Fortran Subprograms

If Fortran is the lingua franca, then certainly it must be true that BASIC is the lingua playpen

Thomas E. Kurtz
Co-Designer of the BASIC language

Functions and Subroutines

- **Fortran has two types of subprograms, functions and subroutines.**
- **A Fortran function is a function like those in C/C++. Thus, a *function* returns a computed result via the function name.**
- **If a function does not have to return a function value, use *subroutine*.**

Function Syntax:

- A Fortran function, or function subprogram, has the following syntax:

```
type FUNCTION function-name (arg1, arg2, ..., argn)
    IMPLICIT NONE
    [specification part]
    [execution part]
    [subprogram part]
 END FUNCTION function-name
```

- type is a Fortran type (*e.g.*, **INTEGER**, **REAL**, **LOGICAL**, etc) with or without **KIND**.
- function-name is a Fortran identifier
- arg1, ..., argn are *formal arguments*.

Function Syntax:

- **A function is a self-contained unit that receives some "input" from the outside world via its *formal arguments*, does some computations, and returns the result with the name of the function.**
- **Somewhere in a function there has to be one or more assignment statements like this:**

  `function-name` = *expression*

  where the result of *expression* is saved to the name of the function.
- **Note that `function-name` cannot appear in the right-hand side of any expression.**

Function Syntax:

- In a type specification, formal arguments should have a new attribute `INTENT(IN)`.
- The meaning of `INTENT(IN)` is that the function only takes the value from a formal argument and does not change its content.
- Any statements that can be used in `PROGRAM` can also be used in a `FUNCTION`.

Function Example

- Note that functions can have no formal argument.
- But, () is still required.

Factorial computation

```
INTEGER FUNCTION Factorial(n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   INTEGER :: i, Ans

   Ans = 1
   DO i = 1, n
      Ans = Ans * i
   END DO
   Factorial = Ans
END FUNCTION Factorial
```

Read and return a positive real number

```
REAL FUNCTION GetNumber()
   IMPLICIT NONE
   REAL :: Input_Value
   DO
      WRITE(*,*) 'A positive number: '
      READ(*,*) Input_Value
      IF (Input_Value > 0.0) EXIT
      WRITE(*,*) 'ERROR. try again.'
   END DO
   GetNumber = Input_Value
END FUNCTION GetNumber
```

Common Problems:

forget function type

```
FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

forget `INTENT(IN)` – not an error

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

change `INTENT(IN)` argument

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
      IF (a > b) THEN
         a = a - b
      ELSE
         a = a + b
   END IF
   DoSomthing = SQRT(a*a+b*b)
END FUNCTION DoSomething
```

forget to return a value

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   INTEGER :: c
   c = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

Common Problems:

incorrect use of function name

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomething = a*a + b*b
   DoSomething = SQRT(DoSomething)
END FUNCTION DoSomething
```

only the most recent value is returned

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomething = a*a + b*b
   DoSomething = SQRT(a*a - b*b)
END FUNCTION DoSomething
```

Using Functions

- **The use of a user-defined function is similar to the use of a Fortran intrinsic function.**
- **The following uses function `Factorial(n)` to compute the combinatorial coefficient *C(m,n)* , where `m` and `n` are *actual arguments*:**

```
Cmn = Factorial(m)/(Factorial(n)*Factorial(m-n))
```

- **Note that the combinatorial coefficient is defined as follows, although it is *not* the most efficient way:**

$$C(m,n) = \frac{m!}{n! \times (m-n)!}$$

Argument Association :

- *Argument association* is a way of passing values from actual arguments to formal arguments.
- If an actual argument is an *expression*, it is evaluated and *stored in a temporary location* from which the value is passed to the corresponding formal argument.
- If an actual argument is a *variable*, its value is passed to the corresponding formal argument.
- Constant and `(A)`, where `A` is variable, are considered expressions.

