Basic data types

Integer

Integer numbers are defined using an `integer` keyword

```
integer :: a, b
integer, parameter :: c = 1
a = 1
b = 2
```

Real

Real numbers are initialized using keywords `real` and `double precision` to explicily request single and double precision real number.

```
real :: a
double precision :: b
```

Complex

Fortran has a build in complex numbers data type which can be initialized using `complex` keyword

```
complex :: a, b
a = (1.0,0.0)
b = (0.0,1.0)
```

where `a` is just a real number 1.0 in complex representation and `b` is an imaginary unit `i`.

Character

A character variable of single or multiple character lenght `len=` can be initiated using the following

```
character :: a
character(len=100) :: b
```

where variable `a` is a single character and `b` is a string of 100 characters.

Logical

Logical variables can take only two values, i.e. `.true.` and `.false.`. The initialization may look like this

```
logical :: value = .false.
logical :: a
a = .true.
```

The toy program shown below initializes all mentioned here basic data types and performs an assignement of the value for each of them.

```
program data_types

  implicit none

  logical   :: a
  integer   :: b
  real      :: c
  character :: d
  complex   :: e

  print *,'Before assignment'
  print *,'logical   : ',a
  print *,'integer   : ',b
  print *,'real      : ',c
  print *,'character : ',d
  print *,'complex   : ',e

  a = .false.
  b = 1
  c = 3.14
  d = 'a'
  e = (1.0,0.0)

  print *,'After assignment'
  print *,'logical   : ',a
  print *,'integer   : ',b
  print *,'real      : ',c
  print *,'character : ',d
  print *,'complex   : ',e

end program data_types
```

The code should return an output like this (an unassigned value can be different)

```
Before assignment
logical   :  F
integer   :            0
real      :    0.00000000
character :
complex   :  (  0.00000000    ,  0.00000000    )
After assignment
logical   :  F
integer   :            1
real      :    3.14000010
character : a
complex   :  (  1.00000000    ,  0.00000000    )
```

Binary operators

Fortran defines number of binary operators which may be classified based on their type to arythmetic, character, relation and logical operators. The form of the operators differs from other programming languages. Table bellow lists all Fortran's binary operators based on their type.

Operator type	Symbol	Description
Arythmetic	`**`	power
	`*`	multiplication
	`/`	division
	`+`	addition
	`-`	subtraction
Character	`//`	merging strings
Relations	`>` or `.gt.`	greather than
	`<` or `.lt.`	lower than
	`>=` or `.ge.`	greather or equal than
	`<=` or `.le.`	smaller or equal than
	`==` or `.eq.`	equal to
	`/=` or `.ne.`	non-equal to
Logical	`.not.`	negation
	`.and.`	conjunctions
	`.or.`	inclusive
	`.eqv.`	equivalence
	`.neqv.`	non-equivalence

Single, double precision

A given type of a variable in Fortran can have multiple precisions. For example a real type can be of single, double or higher precision. The same for integers, however they do not have a similar naming convention. In the context of integers people talk about their length and be 32-bit (4 byte), 64-bit (8 byte) long (or wide).

For example `real` numbers defined as

```
real :: a, b
```

will have a default precision given by a compiler. The precision can be selected manually with the `kind` keyword, e.g.

```
integer, parameter :: dp = selected_real_kind(15)
real(kind=dp) :: a, b

a = 1.0_dp
b = 0.0_dp
```

will give the variables `a` and `b` at least 15 significant digits. Most convenient way of using this is via defined precision module. Compilers will accept statements `real(kind=dp)` as well as `real(dp)`, for example

```
module real_precision

  implicit none

  integer, parameter :: sp = selected_real_kind(6)
  integer, parameter :: dp = selected_real_kind(15)

end module real_precision

program real_numbers

  use real_precision

  implicit none

  real(sp) :: a = 1.0_sp
  real(dp) :: b = 1.0_dp

  print *,a
  print *,b

end program real_numbers
```

will result in

```
   1.00000000
   1.0000000000000000
```

Function `select_real_kind(prec,[range])` takes also `range` parameter which is optional. Single, double and quadrupole precision may be defined as

```
integer, parameter :: sp = selected_real_kind(6, 37)
integer, parameter :: dp = selected_real_kind(15, 307)
integer, parameter :: qp = selected_real_kind(33, 4931)
```

and represent 32, 64 and 128-bit real numbers.

Fortran has build it functions `radix()`, `digits()` and `sizeof()` for quering the integer or real numbers.

I-N rule

Fortran 90 has several constructs which are left from Fortran 77 to ensure its backward compatibility. One of them is the I-N rule. Earlier versions of Fortran let the programmer skip definition of variables. By default Fortran will assign an integer type to all variables which first letter of the name is between `I` and `N`. All other variables will have real type. This implicit type assignement may be prevented by using `implicit none` statement. For example

```
program in_rule

  ! integer ( variable name starts with letter between I and N)
  i = 1
  print *,'i = ', i

  ! real (starting with all other letters)
  a = 3
  print *,'a = ', a

end program
```

giving

```
 i =           1
 a =    3.00000000
```

To prevent the compiler from assigning an implicit type to variables based on the first letter of their name one has to add line

```
implicit none
```

at the top each variables definition. The above program

```
program in_rule

  implicit none

  i = 1
  print *,'i = ', i

  a = 3
  print *,'a = ', a

end program
```

will not compile with error of `i` and `a` having not assigned type, for example gfortran (4.9) will report the following

```
  a = 3
   1
Error: Symbol 'a' at (1) has no IMPLICIT type
i-n-rule.f90:8.3:

  i = 1
   1
Error: Symbol 'i' at (1) has no IMPLICIT type
```

Adding

```
integer :: i
real :: a
```

under the `implicit none` statement gives the solution to that problem.

Many of the older Fortran code take advantage of that rule additionaly implicitly assigning all real numbers to double precision with the statement

```
implicit double precision(a-h,o-z)
```

with this line all variables starting with leters between a-h and o-z will be implicitly treated as double precision real numbers.

`iso_fortran_env` module

The `iso_fortran_env` module simplifies the precision definitions by providing three types `REAL32`, `REAL64` and `REAL128` as `kind` selectors. Definitions of single, double and quadruple precision may look like this

```
use, intrinsic :: iso_fortran_env

integer, parameter :: sp = REAL32
integer, parameter :: dp = REAL64
integer, parameter :: qp = REAL128
```

Basic type convertion

Basic data types can be converted internaly between one another using `int(x)`, `real(x)`, `dble(x)` and `cmplx(x,y)` functions. `int()` converts to an integer and `real(x)` and `dble(x)` to single and double precision real numbers respectively. Function `cmplx(x,y)` returns a complex number with real part `x` and imaginary part `y` respectively (second parameter is optional). Both components are also converted to real numbers. Summary in table below

Function	Final data type
`real(x)`	real (single precision) number
`dble(x)`	real (double precision) number
`int(x)`	integer
`cmplx(x,[y])`	complex number (`y` is optional)

The following code demonstrates the basic data convertion

```
program type_convertion

  implicit none

  real :: a, b
  integer :: e, f
  complex :: g, h

  a = 1.0
  b = 2.0

  e = 1
  f = 2

  print *, a, int(a)
  print *, b, dble(b)
  g = cmplx(a,b)
  print *, g

  print *, real(e)
  print *, dble(f)
  h = cmplx(e,f)
  print *, h


end program type_convertion
```

which produces

```
   1.00000000                1
   2.00000000       2.0000000000000000
 (  1.00000000    ,  2.00000000    )
   1.00000000
   2.0000000000000000
 (  1.00000000    ,  2.00000000    )
```