The Fibonacci sequence

The Fibonacci sequence (which originates from a mediaeval model for rabbit breeding) is the sequence $\{x_n\}$ where

$$x_n = x_{n-1} + x_{n-2} \quad n \geq 3 \tag{1}$$

and $x_1 = x_2 = 1$. The number x_n represents the number of rabbits in the n-th month.
Moreover, the ratio x_n/x_{n-1} tends to the golden ratio

$$\phi = \frac{\sqrt{5}+1}{2} \tag{2}$$

Can we write a computer program generate the first $N = 50$ terms of the Fibonacci series and thence estimate the golden ratio?

Algorithm

Our algorithm (i.e. system of sequential steps) is as follows:

1. Set $x_1 = x_2 = 1$ (initial conditions)
2. Set $n = 3$ (initialise the iterator)
3. Calculate $x_n = x_{n-1} + x_{n-2}$
4. Print out x_n and x_n / x_{n-1}
5. Increment n
6. If $n < 50$ then go back to step 3

Steps 1-5 we know how to do already. But what about step 6?
What data type for n and the $\{x_n\}$? What variable names shall we use?

Program design

- write $\mathtt{x} \equiv x_n$, $\mathtt{y} \equiv x_{n-1}$, $\mathtt{z} \equiv x_{n-2}$
- variable `x`, `y`, `z`, `n` are all of type `INTEGER`
- then iteration step is `x = y + z`
- have to rename `z = y`, `y = x` before next iteration

Fibonacci program

```
PROGRAM fibprog1    ! program name
  IMPLICIT NONE     ! assume nothing about variable names
  INTEGER :: x, y, z ! declare some variables
  REAL :: phi_est   ! approximate phi
  INTEGER :: n = 3 ! declare the iterator
  z = 1             ! initial condition
  y = 1             ! initial condition
2 x = y + z         ! iteration step, preceded by label '2'
  phi_est = x / (1.0 * y) ! approximate phi
  PRINT*, x, phi_est ! print out values
  z = y             ! shift variables
  y = x             ! shift variables
  n = n + 1         ! increment the counter
  IF (n < 50) GO TO 2
END PROGRAM fibprog1
```

The `GO TO` statment

The syntax of the `GO TO` statement is

`GO TO` *label*

where *label* is a statement label elsewhere in the program.
A statement label must be on the same program line as an executable statement (e.g. an assignment or arithmetic operation) rather than an information statement (e.g. a type declaration).
It must be the first thing on the line.
The `GO TO` statement transfers program control to the line of code with the statement label. It leads to confusing code, so should be used sparingly

The `IF` statement

The syntax of the *IF* statement is

`IF` (*logical-expression*) *action-statement*

The statement *action-statement* is executed if *logical-expression* is true.

Note that *action-statement* must fit on the one line; for compound statements, we use the `IF` construct.

The `IF` construct

We use the `IF` construct when we want to handle a condition and then execute a block of code. The syntax is as follows:

IF (*logical-expression*) THEN

  block1

ELSE

  block2

END IF

- The code *block1* is executed if it logical-expression is true, otherwise *block2* is executed.
- The `ELSE` component is optional.

Examples of the IF construct

```
PROGRAM if_prog1            ! start of program
  IMPLICIT NONE             ! assume nothing about variable names
  INTEGER :: x, y           ! declare integers x and y
  PRINT*, 'Please enter two integers'                   ! print promp
  READ*, x                  ! read in x
  READ*, y                  ! read in y
  IF (x < y) THEN           ! do a logical test
    PRINT*, 'The first number is the smaller.' ! print somet
    x = y - x               ! do a bit of math
    PRINT*, 'The difference is ', x ! print some more
  ELSE                      ! if the condition is false, do the i
    PRINT*, 'The first number is NOT smaller.' ! print a dif
  END IF                    ! the end of the IF block
END PROGRAM if_prog1
```

Relational operators

The following relational operators are useful in the construction of logical expressions:

- == : equal to (contrast to assignment)
- /= : not equal to
- < : less than; <= : less than or equal to
- > : greater than; >= : greater than or equal to

Note that

- it is bad programming to test if two variables of type `REAL` are equal.
- these operators have old-fashioned text equivalents
  `.eq.` `.ne.` `.lt.` `.le.` `.gt.` `.ge.`

The (infinite) `DO` construct

We use the `DO` construct when we want to loop without using `GO TO`. The syntax is as follows:

```
DO
   code-block
END DO
```

- When the `END DO` statement is reached, control returns to the beginning of the loop.
- The loop is exited only if there exists an `EXIT` statement within *code-block*.
- The `EXIT` statement takes control of the program immediately after the next `END DO`.

The (definite) `DO` construct

```
INTEGER :: myvar, mymin, mymax
...
DO myvar = mymin, mymax
  code-block
END DO
```

This loop is equivalent to

```
myvar = mymin
IF (mymax >= mymin)
  DO
    code-block
    myvar = myvar + 1
    IF (myvar > mymax) EXIT
  END DO
END IF
```

The (definite) `DO` construct

- The definite `DO` loop is executed a maximum number of times.
- Note that *code-block* is still allowed to contain `EXIT` statements.
- Beware fiddling with the index variable within the loop.
- The beginning, end point, and step size in this sort of `DO` loop must all be integers.

Solving quadratic equations

We want to write a program so solve the quadratic equation

$$x^2 + bx + c = 0$$

This has solution

$$x = -\frac{b}{2} \pm \frac{\sqrt{d}}{2}$$

where the discriminant $d = b^2 - 4c$.
We note that if $d < 0$ then the equation does not have a real solution.

Solving quadratics: program

```
PROGRAM quad1
  IMPLICIT NONE
  REAL :: b, c, d, x1, x2
  PRINT*, 'We are solving the eqn x**2+b*x+c==0'
  PRINT*, 'Key in the parameters b and c'
  READ*, b
  READ*, c
  d = b * b - 4.0 * c
  IF (d < 0) THEN
    PRINT*, 'No real solutions exist.'
  ELSE
    x1 = -0.5 * (b + SQRT(d))
    x2 = -0.5 * (b - SQRT(d))
    PRINT*, 'The solutions are ', x1, ' and ', x2
  END IF
END PROGRAM quad1
```