Programming Techniques

Lecture 4: programs, modules, subroutines, and functions

Hannu Parviainen

Universidad de la Laguna

September 24, 2025

Fortran 90 Program Units

Main kinds

- **program**: where execution begins; may contain internal procedures via `contains`.
- **module**: container of declarations and procedures; attach with `use`.

Procedures

- **subroutine**: performs a task; invoked with `call`; no return value.
- **function**: like a subroutine but returns a value via its function name.

Main Program: Syntax

Skeleton

```fortran
program Main
  implicit none
  ! declarations
  ! executable statements
contains                  ! optional internal procedures
  ! subroutine ...
  ! function   ...
end program Main
```

Main Program: Minimal Example

Internal function example

```
program main
  implicit none
  real :: x
  read *, x
  print *, floor(x)             ! intrinsic
  print *, negative(x)          ! internal function
contains
  real function negative(a)
    real, intent(in) :: a
    negative = -a
  end function negative
end program main
```

Modules

What modules are

- Libraries of functions, subroutines, derived types, and constants.
- Can live in the same file as the main program, but are typically in their own files.
- Can be precompiled and linked with the main program at compilation time.

```
module mymath
  implicit none
  real, parameter :: pi = 3.14
contains
  real function square(a)
    real, intent(in) :: a
    square = a**2
  end function square
end module mymath

program ex1
  use mymath
  implicit none
  print *, square(2.3), pi
end program ex1
```

Compiling a program that uses a module

Compilation result

- Compiling the example produces the executable `ex1` and a module file `mymath.mod`.
- The `.mod` file plays a role similar to a C/C++ header: it contains interface information for procedures in the module.

```
gfortran -o ex1 ex1.f90
```

```
module mymath
  implicit none
  real, parameter :: pi = 3.14
contains
  real function square(a)
    real, intent(in) :: a
    square = a**2
  end function square
end module mymath

program ex1
  use mymath
  implicit none
  print *, square(2.3), pi
end program ex1
```

Modules in separate files & build options

Two-file layout

`mymath.f90` holds the module; `ex1.f90` holds the program.

Compiling

- Compile the module first, then link the object when building the program:

```
gfortran -c mymath.f90
gfortran -o ex1 ex1.f90 mymath.o
```

- Or compile both source files together:

```
gfortran -o ex1 ex1.f90 mymath.f90
```

- For larger projects, use a **Makefile** (covered next lecture).

Listing 1: mymath.f90

```fortran
module mymath
  implicit none
  real, parameter :: pi = 3.14
contains
  real function square(a)
    real, intent(in) :: a
    square = a**2
  end function square
end module mymath
```

Listing 2: ex1.f90

```fortran
program ex1
  use mymath
  implicit none
  print *, square(2.3), pi
end program ex1
```

Subroutines: Syntax and Example

General form

```
subroutine ProcName(dummy_args)
  ! declare dummy arguments
  ! local declarations
  ! executable statements
end subroutine ProcName
```

Internal subroutine inside a program

```
program Thingy
  implicit none
  call OutputFigures(NumberSet)
contains
  subroutine OutputFigures(Numbers)
    real, dimension(:), intent(in) :: Numbers
    print *, "Here are the figures", Numbers
  end subroutine OutputFigures
end program Thingy
```

Functions: Syntax

Return via result(variable_name)

```
function fun(a, b) result(res)
  real intent(in) :: a, b
  real :: res
  res = sqrt(a**2 + b**2)
end function fun
```

Return via function name

```
real function fun(a, b)
  real intent(in) :: a, b
  fun = sqrt(a**2 + b**2)
end function fun
```

Recursive Functions

Concept

- Functions that call themselves.
- Useful for algorithms defined inductively (e.g. factorial, Fibonacci).
- Must be declared with the keyword `recursive`.
- Must include a termination condition to prevent infinite recursion.

Example

```fortran
recursive function factorial(n) result(f)
  integer, intent(in) :: n
  integer :: f

  if (n <= 1) then
    f = 1
  else
    f = n * factorial(n - 1)
  end if
end function factorial
```

Pure Functions

Concept

- Declared with the keyword `pure`.
- No side effects: cannot alter global variables, perform I/O, or modify their arguments (except via `intent(out)`).
- Always return the same result given the same inputs.
- Safe for parallel execution.

Example

```
pure function fun(x, y) result(r)
  real, intent(in) :: x, y
  real :: r

  r = sqrt(x**2 + y**2)
end function fun
```

Elemental Functions

Concept

- Declared with the keyword `elemental`.
- Applied element-wise to array arguments.
- Arguments must be scalars or conformable arrays.
- Very useful for concise array programming.

Example

```
module test
  implicit none
contains
elemental real function square(x)
    real, intent(in) :: x
    square = x*x
  end function
end module test

program ex
  use test
  implicit none
  real, dimension(3) :: x = [1.0, 2.0, 3.0]
  print *, square(x)
end program ex
```

Dummy Arguments and intent

Intent attributes

- `intent(in)`: argument is read-only inside the procedure.
- `intent(out)`: argument is written (output) by the procedure.
- `intent(inout)`: argument is both read and modified.

These clarify usage and improve compiler diagnostics.

Example

```
subroutine scale(alpha, v)
  real, intent(in)    :: alpha
  real, intent(inout) :: v(:)
  v = alpha * v
end subroutine scale
```

Choosing a Procedure Placement

Internal procedures

- Defined after `contains` in a `program` or `module`.
- Have host association; implicit interface for host variables; great for small helpers tightly coupled to one unit.

Module procedures

- Reusable across translation units via `use`; provide explicit interfaces to callers.
- Control visibility with `public`/`private`.

Exercises: Modules, Subroutines, and Functions

Tasks

1. Write a module `mymath` in a `mymath.f90` file containing:
   - a parameter `pi = 3.14159`,
   - a function `circle_area(r)` that returns the area of a circle.
2. Create a main program in a `test_mymath.f90` file that uses the module and prints the area of a circle with radius 2.0.
3. Compile and link the module and program

Exercises: Recursive Functions

Tasks

1. Write a recursive function `factorial(n)` that computes $n!$.
2. Test the function by computing 5! and 10!.
3. Modify the function to handle invalid inputs (e.g. $n < 0$).
4. Write a recursive function `fibonacci(n)` that returns the nth Fibonacci number.
5. Compare the recursive implementation with an iterative one.