as follows:

```
INTEGER, DIMENSION(3,4) :: int2d(3,4) = &
    RESHAPE ( (/ 1,1,1,2,2,2,3,3,3,4,4,4 /), (/3,4/) )  ! Resized
```

Use the **RESHAPE** function to change the shape of a one-dimensional array. This is particularly handy when used with an array constructor to initialize arrays in any desired shape.

9.5 WHOLE ARRAY OPERATIONS AND ARRAY SUBSETS

Two or more conformable arrays may be used together in arithmetic operations and assignment statements. If the arrays are conformable, then the desired arithmetic operation will be performed on an element-by-element basis.

The whole array arithmetic expressions do not require DO-loops and operate with the array names as if they were scalars. Consider the following code segment, where **aa = arr1 + arr2** and **bb =3** $\times$ **arr2** are computed:

```
1  INTEGER, DIMENSION(5) :: arr1 = (/ 3,-2, 1, 5, 4 /)
2  INTEGER, DIMENSION(5) :: arr2 = (/ 4, 1, 3, 1,-4 /), aa, bb
3  INTEGER :: i, j
4  ! element by element operations
5  DO i = 1, 5
6     aa(i) = arr1(i) + arr2(i)  ! array addition
7     bb(i) = 3 * arr2(i)        ! scalar multiplication
8  ENDDO
9  ! whole array operations
10 aa = arr1 + arr2              ! array addition
11 bb = 3 * arr2                 ! scalar multiplication
```

The two arrays have the same shape, i.e., they are *conformable*. The element by element operations in lines 5-8 can be replaced by the equivalent operations using the array names, as illustrated in lines 10 and 11.

Consider following 5×4 matrix.

$$\texttt{arr} = \begin{bmatrix} 2 & -3 & 4 & 1 \\ 5 & 0 & 1 & -3 \\ -1 & 2 & 2 & 7 \\ 1 & 3 & -2 & 5 \\ 3 & 1 & 0 & 4 \end{bmatrix}$$

The array subset corresponding to the second row and second column of the array is selected as `arr(2,:)` and `arr(:,2)`, respectively.

$$\texttt{arr(2,:)} = \begin{bmatrix} 5 & 0 & 1 & -3 \end{bmatrix} \quad \text{and} \quad \texttt{arr(2,:)} = \begin{bmatrix} -3 \\ 0 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$

The array subset corresponding to the matrix obtained by removing the first row and first column is generated as follows:

$$\texttt{arr(2:5,2:4)} = \begin{bmatrix} 0 & 1 & -3 \\ 2 & 2 & 7 \\ 3 & -2 & 5 \\ 1 & 0 & 4 \end{bmatrix}$$

10 FUNCTION AND SUBROUTINE CONSTRUCTIONS

Fortran programs consist of several key elements that work together to create a structured and functional application. The main program is defined with the `PROGRAM` statement, which marks the beginning of the program, and delimited with `PROGRAM-END PROGRAM`. The instructions to carry out the intended task are carried out here. But it is generally impractical to include everything in a main *program* from A to Z. Such programs would not only be too long but also too complicated to be practical. It is also common for a set of tasks to be repeated multiple times. In fortran, to avoid repetition in coding, procedures such as `FUNCTION` and `SUBROUTINE`s are utilized.

10.1 THE SUBROUTINE PROCEDURES

A subroutine in Fortran is a block of code that performs a specific task and can be reused multiple times throughout a program. This is a procedure equivalent to **Module-End Module** construction in pseudocodes. It allows for modular programming, making code more organized and easier to manage. Subroutines may have a number of inputs and output variables or may not produce any output at all (*void* subprogram). An example is a subroutine that terminates the program after printing a specific *warning* or *error message*.

Subroutine Declaration: A subroutine is an independent program unit that begins with the `SUBROUTINE` statement and ends with the `END SUBROUTINE` statement. A subroutine has the following syntax:

```
SUBROUTINE sub_name (p1, p2, ..., pn)
   Declarations of arguments and local variables
   Executable STATEMENTS
RETURN                              !  optional
END SUBROUTINE sub_name
```

where p1, p2, ..., pn are the *optional* dummy input/output arguments, which may be variables and/or arrays that are being passed from the calling program to the subroutine, RETURN is the statement used to exit from a subroutine and return control to the calling program or routine. The RETURN statement is *optional*. The END SUBROUTINE statement marks the conclusion of a subroutine. It signals to the compiler that the subroutine's definition is complete. Because subroutines in a program are compiled separately, local variable names and expression labels can be reused across different routines.

Calling a Subroutine: A subroutine is accessed using the CALL statement. The main, including another subroutines, can call a subroutine. But a subroutine cannot call itself unless it is a so-called *recursive* subprogram.

To access a subroutine and execute the intended task, the calling program requires a CALL statement. The variables, constants, or expressions can be passed into a subroutine as arguments. The syntax for the CALL statement is given as

```
CALL subname (ap1, ap2, ..., apn)
```

where ap1, ap2, ..., apn are the actual input/output arguments. The order and type of the arguments in this list must match the order and type of the arguments declared in the subroutine. When the task in the subroutine is completed, the control is returned to the calling program or subroutine using the RETURN statement. A subroutine may have several RETURNs.