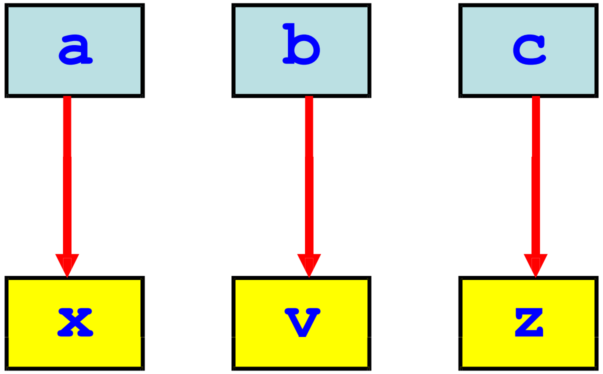
Argument Association : 2/5

- Actual arguments are variables:

```
WRITE(*,*) Sum(a,b,c)



INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……. .
END FUNCTION  Sum
```

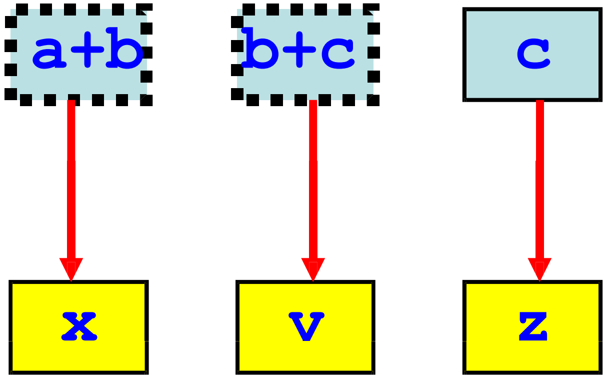


Argument Association : 3/5

- Expressions as actual arguments. Dashed line boxes are temporary locations.

```
WRITE(*,*) Sum(a+b,b+c,c)


INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……. .
END FUNCTION  Sum
```

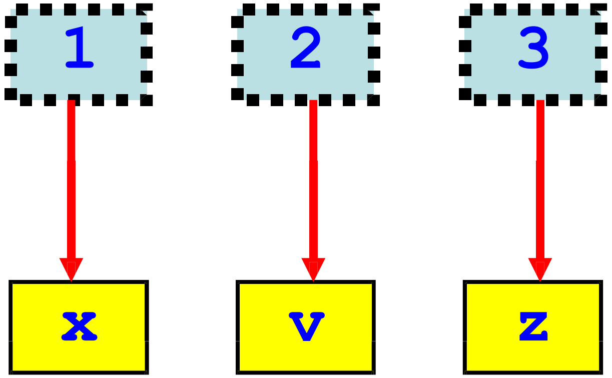


Argument Association :

- **Constants as actual arguments. Dashed line boxes are temporary locations.**

```
WRITE(*,*) Sum(1, 2, 3)


INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……. .
END FUNCTION  Sum
```



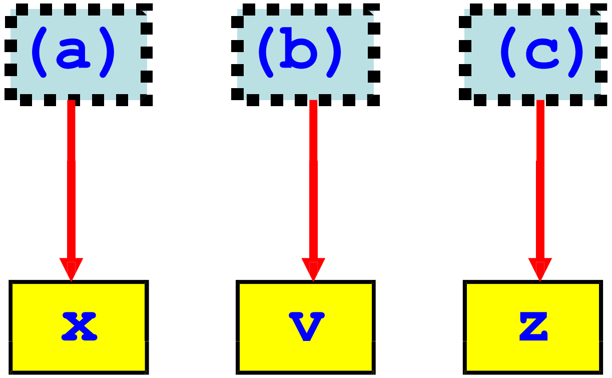
Argument Association :

- A variable in () is considered as an expression. Dashed line boxes are temporary locations.

```
WRITE(*,*) Sum((a), (b), (c))



INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……. .
END FUNCTION  Sum
```



Where Do Functions Go:

- **Fortran functions can be internal or external.**
- ***Internal* functions are inside of a PROGRAM, the *main program*:**

```
PROGRAM program-name
   IMPLICIT NONE
   [specification part]
   [execution part]
CONTAINS
   [functions]
END PROGRAM program-name
```

- **Although a function can contain other functions, internal functions *cannot* have internal functions.**

Where Do Functions Go:

- The right shows two internal functions, `ArithMean()` and `GeoMean()`.
- They take two `REAL` actual arguments and compute and return a `REAL` function value.

