A Brief Introduction to

Fortran-90

for Numerical Programming

by

Pedro B. Perez

North Carolina State University

June 1999

Introduction of Fortran-90 for Numerical Programming

Abstract

This brief review of the Fortran 90 language is intended to introduce some of the Fortran-90 features to programmers familiar with other scientific languages. This work is by no means a complete introduction to the Fortran 90 language and will focus only features pertinent to numerical programming. Examples and references are provided for the reader to commence a self paced training journey. This work is based on lecture notes from CSC112 and CSC302 at NCSU and references to exercises are provided.

Introduction of Fortran-90 for Numerical Programming

Table of Contents

1.0 Introduction 1
2.0 Source Code Form 1
3.0 Data Types 4
4.0 Code Structure 5
5.0 Mathc Operations 7
6.0 List Directed Input and Output 8
7.0 Intrinsic Functions 12
8.0 Structured Programming 14
9.0 Function Sub-Program Unit 22
10.0 Subroutine Sub-Program Unit 28
11.0 Formatted Input and Output 32
12.0 Structured Data Types - Arrays 35

Appendices

A Fortran 90 Compiler at NCSU
B Fortran 90 Derived Data Type
C Fortran 90 Pause Routine

1.0 Introduction

Fortran 90 is a standardized programming language which includes the entire Fortran 77 dialect. The Fortran-90 highlights which are included in this monogram will be presented in the order the author believes will impact the beginning Fortran programmer. The following topics are in included:

- Source code form and Character set
- Code structure
- Input/Output
- Data types
- Program Controls
- Relational Operators
- Arrays
- Intrinsic Functions
- Sub-Programs and modules

No attempt is made to be inclusive of all the Fortran-90 features under the above headings. Only an introductory level explanation with an example are provide. The reader is encouraged to read the reference literature for a complete explanation.

The Fortran 90 compiler used at NCSU as of the date of this document is the Numerical Algorithm Group Fortran 90 compiler. One simply enters at eos prompt "**add nagf90**" and then invoke the compiler by entering at the eos prompt "**f90 *myprogram*.f90**".

2.0 Source Code Form

Free-form Source Code

Source code lines in free-form may now be 132 characters in length. A programmer may use any column in the Fortran record for coding. The reserved column 7 through 72 for Fortran statement is lifted as is the column 6 reservation for continuation characters.

Most Fortran-90 compilers identify the source code form from the extension of the file name. Typically a " *f90* " extension will always be for a free-form source code. File extensions such as " *for* " or " *f* " are likely reserved for fixed-form source code. Programmers are always encouraged to check with the Compiler User Manual for the proper extension. However, the following extensions are recommended since the author have used them successfully.

File Extension Naming Convention

*.f90	Free-form Fortran-90 Source Code
*.for	Fixed-form Source Code

Comments

Comments may appear anywhere after an exclamation point (!) in free-form source codes. The exclamation point terminates a Fortran record and the compiler will proceed to the next line.

```
a = pi * r**2 !  Calculate the area of a circle
```

Variable and Program Unit Names

Fortran-90 allows a variable or program unit name length to be up to 31 characters in length and recognizes the underscore (_) in symbolic names.

```
area_of_a_circle = pi * circle_radius**2
```

Multiple Statements on a Line

Fortran-90 allows for multiple statements to appear on the same Fortran line. A semicolon (;) separates these statements. This feature should be used with care since it may lead to unreadable coding.

```
a = 0.0 ; b = 3.5 ; n = 4  ! Initialize three variables
```

Fortran-90 Character Set

Fortran-90 adds eight new characters to the Fortran 77 standard character set. These are listed in Table I.

Table I
New Characters in Fortran-90

Symbol	Name	Function
!	Exclamation Point	Precedes a comment
"	Double quotation	Delimits a character constant
%	Percent	Separates derived type components
&	Ampersand	Designates a continuation line
;	Semicolon	Separates statements on a line
<	Less than	Symbolic relational operator for .LT.
>	Greater than	Symbolic relational operator for .GT.
::	Double colon	Allows all attributes of a variable to be declared on a statement
_	Underscore	Used for symbolic names

The Fortran language is not case sensitive. It is recommended to use small caps for source code and comments.

Fortran Line Continuation

Fortran-90 limits the number of continuation lines to 39. However, compiler options may extend this limit. An ampersand (&) is used at the end of initial line to indicate the following line is a continuation.

Example:

```
The following complete statement:

If (exam_grade > 90. .and. lab_grade > 90.) grade = "A"
```

may be written on two lines of source code with a continuation:

```
if (exam_grade > 90. .and. &
    lab_grade > 90.) grade = "A"
```

Character strings which must be continued on the next line require two "&". One at the end of the first line and the other at the start of the continuation line:

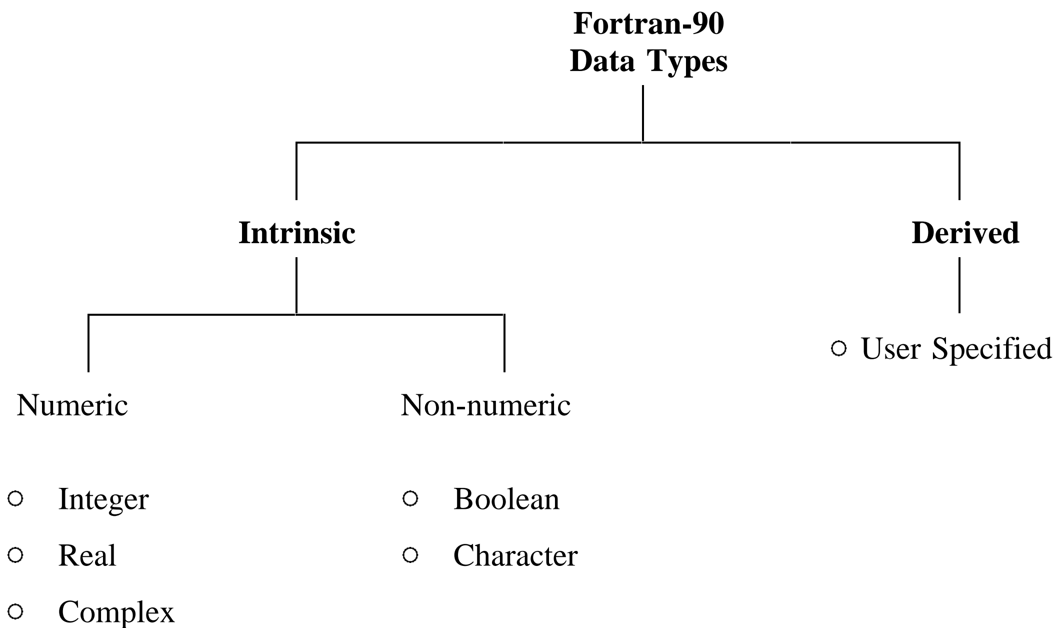
Example:

```
write(*,*)" CSC-302-051 Introduction to Numerical Methods "

write(*,*)" CSC-302-051 Introduction to &
          & Numerical Methods "
```

