Module3: Dot Syntax

§ 3.1 Object References

§§ 3.1.1 Making References with ⍎ and ←

The APL primitive function *execute* (⍎) has been generalised to take the name of an object as its argument and return a *reference* to that object (of notional APL dataType RefSc).

```
RefSc←⍎CharVec                    ⍝ Returns scalar reference to object, name CharVec
```

Extended execute enables one to assign arbitrary names to a single GUI object or namespace.

```
F←⍎'F'
```

This has no noticeable effect as `F` already refers to the `Form`.

```
G←⍎'F'
```

This creates another 'ref' to the `Form`, previously identified by `F` but now, more or less equally, by `G`.

```
Arr←RefSc⍎CharVec                 ⍝ Returns result of executing CharVec in RefSc
```

The dyadic definition of *execute* with a space to its left is 'execute Rarg in space Larg'.

Changes to the object of one reference will make changes to the object of all equivalent references as there is in (virtual) reality only one underlying object. Thus for example,

```
F⍎'A←1'
⎕←G⍎'A'
1
```

```
VAR←RefSc                         ⍝ Creates new name, VAR, for the object referred to
```

Assignment has been generalised to take a ref on the right and assign that ref to the name on the left. So refs can be assigned to names in the same way as variables, and their values can be accessed just like (shared) variables. Thus objects are like 'deep variables' with 'shallow references'.

```
J←I←H←G
⎕←J⍎'A'
1
H⍎'A←5'
⎕←J⍎'A'
5
```

Notice that the `Form` itself will not disappear on `⎕EX'F'` - not until the last ref to it has been erased.

In version 9.0 the system functions `⎕CS`, `⎕NQ` and `⎕DQ` accept namespace refs as arguments as well as quoted names (character vectors).

Objects are essentially like variables and therefore, quite naturally, user defined functions may take refs as arguments and return refs as results.

A number of other primitive APL functions have been extended to accept arguments that are references to objects, or to return references to objects as their results.

```
BoolSc←RefSc1=RefSc2              ⍝ Determines the absolute equality of refs
```

If `⎕←RefSc1=RefSc2 ↳1` then `RefSc1` and `RefSc2` are two references to the same space. *Not equal* (≠) returns the opposite. *Match* (≡) and *not match* (≢) have been similarly extended. (Note that `⎕←0≡G=0`.)

```
BSc←RSc1≡RSc2                     ⍝ Determines the absolute identity of refs
```

3.1.1.1 Create two refs to the same object. Try varying some more or less subtle aspects of each ref (*eg* the `⎕AT` of some internal function) to see if their equality and identity can be made to diverge.

In APL 1, arrays may be indexed or otherwise manipulated without giving them a name. From what we have seen so far it would appear that objects have to be given a name, but this is not the case.

`RefSc←⎕NS⍬`	⍝ Return a ref, `RefSc`, to an **unnamed namespace**

Note ⊨ `⍬≡⍬`

A (pseudo-niladic) call to `⎕NS` returns a reference to an 'unnamed' namespace that can be manipulated just like a 'named' object. This is such a natural object that John Scholes has suggested that it be given a special symbol, `@`, which one might call *anon*.

`BSc←ñ1≅ñ2`	Whether namespaces are isomorphic

Isomorphism of spaces (≅) implies 'topological' or operational similarity but not absolute identity.

`@`	New anonymous namespace such that ⊨ `@≅⎕NS''`

In Dyalog version 11, `@` in fact has become the new executable niladic system functions, `⎕THIS`.

`BSc1 ⇒ BSc2`	Material implication of `BSc2` from truth value of `BSc1`

`(⊨ñ1≡ñ2) ⇒ ⊨ñ1≅ñ2`

The truth table of *implies* (⇒) may be defined as

`BSc1 ⇒ BSc2 ↦ ~BSc1∧~BSc2`

The most important inference is that if the *truth-value* of `BSc1` is true then `BSc2` is also true.

Logical Aside:

P ⇒ Q , P	∴ Q	is a *valid argument* (Modus Ponens)
P ⇒ Q , ~Q	∴ ~P	is a *valid argument* (Modus Tollens)
P ⇒ Q , Q ⇒ R	∴ P ⇒ R	is a *valid argument* (Hypothetical Syllogism)

3.1.1.2 Compare the deep similarity (isomorphism) of spaces `A` and `B` with the deeper identity of spaces `C` and `D`.

```
A←⎕NS''
B←⎕NS''
C←D←⎕NS''
```

Although these objects have much in common with ordinary (APL 1 & 2) variables, the system function `)VARS` does not report the names of global objects, and the name class of (scalar) objects is not 2, but 9. Instead, `)OBS` reports the names of global objects and `⎕NL 9` returns a matrix of all 'visible' global and local objects.