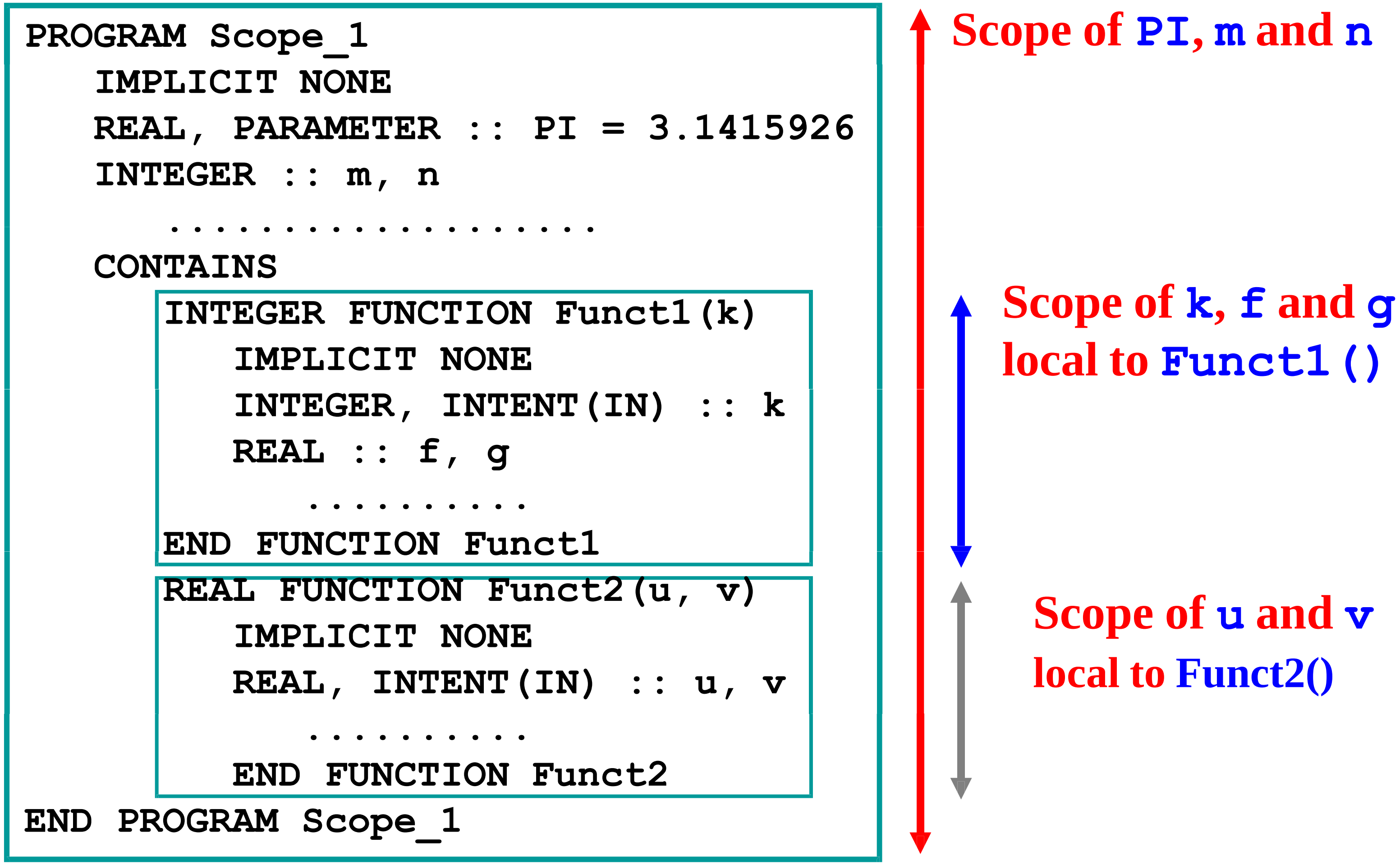
```
PROGRAM TwoFunctions
   IMPLICIT NONE
   REAL :: a, b, A_Mean, G_Mean
   READ(*,*) a, b
   A_Mean = ArithMean(a, b)
   G_Mean = GeoMean(a,b)
   WRITE(*,*) a, b, A_Mean, G_Mean
CONTAINS
   REAL FUNCTION ArithMean(a, b)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a, b
      ArithMean = (a+b)/2.0
   END FUNCTION ArithMean
   REAL FUNCTION GeoMean(a, b)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a, b
      GeoMean = SQRT(a*b)
   END FUNCTION GeoMean
END PROGRAM TwoFunctions
```