3.0 Data Types

FORTRAN-77 included five intrinsic data types. Fortran-90 supports these five and allows the programmer to derive a new data type from these intrinsic types:



Intrinsic Data Type

Fortran requires identifiers to be declared a type. This is performed in the declaration part of the program with type declarations REAL, INTEGER, COMPLEX, CHARACTER, and LOGICAL followed by the list of identifiers:

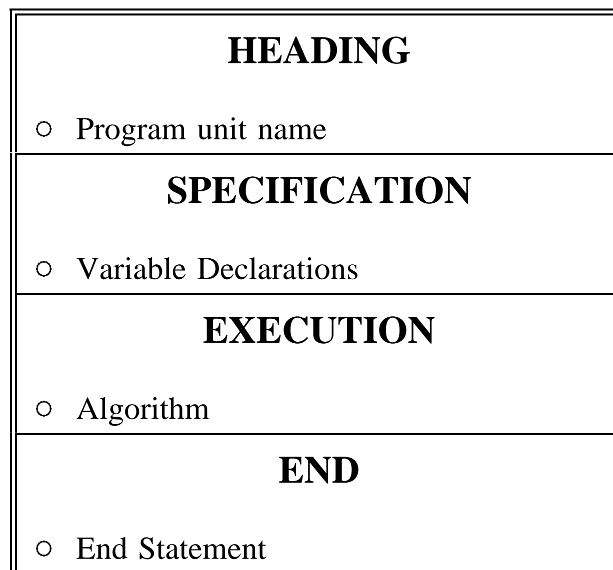
```
REAL                  :: AREA, PI, DIAMETER
INTEGER               :: NUMBER_OF_STUDENTS
CHARACTER(LEN=24)     :: STUDENT_NAME
```

The "LEN=" attribute to the type CHARACTER specifies the character string length. The double colon (::) is a Fortran 90 feature which allows for declaring multiple attributes to identifiers. The attributes are on the left of the double colon (::) and the identifiers are on the right.

It is imperative to declare all variables and avoid default type (implicit) declarations. Use **IMPLICIT NONE** in the declaration part of the program to enforce type declaration.

4.0 Code Structure

A Fortran code consists of program units of which one and only is called the PROGRAM unit. Modular components include the Program, Subroutines, Functions, and Modules. Program units may consist of data and algorithms. Each program unit is structured as follows:



Program Unit Composition

Program Heading

- The FORTRAN program heading has the form

PROGRAM *name*

name is a legal FORTRAN identifier

This is the first statement in a Fortran program.

Specification

The specification part of the program must appear next.

- REAL, INTEGER, CHARACTER, COMPLEX, etc type declarations statements

Execution Statements

Statements in FORTRAN are classified as **executable** or **non-executable**

Non-executable Statements provide information to be used during compilation of the program. These do not cause any specific action to be performed during program execution.

REAL, TYPE, PROGRAM, etc.

Executable Statements specify actions to be taken during program execution

Executable statements are placed in the **Execution Part** of the program following the declaration part.

END and STOP Statements

- END Statement

  The END statement is the last statement in every FORTRAN program.

  The END statement terminates execution and is an executable statement. Only one END statement per program unit is allowed.

  Syntax: **END** or **END** *program_unit name*

  where *program_unit* is the Fortran program or subprogram and *name* is the name of the program unit:

  end program example

- STOP Statement

  Allows for the program execution to stop prior to terminating execution with the END statement.

  Syntax: **STOP**

PAUSE Routine

Allows for program execution to be interrupted and re-started by referencing a routine supplied by the programmer.

Syntax: **CALL PAUSE**

This routine must be a user provided. See the appendices.

Example

A simple program in Fortran 90 and C/C++ follows:

Fortran 90

```
program hello_world
implicit none
write(*,*) " Hello World of Mine "
end program hello_world
```

C++

```
#include <stdio.h>
#include <iostream.h>
main ()
{
cout  << "Hello World of Mine" << endl ;
}
```

5.0 Math Operations

FORTRAN arithmetic operators are given by + - * / **

C/C++ programmers should note that Fortran supports exponentiation (**).

Assignment Operator

The assignment statement (=) is used to assign values to variables:

variable = expression

The expression may be a constant, variable, or formula. For example,

PI = 3.1416

CAREA = PI * R**2

The assignment associates a memory location with the variable. The assignment statement first evaluates the expression (right-hand side) and then associates results with the variable.

Mixed-mode Assignments

A reminder that when variables of the same type are arithmetically combined, the result preserves the type:

9.0 / 4.0 = 2.25	(Real)	1.0 / 2.0 = 0.50	(Real)
9 / 4 = 2	(Integer)	1/ 2 = 0	(Integer)

An integer quantity combined with a real quantity causes the integer to be converted to real and the result will be real

Example of Exponent Trouble

Avoid raising a base to a floating point value unless that is exactly what is needed and the base will always be greater than 0.