§§ 3.1.2 Parent.Child Hierarchy

`#`	⍝ Returns a ref to the Root space

The display name of the Root space is the one element vector `(,'#')`, thus ⊨ `⍕#↦,'#'`. Similarly, ⊨ `⍕⎕se↦'⎕SE'` and furthermore ⊨ `#≡⍎'#'` and ⊨ `⎕SE≡⍎'⎕se'`

`)NS`	Displays the name of the current namespace

If you change space to the Root space and hit `)NS` then you will be told that you are in the Root space `#`. `#` is a direct object reference to the Root space, and `⎕SE` is a direct object reference to the Session space. Every Dyalog primitive GUI object that you can create can trace its roots to `#` (or `⎕SE`).

If we create a `Form` object in `#` called `FRM` then this object can also be referred to as `#.FRM`. This is the beginning of *Dot Syntax* in Dyalog APL. Objects that are children of the Root can have `#.` prepended to their name without repercussions. Objects can be referenced hierarchically.

`CVec←⍕RefSc`	⍝ Returns the display form of `RefSc`

In the above example we have `⊢⍎FRM↳'#.FRM'`. In version 11, `⎕DF` can be used to modify the display.

The `Parent.Child` relationship is valid at all levels. If our `Form` had a child `Button` called `BTN` then this `Button` may be identified while in `#` by the syntax `FRM.BTN`, relating parent and child, surname first.

$\tilde{n}_3 \leftarrow \tilde{n}_1.\tilde{n}_2$	⍝ Returns a direct reference to subspace $\tilde{n}_2$

By means of dot syntax, objects can be referred to in a hierarchical fashion. Dot syntax describes object ancestry. If $\tilde{n}_2$ is a direct descendent (child) of $\tilde{n}_1$ then $\tilde{n}_1.\tilde{n}_2$ returns a reference to $\tilde{n}_2$ from a space containing $\tilde{n}_1$. The notation $\tilde{n}_1$ is used to represent the name of an argument of dataType `RefSc`.

3.1.2.1 Create a ref to an unnamed child of an unnamed namespace.

`VecCVec←⎕WN CVec`	⍝ Returns the name of each child object of `CVec`

This system function returns the names of all objects whose parent's name is given in `CVec`.
In a clear workspace, given

```
'FRM'⎕WC'Form'
'FRM.BTN'⎕WC'Button'
```

then

`⊢(⎕WN'#')≡,⊂'FRM'↳1` and `⊢(⎕WN'FRM')≡,⊂'FRM.BTN'↳1`

Also

```
⎕wn'⎕se'
⎕SE.cbtop  ⎕SE.cbbot  ⎕SE.mb  ⎕SE.popup  ⎕SE.tip
```

Consider John, also known as John Scholes, whose name is now to be written as Scholes.John to avoid any confusion with Daintree.John. In other words, prepending (rather than appending) the ancestral name identifies more specifically the John in question.

§§ 3.1.3 Object.Object. .. Object.Object Rationale

Dot syntax can be used repeatedly to reference objects deep inside an object hierarchy. For example, if a `Form F` has a child `Group G` which itself has a child `Edit E` and if `F.G.E` is called while execution is inside the parent of `F` then the result will be a direct relative reference to the `Edit` object. If `F` is a child of `#` then `#.F.G.E` will return an absolute reference to `E` when called from any space.

An address label written as *Country.City.Area.Road.Number.Surname.Forename* might serve as a useful model of a dotted hierarchy. A more precise analogy might be DOS (or UNIX) directories wherein C: is like the Root `#` and symbol \ (or /) is analogous to a dot.

Unfortunately the *dot* in dot syntax formally does not play the role of any regular APL syntactic element. In the current context, where dot has a namespace on either side and returns a namespace, the *dot* looks like a function. But parsing function expressions from right to left implies that `#.F.G.E` is equivalent to `#.F.(G.E)` and `G` is not necessarily visible from the current space and may give a `VALUE ERROR`.

