Functions

Functions in Fortran may be defined in few ways, the most general would be

```
type function FunctionName(arg1, arg2, ...) result(variable)

    type, intent(in) :: arg1
    type, intent(in) ::
    ...

    [ body ]

end function FunctionName
```

Above `type` is one of the Fortran data types, e.g. `integer`, `real` etc. or a user defined data type. `function` is the mandatory keyword which defines the function. `FunctionName` is a user defined and arbitrary name, which is different than Fortran keywords. The name is followed by list of arguments `(arg1, arg2, ...)`. Last one is a variable which will be the result of the function `result(variable)`, the `variable` is of the type that is defined at the beginning of the line.

Another definition of a function may look like

```
function FunctionName(arg1, arg2, ...) result(variable)

    type, intent(in) :: arg1
    type, intent(in) :: arg2
    ...

    type :: variable

    [ body ]

end function FunctionName
```

In both cases we have used `intent(in)` to prevent variables from being modified in the body of the function. Intent may take three forms

1. `intent(in)` - variable value is passed to the function, but cannot be changed
2. `intent(out)` - variable is initialized inside of the function of subroutine and its initial value (passed from caller) is ignored

3. `intent(inout)` - the variable enters the function/subroutine with a value and leaves with a value (default).

After the function is defines it can be invoked

```
variable = FunctionName(arg1, arg2, ...)
```

where `variable` is the same type as the function.

For example

```
function SimpleFunction(n) result(total)

  implicit none

  integer, intent(in) :: n
  integer :: total

  total = 2 * n

end function SimpleFunction

program function_example

  implicit none

  integer :: input, output
  integer, external :: SimpleFunction

  input = 3

  output = SimpleFunction(input)

  print *,'Result = ', output

end program function_example
```

Note that the function name is specified in the main program as an `external` variable of `integer` type

```
integer, external :: SimpleFunction
```

without this definition the compiler will return an error of unassigned data type. The above program should return the following output

```
 Result =           6
```

Function can be also defined in the body of the main program. In such a case it does not need to be defined as an `external` , neither defined as a variable at all. For example

```
program simple_function

  implicit none

  integer :: val = -4
  logical :: res = .true.

  res = sign_test(val)

  print *,'Is the value positive ? : ', res

  contains

  logical function sign_test(input) result(output)

    integer, intent(in) :: input

    if( input >= 0 ) then
      output = .true.
    else
      output = .false.
    endif

  end function sign_test

end program simple_function
```

Optional parameters

Fortran allows functions to have optional arguments. Last argument of the function may be defined with parameter `optional` and the function can be called with or without that parameter present. In the body of the function programmer may use function `present(x)` to check if the function has been used with or without the optional parameter. To define argument as optional one need to use

```
type, intent(in), optional :: variable
```

and the function's body may contain a conditional check for that parameter

```
if( present (variable) ) then
    [ body 1 ]
else
    [ body 2 ]
end if
```

An example usage may look like this

```
program function_optional_parameter

  implicit none

  real :: a, x, b

  a = 1.0
  b = 2.0
  x = 1.0

  print *,'Without b : ', linear_function(a,x)
  print *,'With b : ', linear_function(a,x,b)

  contains

  ! y = a*x + b
  real function linear_function(a,x,b) result(y)

    implicit none

    real, intent(in) :: a
    real, intent(in) :: x
    real, intent(in), optional :: b

    if( present(b) ) then
      y = a*x + b
    else
      y = a*x
    end if

  end function linear_function

end program function_optional_parameter
```

with output

```
 Without b :    1.00000000
 With b :    3.00000000
```

Saving variable within a function

Fortran contrary to other languages offers saving a current value in the body of a function. If a function modifies a value of a particular local variable and that variable is initiated with

parameter `save`, its current value will be kept in memory and available at the next invocation of that function. Variable which is to be saved has to be initialized as

```
type, save :: variable
```

and has to be a local variable. The `result` variable cannot be saved. An example program which utilizes the `save` feature may look like this

```
program save_variable

  implicit none

  print *, ' One   : ', CounterFunction()
  print *, ' Two   : ', CounterFunction()
  print *, ' Three : ', CounterFunction()
  print *, ' Four  : ', CounterFunction()

  contains

  integer function CounterFunction() result(num)

    implicit none

    integer, save :: counter = 0

    counter = counter + 1
    num = counter

  end function CounterFunction

end program save_variable
```

