Fortran 90/95/...

- \+ good for numerical problems
  - \+ optimising compilers $\Rightarrow$ efficient code
  - \+ many mathematical functions already included
  - \+ complex arithmetic
  - \+ array operations
  - \+ operators expandable to own data types
- \+ standardised $\Rightarrow$ portability
- \+ good subroutine libraries available (NaG etc.)
- \+ compilers accept also old programs, which are abundant
- \- most of the old programs just horrible
- \- neolithic potholes requiring careful coding

Example:

Source code in a file `addition.f90`:

```
program addition
real x,y,z
write(*,*) 'give two numbers'
read (*,*) x,y
z=x+y
write(*,*) 'sum is',z
end program addition
```

Compilation and execution might be:

```
>lf95 -o add addition.f90
Compiling file addition.f90.
Compiling program unit addition at line 1:
>
>add
give two numbers
1,3
sum is 4.00000000
```

Example: A simple equation solver

Write the equation as $x = f(x)$. For example, $x^5 - x - 1 = 0 \Rightarrow$

$$x = x^5 - 1$$

or

$$x = (1 + x)^{0.2}$$

```
program solve
! find the real root
! the equation x**5-x-1=0
real x0, x1
x0 = 0.5       ! guess initial value
x1=(1.0+x0)**0.2
! iterate until the values do not change
do while (x1.ne.x0)
  x0 = x1
  x1 = (1.0+x0)**0.2
end do
write (6, *) x1, x1**5-x1-1
end
```

```
>lf95 -o solve solve.f90
Compiling file solve.f90.
Compiling program unit solve at line 1:
>./solve
1.16730404 5.03132583E-07
```

Program format

Fixed and free form; cannot be mixed. In F77 fixed form only. In the following only the free form will be discussed.

Lower- and uppercase letters are considered the same (except possibly in file names depending on the operating system). However, use lower- and uppercase letters in a consistent way and write the same name always in the same way to keep the code readable. Abundant use of uppercase letters often makes text difficult to read.

A statement ends at the end of line; no separators are needed.

If a statement continues to several lines, it has to be shown explicitly. An &-sign at the end of line means that the statement will continue on the next line:

```
y = 1.0 + x + 0.5 * x**3 &
  + 1.0/6 * x**4
```

Maximum length of a statement is 40 lines.

An exclamation mark (!) means that the end of the line is a comment that the compiler will just omit. A comment ends automatically at the end of the line; no specific terminator is needed.

Simple variables

Simple variables have an implicit type:

- integer, if the first character is `I–N`
- real otherwise

Dangerous! Misspelled name will mean a different variable:

```
x0=1.0
if (xO.gt.0) ...
```

Declare all variables! Prevent the implicit typing:

```
implicit none
```

Simple types:

```
integer
real
logical  (value is .true. or .false. )
complex
double precision
character
```

Internal representation of these may be different on different computers. Thus even the same program may give different results. (This could be a sign of a unstable algorithm that depends on the least significant bits of the numbers.)

The complete form of a variable declaration is

```
tyyppi (parameters), attributes :: nimi
```

Parameters determine the internal representation of the variable, attributes determine array sizes and other properties related to memory allocation.

In F90 the precision can be defined in a machine independent way. But at the hardware level only a few different types are realised; arbitrarily high precision is not available.

The precision of a variable is given by the kind attribute:

```
integer count
integer (kind=selected_int_kind(5)) :: count
integer (selected_int_kind(5)) :: count
```

Two latter forms declare an integer the representation of which requires at most 5 decimal digits.

The value of the kind attribute is a small integer that will determine which one of the few internal representations will be used. The details of these representations may vary.

⇒

The actual values of the attribute are not specified in the standard.

⇒

Intrinsic functions, like `selected_int_kind` are used to dig up the attribute value (that the programmer doesn't need to know).

```
integer, parameter :: maxn=1000
real, parameter :: pi=3.141592654
```

`maxn` and `pi` are constants that cannot be altered in the program.

```
real (kind=selected_real_kind(5)) :: a, b
real (kind=selected_real_kind(5,20)) :: c
```

Real numbers with an accuracy of 5 decimals; range of `c` is $10^{-20} - 10^{20}$.

```
integer, parameter :: short=selected_int_kind(4), &
              long=selected_int_kind(7), &
              longreal=selected_int_kind(10, 100)
integer (kind=short) :: i,j
integer (kind=long) :: bigint
real (kind=longreal) :: big
```

Constants

Integers

```
123
123_short
1234567_long
```

Real numbers

```
1.5
-1.5
1.5E10
1.5E-10
1.5_longreal
1.5E-10_longreal
```

Assignment operator `=`

```
i=100
x=1.5
```

Expressions

```
1.0+2.0*y/z**2-3.5*(y+x)
```

Normal associativity:

