Programming the computer

Programming the computer consists in producing a series of commands that the computer can understand in order to produce a desired action. Typical actions that electronic computers can perform are input and output of data, data processing, and control of interfaces or other devices.

At the most basic level, an electronic computer simply interprets voltage pulses as data or commands. From a mathematical point of view, the basic processing of information by the computer consists in the manipulation and interpretation of binary digits or *bits* (i.e., zeros and ones). At this level, thus, commands are represented by strings of zeros and ones. Communicating with the computer at its most basic level takes place through the use of *binary* or *machine language*. Trying to type in these binary commands for computer programming would be overwhelmingly slow and tedious. The use of human-like language significantly facilitates the programming of computers.

The next level of computer language is referred to as *assembly* or *assembler language*. It consists of simple commands like ADD, STORE, RECALL, etc., followed by memory addresses within the computer. Although an improvement over machine language, assembly or assembler languages are still quite primitive. High-level languages such as *FORTRAN 90*, *C++*, *C#*, *Java*, *Visual Basic 6.0*, etc., with their human-language-like syntaxes (typically, English) facilitate the programming of the computer for most human programmers.

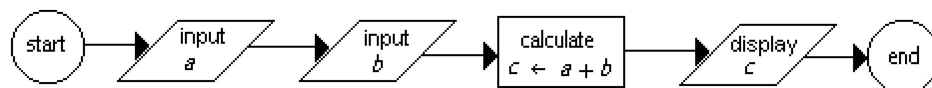
Each one of these high-level languages possesses its own syntaxes or language rules. Violation of the syntaxes of a specific language will result in failure of programming the computer. Programs in high-level languages are typically typed into the computer in the form of text files, and then run through a special program referred to as an *interpreter* or a *compiler*. The interpreter or compiler translates the programs into machine language, the basic language that all computers understand.

Tools for programming

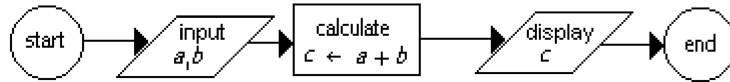
Typically, to write a program, the programmer starts by selecting or designing an *algorithm*, i.e., a plan for performing the action required from the computer. An algorithm can be simply a series of sequential steps that the computer must perform to produce a result. For example, if we intent to use the computer to add two numbers (a relatively simple operation, mind you), we can describe the corresponding algorithm in words as follows: *input the first number, input the second number, add the two numbers, store the resulting number into a memory location, show the number in the screen*.

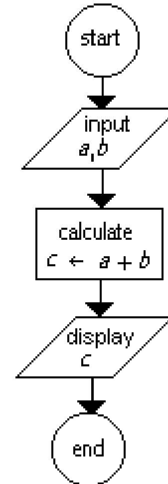
Flow charts

An algorithm can be presented or described using a flowchart. A flowchart is simply a collection of geometric figures connected by arrows that illustrate the flow of the algorithmic process. The geometric figures contain information in the form of commands or mathematical operations describing the algorithm and the arrows indicate the sequence of steps in the algorithm. The following flowchart describes the algorithm written above for the addition of two numbers.



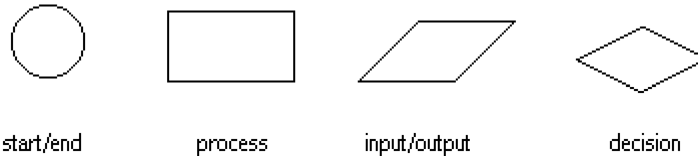
This flowchart is quite detailed for a simple operation. A more simplified version is shown below:





Many at times, a sequential flowchart will be presented with a vertical flow as illustrated in the figure to the right.

The basic components of a flowchart are the following geometric shapes, plus the arrows:



In the previous example we used all but the "decision" shape. We will show examples of decision algorithms later in the class.

Coding a program

The listing of a program written in a high-level language is also referred to as *code*. For example, the code corresponding to the flowchart shown earlier, when written in Visual Basic 6.0, would look like this:

```
Private Sub Add_Click()
         Dim a As Single
         Dim b As Single
         Dim c As Single
         a = Val(txtA.text)
         b = Val(txtB.text)
         c = a + b
         picOutput.Print "c =", c
End Sub
```

The line `Private Sub Add_Click()` can be thought of as representing the *start* circle in the flowchart, while the line `End Sub` can be thought as representing the *end* circle in the flowchart. The statements `a = Val(txtA.text)`and `b = Val(txtB.text)` represent the input boxes in the flowchart, while the statement `picOutput.Print "c =", c` represents the output box in the flowchart. The statement `c = a+b` represents the process box in the flowchart above.