2.0 ** 3 = 2.0 * 2.0 * 2.0 = 8.0

2.0 ** 3.0 = $e^{3.0 \ln (2.0)}$ = 8.0

(-4.0) ** 2 = (-4.0) * (-4.0) = 16.0

(-4.0) ** 2.0 = $e^{2.0 \ln (-4.0)}$ = UNDEFINED

Priority Rules

Expressions containing the arithmetic operations (+ - / * and **) are evaluated following the FORTRAN priority rules which are identical to c++ with the addition of exponentiation:

1. All exponentiations are done first; consecutive exponentiations are performed from right to left.
2. All multiplications and divisions are performed next in the order in which they appear left to right.
3. Addition and subtraction are performed last in the order in which they

Examples:

2 * 3 * * 2 = 2 * * 9 = 1 8

2 + 4 * *2 / 2 = 2 + 16 / 2 = 2 + 8 = 1 0

The standard order of evaluation can be overridden using parentheses. Nested parentheses are evaluated from the inner-most set. Parentheses must balance; that is, they must occur in pairs.

<u>Accumulating and Counting</u>

The assignment statement may be very useful in accumulating results (summation and product) or counting. For example,

$$\sum_{i=m}^{n} x_i = x_m + x_{m+1} + \ldots + x_n$$

pseudocode

```
for i = m to n do
 sum ← sum + x_i
end do
```

Fortran-90

```
sum = 0.0
do i = m,n
 sum = sum + x(i)
end do
```

C++

```
sum = 0.0 ;
for(int i=m; i<=n; i++)
        sum + = x[i] ;
```

Product

$$\prod_{k=m}^{n} x_k = x_m \cdot x_{m+1} \cdot x_n$$

pseudocode

```
for k = m to n do
prod ← prod * x_k
end do
```

Fortran-90

```
prod = 1.0
do k = m,n
prod = prod * x(k)
end do
```

C++

```
prod = 1.0 ;
for(int i=m; i<n; I++)
        sum + = x[i] ;
```

for $n < m$ then product = 1.0

Nested Polynomial Multiplication

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1} + a_n x^n$$

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + x(a_n)) \ldots))$$

- Example

$$\begin{aligned} p(x) &= 5 + 3x + 1x^2 + 9x^3 + 2x^4 + 6x^5 \\ &= 5 + x(3 + 1x + 9x^2 + 2x^3 + 6x^4) \\ &= 5 + x(3 + x(1 + 9x + 2x^2 + 6x^3)) \\ &= 5 + x(3 + x(1 + x(9 + 2x + 6x^2))) \\ &= 5 + x(3 + x(1 + x(9 + x(2 + 6x)))) \\ &= 5 + x(3 + x(1 + x(9 + x(2 + x(6))))) \end{aligned}$$

pseudocode	**Fortran-90**	**C++**
$p \leftarrow a_n$	p = a(n)	p = a(n) :
for i = n-1 to 0 step -1 do	do i = n-1,0,-1	for (int = n-1; i>0; i--)
$p \leftarrow a_i + xp$	p = a(i) + x * p	p = a[i]+x*p ;
end do	end do	

6.0 List Directed Input and Output

A program's ability to communicate with a user is critical. Fortran supports I/O functions in two forms: formatted and unformatted. The simplest to use is unformatted I/O and is called list directed.

- Input — FORTRAN source code allows for **input** data to be received interactively or by an input file. Simplest for is the **list** directed output (no format):

  READ*, *list* or **READ(*,*)** ***list***

  READ*, RADIUS or **READ(*,*) RADIUS**

- Output — FORTRAN source code allows for **output** data to be transmitted interactively or by an input file. Simplest for is the **list** directed output (no format)

  PRINT*, *list* or **WRITE(*,*)** ***list***

  PRINT*, ' AREA OF A CIRCLE IS ', CAREA

  WRITE(*,*) ' AREA OF A CIRCLE IS ', CAREA

○ READ(* , *) and WRITE(* , *) Statements

- (* , *)

  First * is the unit number and the second * is the format specifier or label number for the format specification

7.0 Intrinsic Functions

Intrinsic functions are available to a program by referencing the function's name. The function returns its results for the given argument and this result may be assigned to a variable:

Some Fortran 90 Intrinsic Functions

Function	Description	Argument	Value
ABS(x)	Absolute value of x	Integer or Real	Same as argument
SIGN(x,y)	|x |times sign of y	x and y of same kind	Same as x
COS(x)	Cosine of x in rads	Real	Real
EXP(x)	Exponential function	Real	Real
INT(x)	Integer part of x	Real	Integer
LOG (x)	Natural log of x	Real	Real
LOG10(x)	Common log of x	Real	Real
MAX(x_1,..,x_n)	Maximum of x_1,..,x_n	Integer or Real	Same as argument
MIN(x_1,..,x_n)	Minimum of x_1,..,x_n	Integer or Real	Same as argument
REAL(x)	Converts x to real	Integer	Real
SIN(x)	Sine of x in rads	Real	Real
SQRT(x)	Square root of x	Real	Real
TAN(x)	Tangent of x rads	Real	Real
random_seed	Provides Seed	None	Internal
random_number	quasi-random number	Internal seed	Real

```
program get_pi
implicit none
real :: pi
pi = 4.0 * atan(1.0) ! Intrinsic Function atan (arc-tangent)
write(*,*) ' PI = ',pi
end program get_pi
```

It is important to note that all trigonometric intrinsic functions such as sine and cosine use radians for angles **not degrees.**

8.0 Structured Programming

Fortran programs may be structured in sequential, selective or repetition forms. Each structure offer powerful tools for solving numerical problems. Logical expressions and variables are essential to advanced methods of control and will be introduced in this section.

Sequential Programming

Each FORTRAN statement is executed exactly one time in the order of appearance. Simple to code and follow-up, however, limited to relative simple programming needs.

Logical Operations

Logical Data Type

There are only two logical constants in FORTRAN and a variable typed LOGICAL may only have one of these two logical constants:

.TRUE.
.FALSE.