In this example we initialized variable `counter` as

```
integer, save :: counter = 0
```

the initial value 0 is only assigned at the first invocation of the function. The variable `counter` is modified at every invocation and its value is incremented by 1. The current value is available next time the function is used. The above program will output the following

```
  One   :            1
  Two   :            2
  Three :            3
  Four  :            4
```

Recursive functions

Recursive function (a function that calls itself) may be also defined in Fortran with `recursive` parameter before the definition of the function, i.e.

```
recursive function FunctionName(arg1, ... ) result(variable)
    [body]
end function FunctionName
```

An example usage may be a factorial function

```
program recursive_factorial

  implicit none
  integer :: m = 3

  print *, 'Factorial ',m,' ! = ', factorial(m)

  contains

  recursive function factorial(m) result (fac)

    implicit none

    integer, intent(in) :: m
    integer :: fac

    if( m == 0 ) then
      fac = 1
    else
      fac = m * factorial(m - 1)
    end if

  end function

end program recursive_factorial
```

which gives

```
 Factorial           3  ! =           6
```

Passing function as an argument

Fortran allows a function to be an argument to another function of subroutine. For that an `interface` construct has to be placed in the body of the function or the subroutine that will take such argument. Interface is used to define the type of the function and all of its arguments and has a general form

```
interface
    real function MyFunction(arg1, arg2, arg3, ...)
        type, intent(in) :: arg1
        type, intent(in) :: arg2
        type, intent(in) :: arg3
        ...
    end function MyFunction
end interface
```

after the interface is defined a function can be used as an argument. For example

```
program function_as_an_argument

  implicit none

  print *, 'Linear function'
  call EvaluateRange(-1.0,1.0,0.2,LinearFunction)

  print *, 'Quadratic function'
  call EvaluateRange(-1.0,1.0,0.2,QuadraticFunction)

  contains

  ! y = a * x
  real function LinearFunction(a,x) result(y)

    implicit none
    real, intent(in) :: a, x

    y = a * x

  end function LinearFunction
```

```
  ! y = a * x^2
  real function QuadraticFunction(a,x) result(y)

    implicit none
    real, intent(in) :: a, x

    y = a * x**2

  end function QuadraticFunction

  ! Evaluate function `my_func` on grid [x0,x1] with step dx
  subroutine EvaluateRange(x0, x1, dx, my_func)

    implicit none
    real,intent(in) :: x0, x1, dx
    real :: a, x

    interface
      real function my_func(a,x)
        real, intent(in) :: a, x
      end function my_func
    end interface

    a = 1.0

    x = x0
    do while (x <= x1)
      print *,x, my_func(a,x)
      x = x + dx
    end do

  end subroutine EvaluateRange

end program function_as_an_argument
```

with output

```
Linear function
 -1.0000000000000000       -1.0000000000000000
-0.80000000000000004      -0.80000000000000004
-0.60000000000000009      -0.60000000000000009
-0.40000000000000008      -0.40000000000000008
-0.20000000000000007      -0.20000000000000007
 -5.5511151231257827E-017  -5.5511151231257827E-017
 0.19999999999999996       0.19999999999999996
 0.39999999999999997       0.39999999999999997
 0.59999999999999998       0.59999999999999998
 0.80000000000000004       0.80000000000000004
  1.0000000000000000        1.0000000000000000
Quadratic function
 -1.0000000000000000        1.0000000000000000
-0.80000000000000004       0.64000000000000012
-0.60000000000000009       0.36000000000000010
-0.40000000000000008       0.16000000000000006
-0.20000000000000007        4.0000000000000029E-002
 -5.5511151231257827E-017   3.0814879110195774E-033
 0.19999999999999996        3.9999999999999980E-002
 0.39999999999999997       0.15999999999999998
 0.59999999999999998       0.35999999999999999
 0.80000000000000004       0.64000000000000012
  1.0000000000000000        1.0000000000000000
```

after being compiled with `gfortran -fdefault-real-8 function-as-an-argument.f90 -o test.x`