`if` conditional

Simple `if` conditional in Fortran resembles many programming languages and has a general structure

```
if (logical expression) then
    [ body 1 ]
else
    [ body 2 ]
end if
```

where `logical expression` is either a one utilizing of the logical binary operators `>`, `<`, etc., or check of a value (positive value is equivalent to `.true.` and negative to `.false.`). When the logical expression (or more than one connected with `.and.` or `.or.` operators takes value equal `.true.`) then program proceeds with `[body 1]`, otherwise with `[body 2]`. For example

```
program if_condition

  implicit none

  real :: val

  call random_number(val)

  if (val > 0.5 ) then
    print *, 'Value larger than 0.5 : ', val
  else
    print *, 'Value smaller than 0.5 : ', val
  end if

end program if_condition
```

The program above will use subroutine `random_number(x)` to generate a single random number from range (0,1) and then checks if the value is larger or smaller than 0.5. The output may look like this

```
Value larger than 0.5 :   0.997559547
```

If more complicated decision making tree is needed Fortran also supports `else if` branching of the basic `if` conditional

```
if (logical expression 1) then
    [ body ]
else if (logical expression 2) then
    [ body ]
else if (logical expression 3) then
    [ body ]
else if (logical expression 4) then
    [ body ]
else
    [ body ]
end if
```

Compatibility with earlier versions of Fortran has left a rather unusual `if` conditional which is based on the value of the variable used in the logical expression

```
 if (variable) LineX, LineY, LineZ
 ...
 LineX [ some code ]
 ...
 LineY [ some code ]
 ...
 LineZ [ some code ]
```

Above `LineZ`, `LineY` and `LineZ` are numbers that identify first line of the code block. The condition `if (variable) LineX, LineY, LineZ` checks the value of the `variable` and in case of a negative number proceeds to `LineX`. If the value is 0 then the code proceeds from `LineY` and otherwise (positive value) from `LineZ`. The code below will utilize that feature

```
program value_if_condition

  implicit none

  integer :: a, b, c

  a = -1
  b = 0
  c = 1

  call value_check(a)
  call value_check(b)
  call value_check(c)

end program value_if_condition

subroutine value_check(val)

  implicit none
  integer, intent(in) :: val

  if( val ) 10, 20, 30

10 print *, 'Value < 0'
   goto 100

20 print *, 'Value = 0'
   goto 100

30 print *, 'Value > 0'
   goto 100

100 print *, 'Done'
    continue

end subroutine value_check
```

which will output

```
Value < 0
Done
Value = 0
Done
Value > 0
Done
```

In the code above the subroutine uses the value based `if` conditional and has three target code blocks maked with line number `10`, `20` and `30`. Each of those blocks ends with statement `goto 100` which is a common ending place where program finaly exits the conditional execution. We printed a message in that place, however simple `continue` keyword is enough to proceed further.

`case` conditional

`case` conditional structure is designed to simplify complicated `if ... else if` decision trees that are based of checking value of a single variable, for example

```
if (var == val1) then
    [ block 1 ]
else if (var == val2 ) then
    [ block 2 ]
else if (var == val3 ) then
    [ block 3 ]
else
    [ block 4 ]
end if
```

can be replaced with `case` conditional

```
select case (var)
    case (val1)
        [ block 1 ]
    case (val2)
        [ block 2 ]
    case (val3)
        [ block 3 ]
    case default
        [ block 4 ]
end select
```

Specialized blocks are specified for values `val1`, `val2` and `val3`. The default case is given at the end and it applies to all other values of variable `var` than `val1`, `val2` or `val3`. Example program may look like this

```
program case_condition

  implicit none

  integer :: val

  val = 10

  select case (val)
    case (1)
      print *, 'One', val
    case (2)
      print *, 'Two', val
    case (3)
      print *, 'Three', val
    case default
      print *, 'Value different than 1,2,3 : ', val
  end select

end program case_condition
```

which will output

```
Value different than 1,2,3 :           10
```

Conditional labeling

Similar to loops programmer can assign a label to conditional statements to make it easy to edit complicated nested `if` conditionals

```
IfLabel: if (val > 0) then
    [ body ]
end if IfLabel
```

For example

```
program if_label

  implicit none

  integer :: val = 3.0

  IfLabel: if (val > 0.0 ) then
    print *, 'Value larger than 0.0'
  end if IfLabel

end program if_label
```

which gives

```
 Value larger than 0.0
```