However, interpreting *dot* as an operator with namespaces for both operands and derived result is more consistent with APL use of dot and with the required order of execution. Parsing operator expressions proceeds from left to right implying that `#.F.G.E` is equivalent to `((#.F).G).E` as required.

3.1.3.1 Examine the display forms of the derived functions

```
+∘×∘+∘×∘÷ ↳     +∘×  ∘+  ∘×  ∘÷
```

```
      -.+.×.÷ ↳   -.+ .× .÷
```

Experiment with the effect of parentheses in these expressions and others in order to exhibit various alternative roles of the operators.

3.1.3.2 Write an operator such as

```
      ∇ r←a(f Ô g)b
[1]     r←a f g b  ∇
```

and a set of functions such as

```
      ∇ r←{a}f1 b
[1]     :If 0=⎕NC'a'
[2]         r←÷b
[3]     :Else
[4]         r←a÷b
[5]     :End        ∇
```

then trace the order of execution of various expressions such as

```
     3  (f1 Ô f2) Ô f3 Ô f4 Ô  f5 4 ↳0.75
```

Try to force syntax errors. Note any interesting conclusions.

§ 3.2 Direct Property Access

§§ 3.2.1 Object.Variable Syntax

The value of a variable may be accessed or assigned by name from outside a namespace.

```
      Arr←ñ.VAR                    ⍝ Read variable named VAR inside visible space ñ
```

```
      ñ.VAR←Arr                    ⍝ Write variable named VAR inside visible space ñ
```

Namespaces can contain variables. Dot syntax extends to variable names on the right of a dot. This facilitates direct access to variables in other spaces.

GUI objects are essentially namespaces containing predefined properties etc .. , and properties are essentially variables. Therefore the above syntax should and does apply to objects and their properties.

3.2.1.1 Access the *Caption* of the *Form F* directly from the Root, where

```
      '#.F'⎕WC'Form' 'This is It'
```

Note that ⎕*WX* must be set to 1 in space *F*. The default value of ⎕*WX* in a clear WS is determined by the value of the registry parameter **default_WX**. This can be changed in the registry using **REGEDIT.EXE** at location **HKEY_CURRENT_USER\Software\Dyadic\Dyalog APL/W 9.0** or directly through the APL Session in [Options][Object Syntax][Expose GUI Properties].

3.2.1.2 Create a *Calendar CAL* on a *Form F* and experiment with properties such as

```
      F.CAL.CircleToday←0
      F.CAL.CalendarCols←?6⍴⊂3⍴255
      F.C.MinDate←38717          ⍝ 2006 1 1 6
```

3.2.1.3 Create a *RichEdit RE* on a *Form F* and experiment with properties such as

```
      F.RE.SelText←(1 1)(1 20)
      F.RE.CharFormat[1]←⊂'Italic'
      F.RE.CharFormat[5]←50      ⍝ Superscript
      F.RE.PageWidth←5×1440
```

```
F.RE.ParaFormat[1]←⊂'Centre'
F.RE.ParaFormat[3 5]←288 1
```

§§ 3.2.2 Object.Object. .. Object.Property Rationale

3.2.2.1 Create a namespace `#.A.B.C.D` containing a variable `V` with the value `1`. Access this variable from each of the spaces `#`, `A`, `B`, `C` and `D`.

The analogy with DOS directories can be extended to files in directories. **File** names the end of a directory string are like **variable** names at the end of a namespace string.

3.2.2.2 Create a `Menu` on a `MenuBar` on a `Form` and set the `Caption` and `Active` properties of the `Menu` directly from the `Form` space.

Since objects are essentially APL variables, the rationale behind multiply dotted expressions ending with a variable name is exactly the same as that for a similar expression ending with a namespace.

§§ 3.2.3 Using Object.Object. .. Object.Property Constructions

Object construction, as used in Visual Basic, can now be adopted, almost intact, into APL. This makes translation from VB often very straightforward. In particular, macros recorded in Microsoft Office products can be viewed in VBA (via **Alt+F11**) and translated easily into APL. This, as we shall see later, is a powerful way to transcribe Office OLE programs into APL.

3.2.3.1 Rewrite the function `∇MAKE_Form∇` in §§ 2.3.3 setting as many properties as possible using direct assignment from the Root (or by any preferred compromise with or without `⎕W`...).