Note: The periods are required

Logical Variable Declaration

Logical variables are declared in the specification part of the program unit (i.e. main program or sub-program) just as other types:

LOGICAL :: *name-list*

e.g. LOGICAL :: LOVE, CHECK, SANITY

Logical variables may specify actions to be taken during program execution

Logical Expressions

- Simple Expressions
- Logical Variables
- Logical Constants
- **Relational Expressions**

○ Compound Logical Expressions

Combination of logical expressions using logical operators

○ Relational Operators

expression-1 *relational-operator* expression-2

Relational Operators

Symbol	Meaning
<	Is less than
>	Is greater than
==	Is equal to
<=	Is less than or equal to
>=	Is greater than or equal to
/=	Is not equal to

Examples: X = 4.5 N = 400
(X < 5.0) is .TRUE. (N == -999) is .FALSE.

Logical expressions formed by comparing **real** variables with .EQ. are evaluated as .FALSE. even though the quantities are algebraically equal.

Example:

```
Real :: X
Logical :: EQUALS
Y = X * (1.0/X)
EQUALS = (Y.EQ.1.0)      results is usually .FALSE.
```

- Compound Logical Expressions

  Compound logical expressions are formed by combining logical expressions using logical operators.

Logical Operators
.NOT.
.AND.
.OR.
.EQV.
.NEQV.

For: LOGICAL :: P,Q

.NOT.	.NOT. P	.TRUE. if P is .FALSE. .FALSE. if P is .TRUE.
.AND.	P.AND.Q	Conjunction of P and Q .TRUE. only if both P and Q are .TRUE.
.OR.	P.OR.Q	Disjunction of P and Q .TRUE. if P or Q or both are.TRUE.
.EQV.	P.EQV.Q	Equivalence of P and Q .TRUE. if both P and Q are .TRUE. or .FALSE.
.NEQV.	P.NEQV.Q	Non-equivalence of P and Q .TRUE. if P or Q is .TRUE. and the other is .FALSE.

Logical Truth Tables

P	.NOT. P
.TRUE.	.FALSE.
.FALSE.	.TRUE.

P	Q	P.AND.Q	P.OR.Q	P.EQV.Q	P.NEQV.Q
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

○ Order of Combined Logical Operations and Relational Expressions

1. Relational Operators (<=, >= , == , /= , < , >)
2. .NOT.
3. .AND.
4. .OR.
5. .EQV. (or .NEQV.)

Example:

N = 4
N**2 + 1 > 10 .AND. .NOT. N< 3 .TRUE.

Example:

```
        program decay
!
! Calculate the decay of radioactive material and provide error checks
! for input.
!
        implicit none
        logical :: error
        real :: n,n0,lambda,t
        write(*,*) “ Enter n_0,lambda, and time”
        read(*,*) n0,lambda,t
        error = (lambda.lt.0.0).OR.(t.lt.0.0)
        if(error.eqv..true.)then
          write(*,*)" Error detected in input"
          stop
        end if
        n = n0 * exp(-lambda * t)
        write(*,*)" Radioactive nuclide remaining ",n
        end program decay
```

Selective Programming Structure

This structured programming approach provides two or more paths of program execution. The control scheme is by logical expressions.

IF construct or Block **IF** statement

```
construct_name :    IF ( logical-expression ) THEN
                        statement-sequence
                    END IF          construct_name
```

Examples:

```
IF ( X >= 0 ) THEN              CHECK_X:  IF ( X >= 0 ) THEN
    Y = X**2                                  Y = X**2
    Z = SQRT(X)                               Z = SQRT(X)
END IF                                    END IF      CHECK_X
```

The Logical expression is evaluated first. If it returns .TRUE., then the next two statements are executed. If the logical expression returns .FALSE., then the "nested" expressions are bypassed.

The logical expression in an IF construct must be enclosed in parenthesis. The structure on the right is given the name CHECK_X to easily identify the purpose.

Logical IF Statement

Simplified form of the IF construct (and an old popular one!) for single statement sequences.

IF (*logical-expression*) action-statement

Execution of the IF statement causes evaluation of the logical expression. If the logical expression is .TRUE., the action statement is executed. If .FALSE. , the action statement is not executed.

Examples:

```
REAL :: LAMBDA
IF ( LAMBDA < 0.0 ) LAMBDA = ABS(LAMBDA)
```

IF-THEN-ELSE Constructs

IF constructs also allow for an alternative statement sequence for execution when the logical expression is .FALSE.

```
construct_name :     IF ( logical-expression ) THEN
                     statement-sequence-1
                     ELSE
                     statement-sequence-2
                     END IF               construct_name
```

IF-ELSE-IF Constructs

The IF-ELSE-IF construct supports Multi-Alternative selection structures.

```
IF ( logical-expression-1 ) THEN
      statement-sequence-1
ELSE IF ( logical-expression-2 ) THEN
      statement-sequence-2

            . . .
ELSE
      statement-sequence-n
END IF
```

Example

```
IF ( x < = 0.0 ) THEN
      funct = -x
ELSE IF ( x < 1.0 ) THEN
      funct = x**2
ELSE
      funct = 1.0
END IF
```

Fortran Repetition Structures

The third Fortran control structure is the Repetition Structure which is also called Loops

Loop Control

- Loops controlled by a counter (DO Loops)
- Loops controlled by a logical expression (DOWHILE Loops)

DO Loops

- Loop is executed once for each value of a pre-determined control variable range
- The initial value, the limit, and the step size of the control variable are determined before repetition begins
- These values may not be modified during loop execution

DO Loop Structure

```
DO index = lower_limit,upper_limit, step_size
        { statement sequence}
END DO
```

- The Loop structure automatically increments or decrements the loop index variable

```
sum_do:   do i=10,1,-1                    !
              sum = sum + i**2            ! Named Fortran 90 Do
          end do  sum_do                  ! construct


do while( i < 10 )
      sum = sum + i                   ! Note:  DO WHILE requires
       i = i+ 1                       ! incrementing control index
end do                                !
```

Below are Fortran-90 DO structures with an EXIT. The EXIT statement causes the process to exit the loop and can be very useful.

```
do i=1,100                              !
    if(.not.(sum < = 100.0)) exit       ! Highly desirable EXIT
    sum = sum + i                       ! of DO construct prevents
end do                                  ! infinite loops.  Safer
                                        ! than DO WHILE
```

THINGS TO REMEMBER

- Repetition structures with logic controls may easily become infinite. **Make sure the structure has a finite number of cycles**

EXAMPLE of INFINITE LOOP

```
do while( i > 0)          !
    sum = sum + I         ! Note: > will lead to
    i = i+ 1              ! an infinite loop
 end do                   !
```

- Never change the loop index variable in a counter controlled DO loop

9.0 FUNCTION Program Unit

○ INTRINSIC Functions

Compiler provided, relatively "safe" to use, careful with type of function and type of argument.