The lines that start with `Dim` are specification statements that are used to identify the variables used in the code. For example, the statement `Dim a As Single`, for example, identifies variable *a* as a *single-precision* variable. (Single-precision variables are used to store real numbers -- i.e., numbers that may have a decimal part -- using one word [8 bytes] of memory. To carry more decimals, *double-precision* variables -- i.e., those incorporating two words of memory, or 16 byte -- can be used. Integer numbers can be stored in *integer* variables).

Specification statements need not be included in a flowchart. Furthermore, some high-level programming languages do not require the inclusion of specification statements. For example, the code for the previous flowchart can be written in SCILAB as follows:

```
Function add()
   a = input("Enter a:")
   b = input("Enter b:")
   c = a + b
   disp(c, "c=")
```

In SCILAB all variables are stored in double-precision format. Thus, as soon as the statement is executed, a double-precision memory location with the name *a* is created. Also, notice that this SCILAB function does not require an *end* statement.

Pseudo-code

While flowcharts are useful guides for describing an algorithm, producing a flowchart can become quite complicated, particularly if done by hand. Also, modification of a hand-made flowchart can be quite involved. Luckily, flowcharting software are now available that can simplify the process. If one doesn't want to get involved in producing a flowchart, one can use another technique known as *pseudo-code*.

Pseudo-code simply means writing the algorithm in brief English-like sentences that can be understood by any programmer. For example, a pseudo-code corresponding to the algorithm described in the flowchart shown earlier is presented next:

```
Start
  Input a, b
  a ← b + c
  Display c
End
```

Notice the use of the algorithmic sentence $a \leftarrow b + c$, in both the flowchart and the pseudo-code, to indicate that the addition of *a* and *b* is to be stored in variable *c*. This algorithmic sentence was translated in the Visual Basic 6.0 code as `c = a + b` because, in that high-level language, the equal sign represents an *assignment* operation (i.e., the value $a+b$ is assigned, or stored into, *c*). Since the equal sign (=) represents assignment in most high-level languages, it is possible to write the sentence `n = n + 1`. This sentence, rather than representing an algebraic equality that will result in the wrong result $0 = 1$, indicates that the value contained in variable *n* is to be incremented by *1*, and the resulting value is to be stored into *n*.

Basic programming structures

There are three basic programming structures: *sequence*, *decision*, and *loops*.

Sequential structure

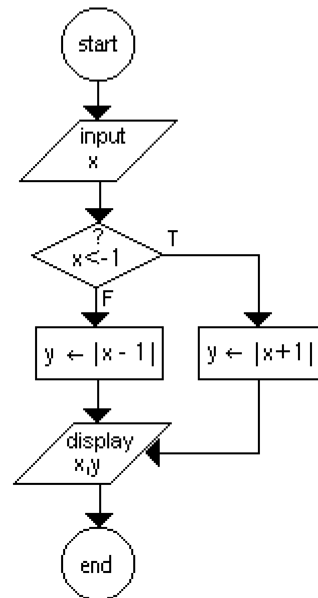
A *sequential structure* was used in the previous section to illustrate the use of flowcharts. Sequential structures have a single entry point and a single output point, and consist of a number of steps executed one after the other. Sequential structures can be useful in simple operations such as the addition of two numbers as illustrated earlier. The following pseudo-code illustrates a sequential structure consisting of entering a number and evaluating a function given by a single expression:

```
start
  request x
  calculate y = 3 sin(x)/(sin(x)+cos(x))
  display x, y
end
```

Decision structure

A *decision structure* provides for an alternative path to the program process flow based on whether a logical statement is true or false. For example, suppose that you want to evaluate the function

$$f(x) = \begin{cases} |x+1|, \text{if } x < -1 \\ |x-1|, \text{if } x \geq -1 \end{cases}.$$



The flowchart to the right indicates the process flow of a program that requests a value of *x* and evaluates *y = f(x)*.

