Module5: Control Structures

§ 5.1 Logical Decisions and Jumps

In APL 1 and APL 2, program flow was controlled by branch (→) and also, sometimes, by execute (⍎) and ⎕SIGNAL plus ⎕TRAP. In APL 3 program flow may be controlled by more readable control structures such as :If. There are 8 different types of control structures in Dyalog APL. They are defined by the control words :If, :While, :Repeat, :For and :Select, :With, :Trap, :Hold. Like line labels, these structures are only usable inside programs, not in immediate execution mode.

§§ 5.1.1 The :If Statement

:If is a simplified version of branch (→).

```
      :If Prop⋄ ... ⋄:End                ⍝ If Prop is true execute ... code
```

If ⊢Prop , *ie* Prop=1↳1, where Prop is a logical (Boolean) proposition and therefore either true(1) or false(0), then execute the code indicated by ..., otherwise end the :If statement.

```
      :If Prop⋄ ... ⋄:Else⋄ ... ⋄:End    ⍝ If Prop is false execute second ...
```

If ⊢Prop, then execute the code indicated by the first ... expression, otherwise ⊢~Prop and the second ... expression is executed before ending the :If statement.

```
      :If Prop1⋄ ... ⋄:ElseIf Prop2⋄ ... ⋄:End
```

A :If statement may be embedded in another by means of the :ElseIf conditional.

5.1.1.1 Rewrite

```
      ⍎(0=⎕NC'Forename')/'Forename←'''''
```

in a :If statement.

§§ 5.1.2 Further truth Conditionals

```
      :If Prop1⋄:AndIf Prop2⋄...⋄:AndIf PropN⋄...⋄:End
```

Any number of :AndIf conditionals may be included after :If or :Else or :ElseIf conditionals.

```
      :If Prop1⋄:OrIf Prop2⋄...⋄:OrIf PropN⋄...⋄:End
```

Any number of :OrIf conditionals may be included after :If or :Else or :ElseIf conditional segments. :AndIfs and :OrIfs may not be mixed within individual segments of code.

5.1.2.1 Rewrite the function below in APL 1 (1st generation) language.

```
     ∇ pass(A B C D E)
[1]    :If A
[2]    :AndIf B
[3]        'A and B'
[4]    :ElseIf C
[5]    :OrIf D
[6]    :OrIf E
[7]        'C or D or E'
[8]    :Else
[9]        'not A and not C'
```

```
[10]    :End
      ∇
```

Which notation is more legible? What other pros- and cons- can you think of?

5.1.2.2 Rewrite the expression

```
      ⍎(P∧Q)/'''P and Q'''
```

where `P` and `Q` are propositions, without using any of the symbols `⍎→∧`. Replace `(P∧Q)` with an arbitrary logical expression involving logical connectives *and* (∧), *or* (∨) and *not* (~), *eg* `((P∨Q∨R)∧~P∧Q)`, and rewrite it in a `:If` statement control structure?

§§ 5.1.3 The `:Select` Statement

`:Select` is a simplified version of branch (→).

```
      :Select Arr◊:Case Arr1◊...◊:Case Arr2◊...◊...◊:End   ⍝ Execute ... case
```

Execute the code in the `:Case` segment expression which satisfies `⊢Arr≡ArrN` . The last ... in this structure implies the possibility of more `:Case ArrX◊...` code snippets.

```
      :Select Arr◊:Case Arr1◊...◊:Case Arr2◊...◊...◊:Else◊...◊:End
```

5.1.3.1 Run the function below with various sorts of arguments, such as

```
      WhatIs ⎕NULL

      ∇ WhatIs I
[1]     :Select I
[2]     :Case 1 ◊ 'I=1'
[3]     :Case 2 ◊ 'I=2'
[4]     :Else ◊ '((I≠1)∧(I≠2))∨(~I=1)∧~I=2'
[5]     :EndSelect
      ∇
```

and then replace `:Else` with a suitable `:CaseList` conditional qualifier, the definition of which is to be found in [Help][Language Help] or the invaluable Dyalog APL *Language Reference*.

§ 5.2 Looping Constructs

§§ 5.2.1 The `:For` Statement

In many computer languages, a For statement provides a compact way to iterate over a range of values. `:For` is a specific application of branch (→).

```
      :For Var :In Vec◊..Var...◊:End   ⍝ Execute ..Var... for each Sc in ,Vec
```

In each iteration, `Var` takes the value of the next element of vector `,Vec`.

5.2.1.1 Rewrite the expression `+/NumVec` in a `:For` statement, using `⎕MONITOR` to compare efficiencies.

5.2.1.2 When might this looping mechanism sometimes be preferable to using operators such as each (¨)?

Hint: try tracing examples of both options.