Use of a FUNCTION in a fortran statement returns the result of the FUNCTION to the statement.

Example

Pythagoras Theorem

$$C = \sqrt{A^2 + B^2}$$

```
HYPOTENUSE = SQRT(A**2 + B**2)
```

- SQRT is an intrinsic function of type real in this application
- Arguments to the function must also be real
- Execution sequence:

  (B**2 + A**2)
  SQRT
  Assignment

○ External Functions

- Programmer written to meet specific needs (e.g. factorials)
- Some care must be used
- Structure of an external function is similar to program unit

```
type    FUNCTION NAME (d1, d2, .... )
        •
        •
        specification part
        •
        execution part
        •
        RETURN
        END FUNCTION NAME
```

type

Declares the type of the result

Name

It is called the "return variable" and the function must assign the results of the calculation to this variable. This is the only time a program unit name is used in an assignment statement.

d1, d2, ...

These are called dummy or formal arguments which represent the actual arguments (variables) when the function is referenced.

RETURN and END Statements

- RETURN

Functions may have at least one RETURN statement which instructs the sub-program to pass the calculated result to the program unit.

- END Statement

The END statement is the last statement in every FORTRAN program unit. The END statement terminates execution and is an executable statement. Only one END statement per function unit is allowed

The external FUNCTION is referenced by the functions's NAME from the referencing program unit

function_name (actual-argument-list)

- The actual-argument-list contains the actual arguments in the calling program unit.
- The number of actual arguments must be equal to the number of dummy (or formal) of the FUNCTION
- Each actual argument must agree with the type of the dummy argument.
- Corresponding pairs of actual and dummy arguments are associated with the same memory location

C A U T I O N

- Changing the values of dummy arguments within a FUNCTION program unit changes the values of the corresponding actual arguments that are variables.

  The FUNCTION dummy arguments reference the memory storage address of the actual argument

  If the FUNCTION changes the dummy argument, it will inadvertently change the actual argument in the referencing program unit which usually leads to errors. How does the programmer protect the actual argument? This is done by specifying the "intent" of each dummy argument.

Dummy Argument and its INTENT

A dummy argument may be protected by specifying an INTENT(IN) for the dummy argument in the procedure.

Example

```
      real function cube_root(x)
      implicit none
!
! calculates the cube root of a positive real number using logs
!
      real intent(in) :: x            ! protect from changing dummy
                                      ! argument
      real :: log_x                   ! local variable to function
!
      log_x = log(x)
!
      cube_root = exp(log_x / 3.0)    ! the result variable cube_root
!
      end function cube_root
```

In this example:

- Local variable is not accessible outside of the FUNCTION CUBE_ROOT
- INTENT(IN) tells compiler X can not be changed by the procedure
- Variable name CUBE_ROOT is same as function name. Special variable called the result variable
- The type of the function program unit is declared BOTH in the referencing program unit and in the function.

Example

```
        program example
        implicit none
!
! uses function cube_root
!
        real, external :: cube_root
        real :: x , a
!
                    o
!
        a = cube_root(x)
!
                    o
        end program example
!
        real function cube_root(x)
        implicit none
!
! calculates the cube root of a positive real number using logs
!
        real intent(in) :: x            ! protect from changing dummy
                                        ! argument
        real :: log_x                   ! local variable to function
!
        log_x = log(x)
!
        cube_root = exp(log_x / 3.0)    ! the result variable cube_root
!
        end function cube_root
```

Example

```
      real function fact(m)
      implicit none
!
! Calculates the factorial of an integer
!
      integer :: j                ! Use j as local variable
      integer, intent(in) :: m
      real :: factorial
!
      j = m
      factorial = 1.0  !   Initialize here to reset value on each reference
!
      if(j < 0) then
            write(*,*) " Error integer", j , "must be positive"
            stop
      else if(j == 0) then
            fact = 1.0
            return
     else
       do while(j > 0)
          factorial = factorial * j
          j = j - 1
       end do
     end if
!
     fact = factorial
!
     return
     end function fact
```

10.0 Subroutine Program Unit

A SUBROUTINE differs from a FUNCTION in how it is referenced and how the results (if any) are returned. A FUNCTION is referenced simply by using its name followed by the arguments (if any) in parenthesis. A SUBROUTINE is accessed by means of a CALL statement followed by the name of the subroutine and the arguments (if any) in parenthesis.

Unlike a FUNCTION a SUBROUTINE does not need to return anything to the referencing program unit. If the SUBROUTINE does return results it does so by means of one or more arguments.

Subroutine Program Unit Structure

SUBROUTINE *name (dummy-argument-list)*
○
Declaration Part
○
Execution Part
○
END SUBROUTINE name

name is a legal FORTRAN identifier. The name should be distinct from all other names in the program and should be chosen to indicate the purpose of the subroutine

Dummy-argument-list or Dummy arguments are used to "associate" values to and from the subroutine.

If there are no formal arguments, the parentheses may be omitted

Documentation should follow the SUBROUTINE as comment lines.

<u>Specification</u>

This section includes **non-executable** statements. These provide information to be used during compilation of the program.

- REAL, INTEGER, CHARACTER, COMPLEX, etc
- EXTERNAL, INTENT(IN)

Execution Statements

Statements in FORTRAN are classified as **executable** or **non-executable**

- Executable Statements

  Specify actions to be taken during program execution

RETURN and END Statements

- RETURN

  Subroutines have at least one RETURN statement which instructs the sub-program to pass the calculated values to the main program unit

  Syntax: **RETURN**

  Note: Declared Obsolete in Fortran-90

- END Statement

  The END statement is the last statement in every FORTRAN subroutine. IT IS REQUIRED.