The diamond contains the logical statement that needs to be checked to determine which path (T - true, or F - false) to follow. Regardless of which path is followed from the diamond, the control is returned to the *display* statement. Notice that the *input* statement and the decision statement form a sequence structure in this flowchart. As in this example, the three types of structures under consideration (sequence, decision, loop) do not appear alone, but two or three are commonly combined in many algorithms.

The algorithm illustrated above can be written in pseudo-code as follows:

```
start
  input x
  if x<-1 then
     y ← |x+1|
  else
     y ← |x-1|
  display x,y
end
```

The following function represents a possible translation of this pseudo-code into SCILAB code:

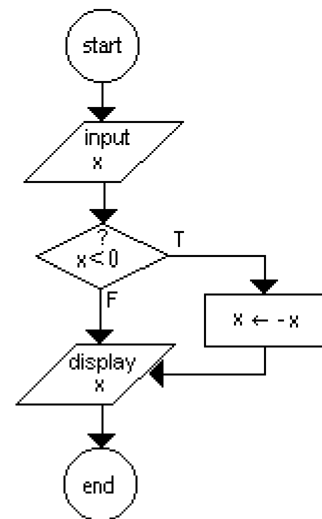
```
function f00()
   x = input("Enter x:")
   if x<-1 then
      y = abs(x+1)
   else
      y = abs(x-1)
   end
   disp(y, "y = ", x, "x = "
```

A possible translation into Visual Basic 6.0 is shown next:

```
Private Sub f00_click()
    Dim x As Double, y As Double
    x = Input("Enter x:")
    If x<-1 Then
        y = abs(x+1)
    Else
        y = abs(x-1)
    End If
    MsgBox("x = " & x & ", y = ", y)
End Sub
```

The decision structure shown above is such that an action is taken whether the condition tested is true or false. In some cases, if the condition tested is false, no action is taken. This is illustrated in the flowchart to the right that describes the definition of the function *absolute value*:

$$f(x) = \begin{cases} x, if \ x > 0 \\ -x, if \ x < 0 \end{cases}$$



Notice that the value of x is redefined as $-x$ if $x < 0$, but it does not change if $x>0$. Thus, action is only taken if the condition $x<0$ is true (T). In pseudo-code, this algorithm will be written as:

```
start
  input x
  if x<0 then
     x = -x
  display x
end
```

The translation into SCILAB, for example, is pretty straightforward:

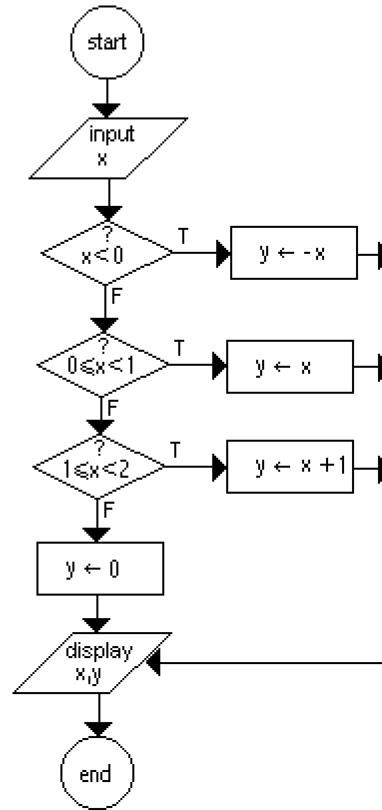
```
function fAbs()
   x = input("x = ")
   if x<0 then
      x = -x
   end
   disp(x, "|x| = ")
```

For Visual Basic 6.0, the code may look like this:

```
Private Sub fAbs_click()
    Dim x As Double
    x = Input("Enter x:")
    If x<0 Then x = -x
    MsgBox("x = " & x & ", y = ", y)
End Sub
```

A decision structure may include more than one or two possible paths. For example, if we define a function *f(x)* by

$$f(x) = \begin{cases} -x, \; if \;\; x < 0 \\ x, \; if \;\; 0 \le x < 1 \\ x+1, \; if \;\; 1 \le x < 2 \\ 0, \; elsewhere \end{cases},$$