Scope Rules:

- ***Scope rules*** **tell us if an entity (*i.e.*, variable, parameter and function) is *visible* or *accessible* at certain places.**
- **Places where an entity can be accessed or visible is referred as the *scope* of that entity.**

Scope Rules:

- ***Scope Rule #1:*** **The scope of an entity is the program or function in which it is declared.**



```
PROGRAM Scope_1
   IMPLICIT NONE
   REAL, PARAMETER :: PI = 3.1415926
   INTEGER :: m, n
      ...................
   CONTAINS
      INTEGER FUNCTION Funct1(k)
         IMPLICIT NONE
         INTEGER, INTENT(IN) :: k
         REAL :: f, g
            ..........
      END FUNCTION Funct1
      REAL FUNCTION Funct2(u, v)
         IMPLICIT NONE
         REAL, INTENT(IN) :: u, v
            ..........
         END FUNCTION Funct2
END PROGRAM Scope_1
```

Scope of `PI`, `m` and `n`

Scope of `k`, `f` and `g` local to `Funct1()`

Scope of `u` and `v` local to Funct2()

Scope Rules:

- *Scope Rule #2* :A global entity is *visible* to all contained functions.

```
PROGRAM Scope_2
   IMPLICIT NONE
   INTEGER :: a = 1, b = 2, c = 3
   WRITE(*,*) Add(a)
   c = 4
   WRITE(*,*) Add(a)
   WRITE(*,*) Mul(b,c)
CONTAINS
   INTEGER FUNCTION Add(q)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: q
      Add = q + c
   END FUNCTION Add
   INTEGER FUNCTION Mul(x, y)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: x, y
      Mul = x * y
   END FUNCTION Mul
END PROGRAM Scope_2
```

- `a` , `b` and `c` are global
- The first `Add(a)` returns `4`
- The second `Add(a)` returns`5`
- `Mul(b,c)` returns`8`

Thus, the two `Add(a)`'s produce different results, even though the formal arguments are the same! This is usually referred to as *side effect*.

Avoid using global entities!

Scope Rules:

- *Scope Rule #2* :A global entity is *visible* to all contained functions.

```
PROGRAM Global
   IMPLICIT NONE
   INTEGER ::  a = 10, b = 20
   WRITE(*,*)  Add(a,b)
   WRITE(*,*)  b
   WRITE(*,*)  Add(a,b)
CONTAINS
   INTEGER FUNCTION Add(x,y)
      IMPLICIT NONE
      INTEGER, INTENT(IN)::x, y
      b    = x+y
      Add = b
   END FUNCTION Add
END PROGRAM Global
```

- The first `Add(a,b)` returns 30
- It also changes `b` to 30
- The 2nd `WRITE(*,*)` shows 30
- The 2nd `Add(a,b)` returns 40
- This is a bad side effect
- **Avoid using global entities!**