- first ** (raising to a power)
- then * and / from left to right
- finally + and - from left to right
- parentheses ( ) can be used to alter the evaluation order

1) First, the expression on the right hand side of the assignment operator `=` is evaluated,
2) then converted to the type of the variable on the left hand side and
3) finally stored to the variable.

```
real x
x = 1/2   ! x=?
```

Be careful! Division of integers will result in an integer (integral part of the quotient)

If a real value is wanted, write e.g.

```
x=1.0/2
```

Intrinsic functions

Trigonometric functions (angles always in radians!):

```
sin(x), cos(x), tan(x)
asin(x), acos(x), atan(x), atan2(y,x)
```

Hyperbolic functions:

```
sinh(x), cosh(x), tanh(x)
```

Exponent, logarithm etc.

```
exp(x), log(x), log10(x), sqrt(x)
```

Minimum and maximum; arbitrary number of arguments

```
min(x, y, ...), max(x, y, ...)
```

Absolute value

```
abs(x)
```

Example: conversion to spherical coordinates

```
real x,y,z,r,phi,theta
real, parameter :: pi=3.141592654
x=-1.0 ; y=3.0; z=2.0
r=sqrt(x**2+y**2+z**2)
phi=atan2(y,x)*180.0/pi
theta=asin(z/r)*180.0/pi
```

Comparison operators

Two forms, old dotted notation, and new (from F90 onwards) more mathematical notation:

```
==   .eq.
/=   .ne.
<    .lt.
<=   .le.
>    .gt.
>=   .ge.
```

NB: `=` is assignment; comparison is `==`.

```
integer n
logical d
d = n == 100*(n/100)    ! true, if n divisible by 100
d = n .eq. 100*(n/100)
```

Logical constants

```
.true.  .false.
```

Logical operators

```
.and.
.or.
.not.
```

`X.and.Y` is true, if and only if `X==.true.` and `Y==.true.`.

`X.or.Y` is true, if `X==.true.` or `Y==.true.` or both are true.

`.not.X` is true, if `X==.false.`.

Find if the given year y is a leap year:

```
logical leap, d4, d100, d400
integer y
 ...
d4 = y==4*(y/4)
d100 = y==100*(y/100)
d400 = y==400*(y/400)
leap = d4.and.(.not.d100 .or. d400)
```

Basic control strucures: serial code

Usually each statament on a line of its own; no separator is needed:

```
x=1.0
y=exp(-x**2/2)
z=1-y
```

There can be several statements on the same line separated by semicolons:

```
x = 1.0 ;  y = 2.0 ;  z = 0.1
```

Basic control strucures: selection

One alternative

```
if (x > 0.0) y = 1/x

if (x > 0.0 .and. x < 100.0) y=exp(x)

if (x > 0.0) then
  y=1/x
  z=log(x)
end if
```

Statements are executed only if the condition is true, otherwise nothing is done.

Be careful: the following form is not allowed:

```
if (x > 0.0) then y = 1/x
```

Two alternatives:

```
if (x > 0.0) then
  y=1/x
else
  y=0.0
end if
```

Several alternatives

```
if (x > 0.0) then
  y=log(x)
else if (x < 0.0) then
  y=-log(abs(x))
else
  y=0.0
end if
```

Basic control strucures: repetition

Fixed number of iterations:

```
sum=0.0
do i=1,100
  sum=sum+i
end do
```

The default step size of the loop is 1. Other values must be given explicitly

```
sum=0.0
do i=0,100,2  ! sum of even numbers
  sum=sum+i
end do

do i=imin, imax, step  ! variables can be used
  sum = sum+i
end do
```

Repetition as long as a given condition is valid:

```
x=0.2
sum=0.0
term=1.0
do while (term > 0.0001)
  sum = sum+term
  term = term*x
end do
```

`cycle`: return to the beginning of the loop (like continue in C)

```
s=0.0
do i=1,100
  read (5, *) x
  if (x <= 0.0) cycle
  s=s+log(x)
end do
```

$n + 1/2$ cycle loop

Exit the loop by executing an `exit` statement (like break in C)

```
x0 = 0.5
do
 x1=(1.0+x0)**0.2
 ! finish if the accuracy reached
 if (abs(x1-x0) < 0.0001) exit
 x0 = x1
end do

x0 = 0.5
n=0
do
 x1=(1.0+x0)**0.2
 if (abs(x1-x0) < 0.0001) exit
 n=n+1
 if (n > 100) exit
 x0=x1
end do
```

Simple input and output

```
read (unit number, format) list of variables
write (unit number, format) list of variables
open (unit number, file attributes)
close (unit number)
```

Each file has a unique unit number (LUN, logical unit number).

Traditionally 5=card reader (nowadays a terminal), 6=line printer (the same terminal).

Format is a string that defines how the output is formatted. A free form is denoted by `*`.

```
read(5,*) x,y
z=x+y
write (6,*) x,y,z
```