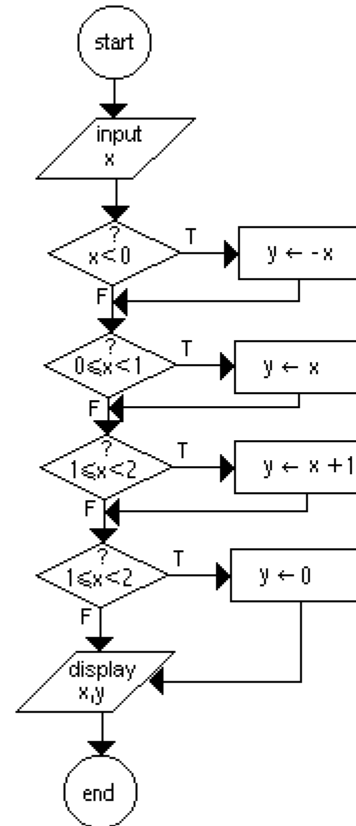
depending on the different conditions listed (e.g., *x<0, 0≤ x<1,* etc.), one of four actions will be taken. This decision structure is represented by the flowchart to the right.

Notice that only three conditions are shown in this flowchart, with a default assignment (y←0) that takes place if none of the three conditions tested is true. In pseudo-code, such a multiple-decision structure will be written as:

```
start
  input x
  if x<0 then
     y ← -x
  else if 0 x<1 thens
     y ← x
  else if 1 <x<2 then
     y ←x+1
  else
     y ← 0
  display x,y
end
```

Notice that the use of the particle *else if* implies that the condition following is one of many for a specific decision structure. If we re-write this pseudo-code as follows:

```
start
  input x
  if x<0 then
     y ← -x
  if 0 x<1 then
     y ← x
  if 1 <x<2 then
     y ←x+1
  if x ≥ 2 then
     y ← 0
  display x,y
end
```



The meaning of the decision structure changes, even though the result may not change. The corresponding flowchart is shown to the right.

From this flowchart, it is clear that the algorithm shown represents four simple decision structures, rather than a single decision structure with four possible outcomes as depicted earlier.

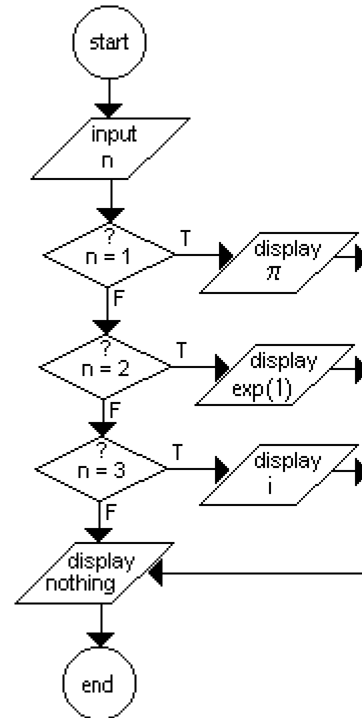
Visual Basic 6.0 codes for the two algorithms presented above are shown in the following table:

<table>
<tr><th>Single decision structure with multiple outcomes</th><th>Combination of multiple decision structures</th></tr>
<tr><td><pre>Private Sub f01A_click()
    Dim x As Double
    x = Input(“Enter x:”)
    If x<0 then
       y = -x
    ElseIf 0<=x<1 then
       y = x
    ElseIf 1<=x<2 then
       y = x+1
    Else
       y = 0
    End If
    MsgBox(“x= ”& x &“, y = ”,y)
End Sub</pre></td><td><pre>Private Sub f01B_click()
    Dim x As Double
    x = Input(“Enter x:”)
    If x<0 then
       y = -x
    End If
    If 0<=x<1 then
       y = x
    End If
    If 1<=x<2 then
       y = x+1
    End If
    If x>2 then
       y = 0
    End If
    MsgBox(“x= ”& x &“, y = ”,y)
End Sub</pre></td></tr>
</table>

Most high-level computer languages include a *case-select* structure to code a multiple-decision structure as described above. In a *case-select* structure, one out of many possible outcomes is selected depending on the value of a selector variable. An example of a *case-select* structure in SCILAB is presented next in the form of a SCILAB script (a file containing SCILAB commands loaded with the command *exec*):

```
mode(-1);//Suppresses command listing in the screen
printf(" \n")
printf("Select a case:\n")
printf("==============\n")
printf("   1 - pi\n")
printf("   2 - e \n")
printf("   3 - i \n")
printf("==============\n")
n = input("")
select n
case 1 then
         disp(%pi,"Pi = "),
case 2 then
         disp(exp(1),"e = "),
case 3 then
         disp(%i,"i = "),
else
         disp("Nothing to display")
end
mode(1);//Restores command listing in the screen
```