Scope Rules:

- ***Scope Rule #3*** **:An entity declared in the scope of another entity is always a different one even if their names are identical.**

```
PROGRAM Scope_3
  IMPLICIT NONE
  INTEGER :: i, Max = 5
  DO i = 1, Max
    Write(*,*) Sum(i)
  END DO
CONTAINS
  INTEGER FUNCTION Sum(n)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    INTEGER :: i, s
    s = 0
    …… other computation ……
    Sum = s
  END FUNCTION Sum
END PROGRAM Scope_3
```

Although `PROGRAM` and `FUNCTION` `Sum()` both have `INTEGER` variable `i`, They are *TWO* different entities.

Hence, any changes to `i` in `Sum()` will not affect the `i` in `PROGRAM`.

Example:

```
PROGRAM HeronFormula
   IMPLICIT NONE
   REAL :: a, b, c, TriangleArea
   DO
      WRITE(*,*) 'Three sides of a triangle please --> '
      READ(*,*) a, b, c
      WRITE(*,*) 'Input sides are ', a, b, c
      IF (TriangleTest(a, b, c)) EXIT ! exit if they form a triangle
      WRITE(*,*) 'Your input CANNOT form a triangle. Try again'
   END DO
   TriangleArea = Area(a, b, c)
   WRITE(*,*) 'Triangle area is ', TriangleArea
CONTAINS
   LOGICAL FUNCTION TriangleTest(a, b, c)
      ......
   END FUNCTION TriangleTest
   REAL FUNCTION Area(a, b, c)
      ......
   END FUNCTION Area
END PROGRAM HeronFormula
```

Subroutines:

- **A Fortran function takes values from its formal arguments, and returns a *single value* with the function name.**
- **A Fortran subroutine takes values from its formal arguments, and *returns some computed results with its formal arguments*.**
- **A Fortran subroutine does not return any value with its name.**

Subroutines:

- **The following is Fortran subroutine syntax:**

```
SUBROUTINE subroutine-name(arg1,arg2,...,argn)
   IMPLICIT NONE
   [specification part]
   [execution part]
   [subprogram part]
END SUBROUTINE subroutine-name
```

- **If a subroutine does not require any formal arguments, “`arg1,arg2,...,argn`” can be removed; however, `()` must be there.**
- **Subroutines are similar to functions.**

The CALL Statement:

- Unlike C/C++ and Java, to use a Fortran subroutine, the `CALL` statement is needed.
- The `CALL` statement may have one of the three forms:
  - `CALL sub-name(arg1,arg2,…,argn)`
  - `CALL sub-name( )`
  - `CALL sub-name`
- The last two forms are equivalent and are for calling a subroutine without formal arguments.

The CALL Statement:

```
PROGRAM Test
   IMPLICIT NONE
   REAL :: a, b
   READ(*,*) a, b
   CALL Swap(a,b)
   WRITE(*,*) a, b
CONTAINS
   SUBROUTINE Swap(x,y)
      IMPLICIT NONE
      REAL, INTENT(INOUT) :: x,y
      REAL :: z
      z = x
      x = y
      y = z
   END SUBROUTINE  Swap
END PROGRAM Test
```

```
PROGRAM SecondDegree
   IMPLICIT NONE
   REAL :: a, b, c, r1, r2
   LOGICAL :: OK
   READ(*,*) a, b, c
   CALL Solver(a,b,c,r1,r2,OK)
   IF (.NOT. OK) THEN
      WRITE(*,*) “No root”
   ELSE
      WRITE(*,*) a, b, c, r1, r2
   END IF
CONTAINS
   SUBROUTINE Solver(a,b,c,x,y,L)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a,b,c
      REAL, INTENT(OUT) :: x, y
      LOGICAL, INTENT(OUT) :: L
      .........
   END SUBROUTINE Solver
END PROGRAM SecondDegree
```