  The END statement terminates execution and is an executable statement. Only one END statement per subroutine unit is allowed

  Syntax: **END**

A subroutine is referenced by the **CALL** Statement

CALL *name (actual-argument-list)*

- The actual-argument-list contains the actual arguments in the referencing program unit.
- The number of actual arguments must be equal to the number of dummy arguments.
- Each actual argument must agree with in type with the dummy argument.

- Corresponding pairs of actual and dummy arguments are associated with the same memory location

- Declaring an dummy argument with INTENT(IN) protects it from being modified.

- Changing the values of dummy arguments within a sub-program changes the values of the corresponding actual arguments that are variables.

INTENT(*option*)

Attribute used ONLY in the declaration of dummy arguments in FUNCTIONS and SUBROUTINES

INTENT(IN)

As with a FUNCTION variables with attribute INTENT(IN) means the variables are used to transfer information to the FUNCTION or SUBROUTINE

INTENT(OUT)

A FUNCTION does not have any variables with this attribute since a FUNCTION does not return "arguments"

A SUBROUTINE variable with attribute INTENT(OUT) indicates that these variables are used to transfer information from the SUBROUTINE back to the referencing program

INTENT(INOUT)

Implies dummy argument may be used for information transmission in both directions.

```
        PROGRAM SUBROUTINE_EXAMPLE
!
        IMPLICIT NONE
        INTEGER :: N=0
        REAL :: FACTORIAL
!
        WRITE(*,*)' Factorial Program, enter an integer  ==> '
        READ(*,*)N
!
        CALL FACT(N,FACTORIAL)      ! Reference SUBROUTINE using CALL
!
        WRITE(*,*)' Factorial of ',N,' is ',FACTORIAL
!
        STOP
        END
!
!*********** subroutine fact ************
!
        SUBROUTINE FACT(M,F)
!
      IMPLICIT NONE
        INTEGER :: J
        INTEGER, INTENT(IN) :: M
        REAL, INTENT(OUT) :: F
!
        J = M
        F = 1.0  ! Initialize here to reset value on each call
        FACTORIALS:       IF(J.LT.0)THEN
                                  STOP
                          ELSE IF(J.EQ.0)THEN
                                  F = 1.0
                          ELSE
                                  DO WHILE(J.GT.0)
                                          F = F * J
                                          J = J - 1
                                  END DO
                          END IF  FACTORIALS
!
      END SUBROUTINE FACT
```

11.0 Formatted Input and Output

Fortran formatted I/O with the READ and WRITE statements allows for programmer specified formats and file access.

READ(*,*) and WRITE(*,*) Statements

- (* , *)
  - UNIT = * (first *)

    The * implies default I/O unit number - Typically the CRT monitor

  - Format Specifier = * (second *)

    The * implies compiler default format for I/O operations

Formatted Output

PRINT *format-specifier, output-list*

WRITE (*unit-number*, *format-specifier*) *output-listing*

The *format specifier* specifies format for displaying expressions in the output list:

- An asterisk (*)
- A character constant or variable specifying format for the output
  ' (list of format descriptors) '
- The label number of a FORMAT statement

  FORMAT (list of format descriptors)

Examples:

```
        PRINT*, NUMBER,TEMP

        PRINT "(TR1,I5,F8.2)"  NUMBER,TEMP


        PRINT 20, NUMBER, TEMP

20      FORMAT(TR1,I5,F8.2)

        WRITE(*,*) NUMBER,TEMP

        WRITE(*, "(TR1,I5,F8.2)" ) NUMBER,TEMP


        WRITE(*,20) NUMBER,TEMP

20      FORMAT(TR1,I5,F8.2)
```

Format Descriptors

Format descriptors (TR1, I5, and F8.2) specify the format in which values of variables in the output-list are displayed.

Example: For NUMBER = 17 and TEMP = 10.25

				1	7				1	0	.	2	5
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Summary of FORMAT Descriptors

Format Descriptor	Intended Use
$_r\mathrm{I}_w$	Integer Data
$_r\mathrm{F}_{w.d}$	Real Data in decimal notation
$_r\mathrm{E}_{w.d}$ & $_r\mathrm{D}_{w.d}$	Real Data in Engineering Notion (single and double precision)
$_r\mathrm{G}_{w.d}$	For E or F I/O depending on the value of the variable
$_r\mathrm{A}_n$	Character Data
$_r\mathrm{L}_w$	Logical Data
T_c	Tab Descriptors
/	Vertical spacing
:	Format scanning control
S	Sign descriptors
$_k\mathrm{PE}$	Scale Factor for scientific notation
BN	Blanks

- Formatted READ is often needed for reading the formatted output of another code

12. Structured Data Types

A collection of data values can be efficiently processed using structured **data type** such as arrays. Arrays are very useful for processing a list of numerical, character, or logical variables and for numerical linear algebra.

1-D Arrays

A list of values stored in memory allow for fast and efficient data retrieval. The list may be organized in memory using an ARRAY data type which groups a fixed number of data items of the same data type in a sequence. Each data item in an array can be directly accessed from memory by specifying the position in the sequence.

Example:

```
REAL VALUE(5)
READ(*,*)    VALUE(1), VALUE(2),..., VALUE(5)
```

VALUE(1)
VALUE(2)
VALUE(3)
VALUE(4)
VALUE(5)

1-D Array Declaration

The name and the subscript range of each one-dimensional array may be declared in two ways:

- **DIMENSION** *array-name(l:u)*

DIMENSION *array-name(l:u)* or **DIMENSION** *array-name (:)*

Fortran-90 enhances the classic array declaration with help of the double colon (::) operator.

```
REAL, DIMENSION(10) :: A
```

The **extent** of the array must be an integer value or integer parameter

```
REAL :: A
DIMENSION A(100)
```

and

```
INTEGER :: LIMIT
PARAMETER (LIMIT=100)
REAL :: A(LIMIT)
```

These methods of declaring the array type and size require the number of elements to be know prior to run time

- **DIMENSION** ***array-name (:)***

Fortran 90 allows the size or dimension of the array to be specified at run-time as:

ALLOCATABLE :: DIMENSION *array-name (:)*

This is a Fortran-90 declaration for an array. The size is determined during run time. These are called ALLOCATABLE arrays since the size is ALLOCATED at run time.