A flowchart for this *case-select* structure is shown to the right. Notice that this flow chart is not different from a multiple-decision structure. In pseudo-code, this flowchart could be translated as follows:

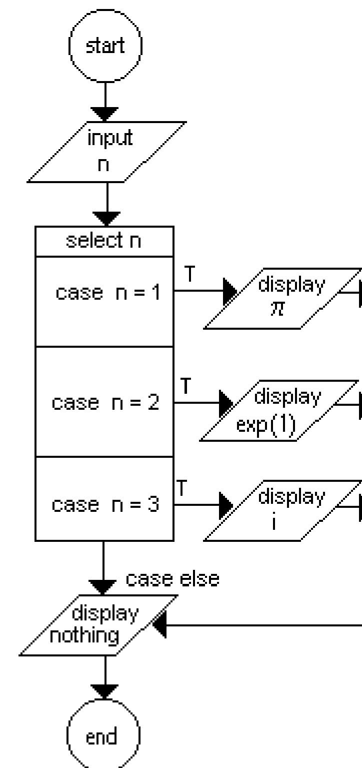



```
start
   input n
   if n = 1 then
      display p
   if n = 2 then
      display exp(1)
   if n = 3 then
      display i
   else
      display nothing
end
```

The *case-select* structure can be incorporated into the pseudo-code as shown next:

```
start
   select n
   case n = 1, display p
   case n = 2, display exp(1)
   case n = 3, display i
   else, display nothing
end
```

Thus, the *case-select* structure is a way to code a multiple-decision structure and not a different programming structure.



If the programmer decides to use the *case-select* structure from the beginning, it is possible to create a *case-select* flowchart symbol as illustrated in the flowchart to the right. Notice that this is not a standard flowchart symbol, but a made-up one to incorporate a *case-select* structure in the flowchart.

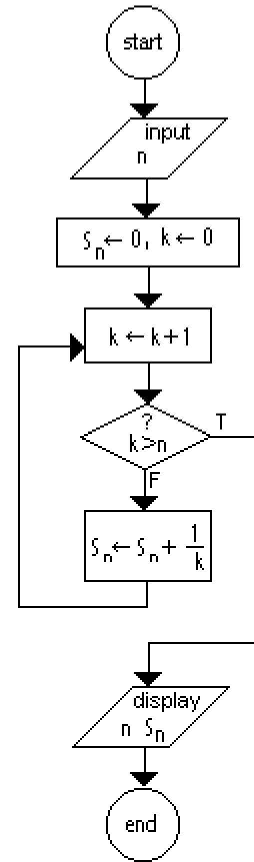
Loop structure

A *loop structure* represents a repetition of a statement or statements a finite number of times. For example, suppose that you want to calculate the following summation

$$S_n = \sum_{k=1}^{n} \frac{1}{n}.$$

The algorithm for the calculation is illustrated in the flowchart shown below. Notice that the loop structure is part of a sequence structure and that it contains a decision structure within, thus, re-emphasizing the fact that the three basic structures (sequence, decision and loops) commonly appear together in many algorithms. Notice also that the loop

structure requires an index variable, in this case *k*, to control when the process will leave the loop.



The summation S_n and the index variable *k* are both initialized to zero before the control is passed to the loop structure. The first action within the loop structure is to increment the index variable *k* by 1 ($k \leftarrow k + 1$). Next, we check if the value of *k* has not grown beyond that of *n* ($k > n$?). If *k* is still less than *n*, the control is passed to incrementing the summation ($S_n \leftarrow S_n + 1/k$), and back to the first step in the loop. The process is then repeated until the condition $k>n$ is satisfied. At this point, the control is passed on to reporting the results *n*, S_n.