Optional arguments can be defined in the subroutine if needed, using the OPTIONAL attribute. Subroutines do not return values like functions; they use arguments to communicate.

Argument Declaration: Type declarations of variables in an argument list are required as usual, but additional information can be supplied with the INTENT attribute specifying the intended use of dummy arguments in procedures. This is because, in many programming languages, the arguments are passed to subprograms either *by value* or *by reference*. When arguments are passed by value, changes to the argument do not affect the original value of the variable in the caller. When an argument is passed by reference, the called function can modify the original value of the variable.

In Fortran, when the INTENT of the arguments is omitted, the arguments are treated as having default intent behavior, i.e., all variables are passed *by reference* to the arguments of the subroutines. Modifying an argument (input variable) accidentally in the subroutine will modify the values of the arguments (or variables) in the calling subroutine or program. Thus, to avoid potential problems, the local variables and all arguments with intended uses should be declared in the subroutine before they are used. The use of the INTENT attribute helps clarify whether an argument is meant to be *input*, output, or both. Without explicit INTENT declarations, the compiler cannot enforce any restrictions on how the arguments are used.

INTENT(IN) indicates that the argument is intended for input only. The procedure can read the value, but it cannot modify it.

INTENT(OUT) indicates that the argument is intended for output only. The procedure is expected to modify this argument, and it will not have a defined value upon entry.

INTENT(INOUT) indicates that the argument can be both input and output. The procedure can read and modify the value of this argument. It is the default intent of the variables.

The following subroutine example has two real x and y variables intended as input arguments, four real variables sums, diff, prod, and divv intended as output arguments, and a real variable n intended as input as well as output argument. Note that n is modified in line 10 of the subroutine.

Table 1.3: Some of the built-in functions available in FORTRAN 95.

Functiom	Description
`ABS(x)`	absolute value of real x
`SQRT(x)`	square root of x
`EXP(x)`	the exponential value of e^x
`FLOAT(x)`	converts the integer `x` to a default real value
`LOG(x)`	the natural logarithm (base e) of x
`LOG10(x)`	the logarithm (base 10) of x
`SIN(x)`	the sine of x (angle in radians)
`ARCSIN(x)`	the inverse sine of x
`SINH(x)`	the hyperbolic sine function $\sinh x$,
`MOD(x, y)`	the remainder of x divided by y
`INT(x)`	convert x to integer
`LEN(x)`	returns the length of string x
`TRIM(x)`	removes trailing spaces from string x

```
1    SUBROUTINE example (n, x, y, sums, diff, prod, divy)
2    IMPLICIT NONE
3    REAL, INTENT(IN) :: x, y  ! INTENT improves code readability and help
4    REAL, INTENT(OUT):: sums, diff, prod, divy ! catch errors at complilation
5    REAL, INTENT(INOUT):: n   ! argument to be passsed by reference
6    sums = x + y              ! assignment of a newly computed value
7    diff = x - y              ! assignment of a newly computed value
8    prod = x * y              ! assignment of a newly computed value
9    divy = x / y              ! assignment of a newly computed value
10   n = n * ( x*x + y*y )     ! input n is modified at the output
11   END SUBROUTINE            ! Optional RETURN is omitted here
```

10.2 THE FUNCTION PROCEDURES

In FORTRAN 95, functions are a key programming construction that calculates a value based on an algorithm and returns it to the caller. It is equivalent to **Function-End Function** construction in pseudocodes. A function can have one or more input arguments that are used in calculations and then return a *single result*. Implementation of functions in a program promotes code reuse, improves organization, and simplifies complex calculations.

In FORTRAN 95, functions can be viewed in two categories: (i) *intrinsic functions*, those provided with the standard Fortran library, and (ii) *user-defined functions*, those prepared by the user.

INTRINSIC FUNCTIONS: Intrinsic functions are built-in library functions provided by FORTRAN 95 that perform a wide range of common operations, such as common mathematical calculations, string manipulations, character manipulations, input/output operations, etc. A small list of such functions is presented in **Table 1.3**, where the variables x and y are single precision.

FORTRAN 95 functions have specific prefixes that indicate the type of data they operate on. In most built-in functions, the prefixes `C`, `D`, and `I` used with the standard function names return the output values of type complex, double precision, and integer, i.e., `CABS(x)`, `DABS(x)`, and `IABS(x)` are used when `x` is declared as `COMPLEX`, `DOUBLE PRECISION`, and `INTEGER`, respectively.

USER-DEFINED FUNCTIONS: FORTRAN 95 also allows the user to write a custom function, tailored to his own needs, to perform a particular task not found in the built-in library functions. A function is only prepared

once, and it can be accessed and executed from the **main** function or any other **functions** whenever needed. Once the function code completes the specific task it is supposed to perform, one output variable (scalar or array) is returned to the calling function.