5.2.1.3 Convert a looping statement such as

```
      Loop:  ◊...◊  →Loop××⍳Bool
```

into a `:For` statement.

§§ 5.2.2 Generalised :For Statements

The Var entry may be replaced by multiple variable names. In this case Vec is expected to be a vector of vectors and the Nth variable in the list of names is assigned at each iteration to the Nth element in the disclosed next element of the control vector.

```
:For Var1 Var2 .. :In VecNVec⋄..Var1..Var2...⋄:End          ⍝ Strand
```

In each iteration, VarN takes the value of the Nth element of iteration subvector of the control vector. An example of a valid line in this case might be

```
:For V₁ V₂ V₃ :In (1 2 3)(4 5 6)(..)...
```

An alternative definition is used if :In is replaced with :InEach. Again Vec is expected to be a vector of vectors but in this case the Nth variable in the list of names is assigned, at each iteration, to the next element in the Nth element of the control vector.

```
:For Var1 Var2 .. :InEach NVecVec ⋄..Var1..Var2...⋄:End    ⍝ Distribute
```

In each iteration, VarN takes the value of the next element of vector N⊃NVecVec. An example of a valid line in this case might be

```
:For V₁ V₂ V₃ :InEach (1 4 ..)(2 5 ..)(3 6 ..)
```

In Module11 we shall see how a *collection object* may be treated as a Vec in a :For statement.

§§ 5.2.3 :Repeat and :While Loops

```
:Repeat⋄...⋄:Until Prop          ⍝ Repeat execution of ... until ⊢Prop
```

This is an infinite loop unless proposition Prop can change from false to true in the process.

5.2.3.1 Write a 2-line function with the infinite loop

```
[1] :Repeat ⍞⍞ [2] :Until 0 ⍞⍞
```

Run the function and break the execution in a number of different ways. Now convert to a 1-line function

```
[1] :Repeat ⋄ :Until 0
```

Try to break this loop. Be prepared to close APL. Repeat the experiment simply with `[1] →1`

```
:While Prop⋄...⋄:End             ⍝ Execute ... while ⊢Prop
```

5.2.3.2 Loop round executing some code while proposition Prop is true (1), or :Until Prop2 is true. Note :AndIf or :OrIf may be included in the structure logic of :While and at the end of :Repeat.

§ 5.3 Digging

§§ 5.3.1 The :With Statement

:With is an alternative form to ⎕CS.

```
:With Obj⋄...⋄:End               ⍝ Execute ... within object Obj
```

Obj may be the name of an object or an object reference (value). The lines of code in ... are executed inside the space of Obj. The effect of :With is similar to that of ⎕CS. Local variables in the outer space continue to be visible.

5.3.1.1 Write a function such as

```
      ∇ drill
[1]     :With ⎕SE
[2]         :With cbbot
[3]             :With bandsb1
[4]                 Dockable
[5]             :End
[6]         :End
[7]     :End
      ∇
```

to drill into `⎕SE.cbtop.ilh.bm` and display the `Type` property at each level.

Within the Dyalog function editor, [Edit][Reformat] indents control structures and substructures according to the settings in [Options][Configure][Trace/Edit]. As in the case of the `:For` statement, `:With` extends to Collections, as described in Module11. `:With` also extends to unnamed namespaces, as described in Module11.

§§ 5.3.2 Digging into SubSpaces

Within an APL program one is usually working with local variables and functions all in the same space. It would therefore be tedious to prefix all names with the space-qualified name, especially for deeply nested spaces. As we shall see when looking at `OLEClient` objects in Module7, the `:With` control structure plays an important role in identifying the current space in a program.

§§ 5.3.3 `:Trap` versus `⎕TRAP`

`:Trap` is a simplified version of `⎕TRAP`.

```
      :Trap ENum⋄...⋄:End              ⍝ Trap error ENum and execute ...
```

When running code ... , in the event of an error having an error number which is in the list `ENum`, no default error action is taken and execution is passed to code after the end of the `:Trap` control structure, if there is any. This is similar to the action of something like `⎕TRAP←ENum 'C' '→1+⎕LC'`.

```
      :Trap ENum⋄...⋄:Else⋄...⋄:End    ⍝ On error in first ..., do second ...
```

If an error of type `ENum` occurs in the first ... segment, then pass execution to the second segment and do not report the error. Further, disable the error trapping in the processing of the second segment.

```
      :Trap ENum⋄...⋄:Case ENum1⋄...⋄:Case ENum2⋄...⋄...⋄:End ⍝ Split
```

The last ... in this structure implies the possibility of more `:Case ENumX⋄...` code snippets. As in `:Select`, `:CaseList` and `:Else` segments may be used here too.

For a summary of the `:Hold` statement, see multi-threading in Module13.

5.3.3.1 Ask for the next module on **OLE Servers** ☺.