The pseudo-code corresponding to the flowchart shown above is the following:

```
start
   input n
   Sn ← 0
   k ← 0
   do while ~(k>n)
       k ← k + 1
       Sn← Sn + 1/k
   end loop
   display n, Sn
end
```

Notice that instead of translating the loop structure in the flowchart with an *if* statement, we used the statement *do while*. (*Do-while* statements are commonly available in most high-level computer programming languages). Notice that a condition, namely *~(k>n)*, is attached to the *do while* statement in the pseudo-code. The statement is to be read as "*do (the statements in the loop) while k is not larger than n*". After the condition in the loop structure is no longer satisfied, i.e., when *k>n*, then the loop ends and the control is sent to the statement following the end of the loop.

[**Note:** The symbol ~ represents negation in mathematical logic notation, thus, if *p* stands for the statement *x>0*, ~p stands for ~(x>0) or x≤0].

A possible SCILAB coding of this process that uses SCILAB's *while* statement is shown next. The end of the loop is signaled by the particle *end*:

```
function myLoop1()
   n = input("Enter n:")
   Sn = 0, k = 0
   while k<= n
        k = k + 1
        Sn = Sn + 1/k
    end
    disp(Sn, "Sn = ")
```

To code this algorithm in Visual Basic 6.0 we can use the *Do While-Loop* structure as follows. In this case, the loop ends at the *Loop* statement:

```
Private Sub cmdLoop1_Click()
     Dim n As Integer, Sn as Single
     n = Val(txtN.text)
     Sn = 0 : k = 0
     Do While k<=n
        k  = k + 1
        Sn = Sn + 1/k
     Loop
     PicOutput.Print "Sn = ", Sn
End Sub
```

An alternative pseudo-code for the loop structure of the last flowchart is shown next:

```
start
   input n
   Sn ← 0
   k ← 0
   do
       k ← k + 1
       Sn ← Sn + 1/k
   loop while k<n
   display n, Sn
end
```

In this pseudo-code the test condition (`loop while k<n`) is located at the end of the loop statements rather than at the beginning as in the previous case. In Visual Basic 6.0, the coding of this pseudo-code is straightforward by using the *Do-Loop While* structure as follows:

```
Private Sub cmdLoop2_Click()
     Dim n As Integer, Sn as Single
     n = Val(txtN.text)
     Sn = 0 : k = 0
     Do
        k  = k + 1
        Sn = Sn + 1/k
     Loop While k<n
     PicOutput.Print "Sn = ", Sn
End Sub
```

A similar structure does not exist in SCILAB, thus, programming the latest pseudo-code in SCILAB will have to be done by using the *while* statement as shown earlier.

Most high-level computer programming languages provide a *for* loop structure by which an index variable is initialized, an increment to the index variable is provided, and a maximum value of the index variable is specified. For example, in SCILAB the *for* structure that we would use to calculate the summation S_n indicated above is shown next:

```
function myLoop2()
   n = input("Enter n")
   for k = 1:1:n
         Sn = Sn + 1/k
   end
   disp(Sn, "Sn = ")
```

The *for* statement in this case uses the so-called *colon operator* (:) to produce a range of values of k. Thus, the SCILAB statement *1:1:n* produces a range of values *1, 2, ..., n*. The general form of the range produced by the colon operator is:

$$k_0:\Delta k:k_f$$

where k_0 = initial value, Δk = increment, and k_f = upper limit. Assuming $\Delta k>0$ and $k_0 \le k_f$, the colon operator will produce the values $k_0, k_0+\Delta k, k_0+2\Delta k, \ldots, k_u$, where k_u is the such that $k_u<k_f$ and $k_u+\Delta k>k_f$. For example, the range *0.25:0.20:1* will produce the set of values *0.2, 0.45, 0.65, 0.85*. In this example, $k_u = 0.85$, which satisfies $k_u<1$ and $k_u+Dk = 1.05>1$. The number of elements in the rank generated by the statement $K_0:\Delta k:k_f$ is $[(k_f-k_0)/\Delta k]+1$, where the symbol $[x]$ represents the *floor* function, i.e., the integer value immediately below x. Thus, the number of elements in the range *0.25:0.20:1* is $[(1-0.25)/0.2] +1 = [0.75/0.2] + 1 = [3.75] + 1 = 3+1 = 4$.