A program unit declares an allocatable array named NUMBERS as follows:

REAL, **ALLOCATABLE** :: NUMBERS(:)

Once the array dimension is know during program execution the size is allocated as follows:

ALLOCATE (NUMBERS(N))

- **Alternate Method for Array Declaration**

type :: array_name(extent)

real :: a(0:10) declares an array “a” with an extent of 11 units of type real storage.

<u>Subscripted Variables</u>

Each individual element of an array is uniquely identified and accessed by means of a subscripted variable

array_variable-name(subscript-identifier)

The subscript identifier is an integer value or variable:

- VALUE(1)
- VALUE(i)

VALUE(1)
VALUE(2)
VALUE(3)
VALUE(4)
VALUE(5)

<u>I/O and 1-D Arrays</u>

Initializing a 1-D array is efficiently performed with a DO LOOP

<u>Interactively initializing a 1-D array</u>

```
      INTEGER, PARAMETER :: LIMIT=50
      REAL :: AXIS(LIMIT)
!
      DO I=1,LIMIT
      READ(*,*) AXIS(I)
      END DO
```

<u>IMPLIED DO LOOP</u>

```
!
      READ(*,*) (AXIS(I), I=1,LIMIT)    ! <==== NICE
!
```

```
      program allocatable_array !
!
! Introduce the concept of allocatable storage for a
! 1-D array for storing a set of observables and calculating
! the average value of the observables.
!
      real, allocatable :: numbers(:) !
      real sum, avg !
      integer n !
!
      sum = 0.0 ; avg = 0.0 !
!
      write(*,"(a)", advance ="no")' How many observables in the set?  '!
      read(*,*) n !
!
      allocate (numbers(n)) !
!
      write(*,"(a)",advance="yes")' Enter the numerical values: '
      do i=1,n !
            read(*,*) numbers(i) !
            sum = sum + numbers(i) !
      end do !
!
      avg = sum/n
!
      write(*,*)' Avg = ', avg
!
      end program allocatable_array
```

```
      program array_demo
!
      implicit none
!
      real, allocatable :: elements(:)
      real :: sum,average
      integer :: i,j,ii
!
      write(*,*)' Enter the array''s starting and ending points '
      read(*,*) i,j
!
      allocate(elements(i:j))
!
      write(*,*)' Enter the values for elements '
      read(*,*) (elements(ii), ii = i,j)
!
      do ii=i,j,1
              sum = sum + elements(ii)
      end do
!
      average = sum/(abs(i)+abs(j)+1.0)
!
      write(*,'(//)')
      write(*,*)' The following values have been entered for elements'
      write(*,'(//)')
!
      do ii=i,j,1
              write(*,10) ii,elements(ii)
      end do
      write(*,'(//A,f8.3)')' The average is ',average
!
   10 format(tr2,'Element ',i2,' is ',f8.4)
!
      end
```

Multidimensional Arrays

Arrays are FORTRAN's version of "structured" data types. Fortran-90 allows up-to seven (7) dimensions which refered to as a **rank** seven array. The shape of the array is determined from the rank and extent.

2-D Array Declaration

The name and the subscript range of each two-dimensional array may be declared in two ways:

DIMENSION *array-name(* l_1:u_1 , l_2:u_2 *)*

Example:

```
REAL, DIMENSION :: FLUX(1:2000,1:1000)

REAL, DIMENSION :: FLUX(2000,1000)
```

The second and prefered way to declare an array is to subscript the variable when declaring the type

```
REAL :: FLUX(1:2000,1:1000)

REAL :: FLUX(2000,1000)
```

- The range or "dimension" of the array must be an integer value or integer parameter

- Memory locations for the data type items are "reserved" according to the dimension of the array

  REAL,DIMENSION :: A(100,100)

  Array A has reserved 10,000 floating point (real) words in memory

- FORTRAN-77 does not allow for dynamic memory allocation for arrays, but FORTRAN-90 does.

- Care is needed to only "reserve" the memory needed

Subscripted Variables

Each individual element of a 2-D array is uniquely identified and accessed by means of a subscripted variable:

variable-name (row-subscript-id, column-subscript-id)

The subscript identifiers are integer values or variables:

VALUE(1, 2) VALUE(i, j)

VALUE (I , J) refers to array element (and contents) in the Ith row and J^{th} column

Each element of the 2-D array can be accessed directly by using a two-subscripted variable consisting of the array name and the desired element subscript

INTEGER IMORN (2 , 3)

11	12	13
21	22	23

IMORN(1,3) = 13 IMORN(2,3) = 23

Two-Dimensional arrays suggests two natural orders for processing the data entries:

- Row wise
- Column wise

The Fortran default processing order is column wise which is different from c/c++ which uses row wise processing.

Row wise processing:

11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

Column-wise processing:

11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

Example: Desired array initialization order:

33.0	41.0	1.2
11.0	12.3	98.0
23.0	14.0	21.0

<u>Desired Order for entering data at the keyboard:</u>

33.0, 41.0, 1.2, 11.0, 12.3, 98.0, 23.0, 14.0, 21.0

Method required: Row-wise Processing

<u>Desired Order for entering data at the keyboard:</u>

33.0, 11.0, 23.0, 41.0, 12.3, 14.0, 1.2, 98.0, 21.0

Method required: Column wise processing (Fortran default)

The programmer can select the processing order by controlling the use and order of subscripts

The FORTRAN **default** convention is that 2-D arrays will be processed column-wise

COLUMN WISE PROCESSING

The **first** subscript is varied **first** and the second subscript varies second

ROW WISE PROCESSING

The **second** subscript is varied **first** and the first subscript varies second

I/O and 2-D Arrays

Initializing a 2-D array is efficiently performed with a DO LOOP

Interactively initializing a 2-D array

Desired array initialization order:

33.0	41.0	1.2
11.0	12.3	98.0
23.0	14.0	21.0

```
        REAL :: DEMO(3,3)                       User must enter:
!
!       ROW WISE PROCESSING:                        33.0
!       Nested Do Loops                             41.0
!                                                   1.2
        DO I=1,3                                    11.0
                DO J=1,3                            12.3
                READ(*,*) AXIS(I,J)                 98.0
                END DO                              23.0
        END DO                                      14.0
```

to obtain:

33.0	41.0	1.2
11.0	12.3	98.0
23.0	14.0	21.0

Use:

Row-Wise Processing Order

ROW WISE PROCESSING

The **second** subscript is varied **first** and the first subscript varies second

```
          REAL AXIS(3,3)
!
          DO I=1,3
                DO 20 J=1,3
                      READ(*,*) AXIS(I,J)
                END DO
          END DO
```

When I = 1 (First Row)

```
        DO J = 1, 3
        READ(*,*) AXIS(1,J)
        END DO
```

When I = 2 (Second Row)

```
        DO J = 1, 3
        READ(*,*) AXIS(2,J)
        END DO
```

When I = 3 (Second Row)

```
        DO J = 1, 3
              READ(*,*) AXIS(3,J)
        END DO
```

Desired array initialization order:

33.0	41.0	1.2
11.0	12.3	98.0
23.0	14.0	21.0

```
        REAL AXIS(3,3)                        User must enter:   33.0
!                                                                11.0
!       COLUMN WISE PROCESSING:                              23.0
!                                                                41.0
        DO J=1,3                                                 12.3
         DO I=1,3                                                14.0
                READ(*,*) AXIS(I,J)                              1.2
          END DO                                                 98.0
        END DO                                                   21.0
```

COLUMN WISE PROCESSING

The **first** subscript is varied **first** and the second subscript varies second

```
REAL :: AXIS(3,3)
DO J=1,3
        DO I=1,3
                READ(*,*) AXIS(I,J)
        END DO
END DO
```

When J = 1 (Column 1)

```
        DO I = 1, 3
        READ(*,*) AXIS(I,1)
        END DO
```

When J = 2 (Column 2)

```
        DO I = 1, 3
        READ(*,*) AXIS(I,2)
        END DO
```

```
When J = 3 (Column 3)
    DO I = 1, 3
        READ(*,*) AXIS(I,3)
    END DO
```

- It must be noticed that the READ statement is encountered nine (9) times in the nested do loop structure above.
- This means the data values must also be entered on nine separate lines.
- If the data is in a file, the file structure for the column processing is, for example:

```
33.0
11.0
23.0
41.0
12.3
14.0
 1.2
98.0
21.0
```

- The same limitation occurs with the WRITE statement

<u>I/O Using Implied Do Loops</u>

AXIS (ROW, COLUMN)

- Row-wise order READ(*,*) ((AXIS(I,J), J=1,COLUMN), I=1,ROW),
- Column-wise order READ(*,*) ((AXIS(I,J), I=1,ROW), J=1,COLUMN)

> NOTE:
>
> The READ statement is only encountered once per loop and all the data for that loop can be on the same line

FILE:

```
33.0    41.0    1.2
11.0    12.3    98.0
23.0    14.0    21.0
```

READ(10,*) ((A(**I**,**J**), **J**=1,3),**I**=1,3)

A(**I**,**J**)

33.0	41.0	1.2
11.0	12.3	98.0
23.0	14.0	21.0

READ(10,*) ((A(**I**,**J**), **I**=1,3),**J**=1,3)

A(**I**,**J**)

33.0	11.0	23.0
41.0	12.3	14.0
1.2	98.0	21.0

Important to Remember

The FORTRAN **default** convention is that 2-D arrays will be processed column-wise

ROW WISE PROCESSING

The **second** subscript is varied **first** and the first subscript varies second

COLUMN WISE PROCESSING

The **first** subscript is varied **first** and the second subscript varies second

Appendix A

Fortran 90 Compiler Available at NCSU

Numerical Algorithms Group's (NAG) Fortran-90 Compiler

A third party Fortran-90 compiler was installed on the eos computing environment in early 1996. Since this is not a native compiler to the workstation, the software most be added to the session:

```
eos%> add nagf90
Numerical Algorithms Group (NAG) "NAGWare f90 Rel.2.1 " Fortran-90 Compiler.
eos%> f90 filename.for {options ...}
```

Above will produce a file named *a.out* which is an executable program

Appendix B

Derived Data Type

This section is presented only because the introduction of derived data type was an important addition to the Fortran language.

Fortran-90 allows for user defined data types that are derived from intrinsic data types providing considerable flexibility in defining data structures.

The programmer defines a derived data type using the new Fortran-90 ***type*** statement in the declaration part of the program. The general form is as follows:

```
TYPE :: derived_name
     intrinsic type declaration :: symbolic_name
                    { ... :: ... }
END TYPE derived_name
```

A variable is declared a derived data type by using the ***type*** statement as follows in the declaration part of the program unit.

```
TYPE(derived_name) :: variable_name
```

The variable has been declared type *derived_name* and is composed of a number of intrinsic data types. It is important to notice the sequential order of the intrinsic data types and to use the % symbol to separate these intrinsic components. The following example demonstrate initializing variable:

```
variable_name % symbolic_name = constant
```

Example

```
!      declare a derived data type

       type :: periodic_table !
               character(len=12) :: element_name !
               real              :: atomic_mass !
               integer           :: atomic_number !
               logical           :: fissile !
       end type periodic_table !
```

```
!     declare a variable to be declared with the derived data type

      type(periodic_table) :: elements(106)  !  Array of the
                                             !  elements
!
!     initialize one a array element for the element Uranium
!
      elements(92) % element_name = "Uranium" !
      elements(92) % atomic_mass  = 238.05 !
      elements(92) % atomic_number= 92 !
      elements(92) % fissile      = .false. !
!
```

Appendix C

Fortran-90 Pause Routine

```
SUBROUTINE F90_PAUSE     !
!
!    subroutine to pause program; requiring user to re-initiate !
     execution
!
IMPLICIT NONE            !
CHARACTER :: PAUSE       !
!
WRITE(*,'(A)', ADVANCE = 'NO') 'Press < ENTER > to continue ==>' !
READ(*,'(A)') PAUSE      !
!
END SUBROUTINE F90_PAUSE !
```