There is no restriction to the number of functions that can be present in a program. A function can be used with any other relevant programs and can be invoked or accessed many times in the same program. The general structure of a user-defined **FUNCTION** is shown below:

```
return-type FUNCTION func_name (p1, p2, ..., pn)
       Argument and local variable declarations
       Executable statements
END FUNCTION func_name
```

where **p1**, **p2**,... , **pn** are the input dummy argument list. **return-type** indicates whether or not the return value of the function is integer, real, double precision type, etc. The **func_name** is a valid **FUNCTION** identifier. The ***return-type*** is the data type of the result returned to the caller.

The return-type, function-name, and argument-list together are often referred to as the function *header*. A function can also have its own internal (*local*) variables that are accessible only internally, i.e., its content is invisible to other functions.

Make sure that the type of any user-defined function is declared both in the function itself and in any procedure that calls the function. Also, to prevent accidentally modifying its input arguments, always declare the arguments with the **INTENT (IN)** attribute.

Invoking Functions in Programs: Functions in both categories are similar to subroutines. They are both called by naming the function along with their entire argument list (i.e., referred to as function call) to which information is passed.

```
PROGRAM example
REAL :: x, y, z
y = SIN(x)*SQRT(x*x+z*z)  ! implemented in arithmetic assignment
PRINT*, x, EXP(-x*x)      ! used with I/O statements in expressions
PRINT*, x, FX(x*x)  ! custom function in I/O in arithmetic expressions
END PROGRAM
REAL FUNCTION FX(x)
REAL, INTENT(IN) :: x
FX = X*EXP(-X)            ! Custom function also uses intrinsic function
END FUNCTION
```

Consider the following example: a real-typed **function avg**, which is designed to calculate the average of an array of real numbers of length **n**. Note that the input argument lists as well as the local variables (**i** and **sums** are properly declared with the **INTENT(IN)** attribute at the beginning of the function before they are used. The sum of the array values, **sums**, is found using a **DO** loop. The average is saved on **avg**, function name.

```
REAL FUNCTION avg(n, arr)
INTEGER, INTENT(IN) :: n
REAL, INTENT(IN), DIMENSION(n) :: arr
INTEGER :: i
```

```
REAL :: sums
sums = 0.0
DO i=1, n
   sums = sums + arr(i)
ENDDO
avg = sums/float(n)  ! float is a built-in function to convert n to real
END FUNCTION avg
```

The following (return-value) function finds the maximum of two real numbers **a** and **b**.

```
FUNCTION findmax(a, b) RESULT(max_value)
    implicit none
    REAL, INTENT(IN) :: a, b
    REAL :: max_value
    IF (a > b) THEN
        max_value = a
    ELSE
        max_value = b
    END IF
END FUNCTION findmax
```

Note that this **float**-type function returns a **float** value under any circumstances.

10.3 RECURSIVE FUNCTION

In Fortran 95, a function defined as *recursive* in a program is allowed to call itself multiple times. A **RECURSIVE FUNCTION** can call itself repeatedly until a certain condition or task is met. That is why a conditional construction is required to prevent the call from going into an infinite loop to terminate the task.

Consider the recursive function $f_n(x) \leftarrow n + x * f_{n-1}(x)$ with $f_0(x) = x$. This function can be made a recursive function as follows:

```
1  REAL RECURSIVE FUNCTION func(n, x) RESULT(f)
2  IMPLICIT NONE
3  INTEGER, INTENT(IN) :: n
4  REAL, INTENT(IN) :: x
5  IF (n==0) THEN
6     f = x
7  ELSE
8     f = float(n) + x * func( n - 1, x)
9  ENDIF
10 RETURN
11 END FUNCTION func
```

In this example, in line 1, the type of the function is stated as **RECURSIVE** and the local intermediate variable (**f**) assigned for the result is **real**. The function has two input arguments (intended as IN): integer **n** in line 3 and real **x** in line 4. The self-calling takes place in line 8 as **func(n - 1, x)**. The recursive function can be executed anywhere in the program or subprograms simply by invoking **func(n,x)**, which takes the value of **f** on return.

Bibliography

[1] CHAPMAN, S. J., *Introduction To Fortran 90/95*. McGraw-Hill's BEST–basic engineering series and tools, 1998.

[2] CHIVERS, I., SLEIGHTHOLME, Jane. *Introduction to Programming with Fortran*. Springer International Publishing, 2018.

[3] CLERMAN, N. S., SPECTOR, W. *Modern Fortran: Style and Usage*. Cambridge University Press, 2012.

[4] COUNIHAN, M. *Fortran 95*. Taylor & Francis, 1996.

[5] HAHN, B. H. *FORTRAN 90 for Scientists and Engineers*. Elsevier Science, 1994.

[6] METCALF, M., REID, J., COHEN, M. *Modern Fortran Explained*. Oxford Univ. Press, 2011.

[7] www.tutorialspoint.com

[8] www.fortrantutorial.com

[9] fortran-lang.org