When using the colon operator in SCILAB, it is possible to have a negative increment, i.e., $\Delta k<0$, in which case we must have $k_0 \ge k_f$. For example, the range *-0.25:-0.20:-1* includes the values -0.25, -0.45, -0.65, - 0.85. The number of elements in the range is still calculated by using $[(k_f-k_0)/\Delta k]+1$. For this case, the number of elements is $[(-1-(-0.25))/-0.2] + 1 = 4$.

Notes:

(1) If $\Delta k = 1$, the increment can be omitted. Thus, in function *myLoop2* shown above, we could replace the *for* statement *for k = 1:1:n* with *for k = 1:n*.

(2) In a range of the form $k_0:\Delta k:k_f$ when $\Delta k>0$, and $k_0 \ge k_f$, or when $\Delta k<0$ and $k_0 \le k_f$, the resulting range is empty. For example, try 5:1:1 or 1:-1:5.

(3) In a range of the form $k_0:\Delta k:k_f$ when $\Delta k<0$, and $k_0 \ge k_f$, or when $\Delta k>0$ and $k_0 \le k_f$, the resulting range is infinitely large. Thus, if used to control a loop structure, the program will enter an *infinite loop* (i.e., a loop without ending). You may accidentally create an infinite loop in a program, in which case, you will have to break out of the loop manually.

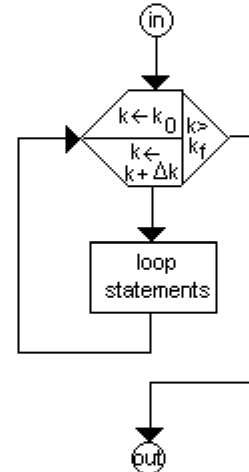
In Visual Basic 6.0, the *for* loop structure is referred to as a *For-Next* loop. A Visual Basic 6.0 coding for the program for the summation presented above is shown next:

```
Private Sub cmdLoop3_Click()
     Dim n As Integer, Sn as Single
     n = Val(txtN.text)
     Sn = 0
     For k = 1 To n Step 1
        Sn = Sn + 1/k
     Next k
     PicOutput.Print "Sn = ", Sn
End Sub
```

Notice that the loop structure starts with the statement *For k = 1 To n Step 1*, and ends with the statement *Next k*. In general, the *For* statement can be written as

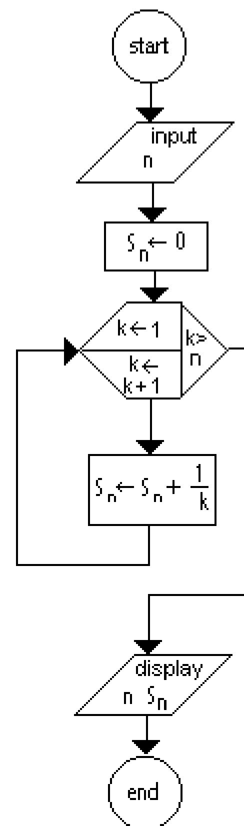
$$For\ k = k_0\ To\ k_f\ Step\ \Delta k$$

As with the colon operator in SCILAB, the index variable k takes the values k_0, $k_0+\Delta k$, $k_0+2\Delta k$, …, k_u, where k_u is the such that $k_u<k_f$ and $k_u+\Delta k>k_f$ if $\Delta k>0$ and $k_0 \leq k_f$. Regardless of whether $\Delta k>0$ or $\Delta k<0$, the number of values that k takes is also calculated as $[(k_f-k_0)/\Delta k]+1$, where the symbol $[x]$ represents the integer *floor* function. The notes given above for SCILAB's colon operator, apply also to Visual Basic 6.0's *For* statement.



As we did with the *case-select* structure in flowcharts, we can create our own flowchart symbol for a *for* loop. One possibility is the hexagonal flowchart symbol shown to the right. Right below the *in* symbol the hexagonal symbol shows the initialization of the index variable ($k \leftarrow k_0$). Right below this initialization, within the hexagonal symbol, the increment of the index variable is presented ($k \leftarrow k+\Delta k$). The increment step connects to the loop statements, indicating that these statements are repeated as long as the condition $k>k_f$ is not reached. This condition is shown in the right section of the hexagonal symbol indicating that, when it is satisfied, the control is sent *out* of the loop.

Thus, for the case of the summation program presented earlier, we can use the flowchart shown below to the right.



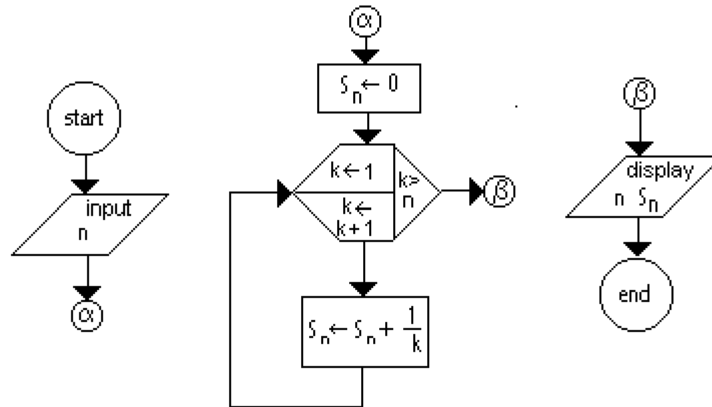
The corresponding pseudo-code will look like this:

```
start
  input n
  Sn ← 0
  for k = 1, Dk = 1, k>n
      Sn ← Sn + 1/k
  end For loop
  display n, Sn
end
```

Recall that the hexagonal *for* loop symbol is not a standard flowchart symbol, but one made up to represent the *for* programming structures that are available in most high-level computer programming languages.

Splitting the flowchart

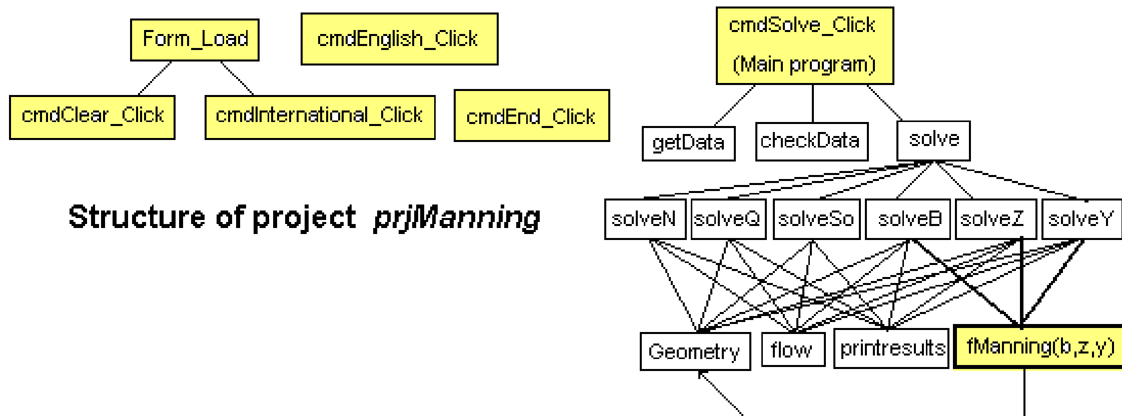
Sometimes, a complete flowchart will not fit in the paper. In such cases, you can split the flowchart by creating sub-flowcharts linked by circles identified with numbers or letters. The following example shows the same flowchart to the right, but split into three parts. The links are identified with Greek letters.



Hierarchy charts

Hierarchy charts are charts that show the different components of a program identified by tasks. A hierarchy chart is not as detailed as a flowchart or pseudo-code, instead, it shows the relationship between different components of a program in a concise manner. The hierarchy chart is used to strategize the coding of a program by splitting the program into self-contained, but interrelated, components. We will learn that such components can be coded as *functions* in SCILAB or as *Sub* procedures or *Function* procedures in Visual Basic 6.0. Once a program is divided into its components, flowcharts or pseudo-code can be developed for the individual components before coding the program.

The following is an example of a hierarchy chart developed to solve the problem of uniform flow in a trapezoidal open channel (a common problem in civil engineering hydraulics). The solution was produced in Visual Basic 6.0. The chart shows all the Sub procedures and Function procedures (one only, *fManning*) included in a Visual Basic 6.0 project called *prjManning*.



Structure of project *prjManning*

This example is presented here just to illustrate the concept of hierarchy charts (also known as *top-down charts*). Details of the program developed will be presented elsewhere.

This document was prepared by Gilberto E. Urroz on June 1-21, 2002.