§ 3.3 Direct Method Invocation

Like much of the Dyalog APL implementation of GUI concepts (such as the use of name-value pairs), *Dot Syntax* is imported from mainstream GUI-oriented computer languages such as Visual Basic and JavaScript. Dot Syntax is a shorthand notation that can be used to specify the properties of any object or to call any method on any object without explicitly having to be in their objects space. One difference worth noting, that we emphasise later, is that the object hierarchy in APL is more literal than that found in VB, which is more illusory.

§§ 3.3.1 Object.Function Syntax

`f←ñ.g`	⍝ `f` refers to function `g` in space `ñ`

Namespaces can contain functions. Dot syntax extends to function names on the right of the dot, allowing immediate access to functions in different spaces. The functions may be niladic or ambivalent and their arguments are found in the correct places for *dot* to be interpreted (informally) as an operator.

In this case the dot has a niladic, monadic or dyadic function `g` on its right and a space `ñ` on its left and returns a function `f` – essentially a call to function `g` from the current space from which `ñ` must be visible. Here the dot looks more like the familiar primitive *inner product* dot operator that takes a dyadic function to left and right and returns a derived dyadic function ($f_2 \leftarrow h_2 . g_2$).
Again the space on the left `ñ` can be replaced with a dotted string referring to any arbitrary subspace. Some justification for this extension has been given above.

3.3.1.1 Run a function defined in one namespace from another namespace, paying particular attention to the values of local and global variables and to the spaces in which sub-functions are actually executed.

§§ 3.3.2 Object.Object. .. Object.Function Rationale

Primitive variables and functions as well as user-defined ones succumb to dot syntax too.

```
F.⎕IO←0
F.⍳ 9↳0 1 2 3 4 5 6 7 8
```

In keeping with a choice of algorithms that APL frequently affords, this introduces more choice. *eg*

```
F.⍎'Caption'↳F.Caption
'F'⍎'Caption'↳F.Caption
F⍎'Caption'↳F.Caption
```

All these expressions give same result, with natural extension of dot syntax to primitive functions ⍳ and ⍎, and a natural generalisation of *dyadic execute* already alluded to. In the case of a scalar ref, *dyadic execute* gives the same results as `Arr←RSc.⍎CVec`.

The following three statements all have the same effect: `⎕EX'F.MB.M'` or `F.⎕EX'MB.M'` or `F.MB.⎕EX'M'`. `⎕WN` can also take a dotted character vector argument and return a dotted result:

```
(⎕WN'F.MB')↳,⊂'F.MB.M'
```

In Visual Basic, you can use the dot syntax to access properties and invoke methods. For example: **Application.Workbooks.Add()** calls method **Add** with no arguments from the collection object returned by the **Workbooks** property of … Note that the value of a property may be an object.

Dyalog APL dot syntax for functions extends to (niladic and monadic) methods, as in VB. *eg*

```
F.Close
F.CAL.KeyPress¨↳⍴⊂'RC'
F.CAL.Size←3 ↳×F.CAL.GetMinSize
F.CAL.MouseDown 77 13 1 0 ⋄ F.CAL.MouseUp 77 13 1 0
F.RE.GotFocus 0           ⍝ F.RE.GotFocus⍬
F.RE.RTFPrint 0           ⍝ F.RE.RTFPrint⍬
F.RE.RTFPrintSetup 0
F.RE.RTFPrint F.RE.RTFPrintSetup'Selection'
#.GetEnvironment'MaxWS'
F.CAL.SelDate←#.DateToIDN(2003 12 1)
```

The last two examples invoke methods on the Root. Root methods and properties are exposed according to the setting of [Options][Object Syntax][Expose Root Properties] which, by default, depends on the value of the registry entry **PropertyExposeRoot** and not on the value of `⎕WX`.

§§ 3.3.3 Defined Operators in Object Space

3.3.3.1 What is the interpretation of `#.+.#.×` ? Can you space-qualify the inner product operator?

Dot (`.`) is not a token with a strict interpretation as a rational APL syntactic element; a variable, function or operator. To see this clearly, consider dot syntax as applied to user-defined operators.

Given an operator $\hat{O}_1$ in space `#.A.B`, this operator can be referenced from any point in the code by the notation `#.A.B.`$\hat{O}_1$. If dot is to be interpreted as an operator, then this dotted list of tokens involves an operator adjacent to another operator, which is a situation that would set a new precedent in APL.

The complexities of interpretation are not helped by the fact that dot is already used in APL in at least two other different places. It is a neutral symbol used to represent the decimal point. (Perhaps in a later version of APL this symbol will be supplied by the decimal symbol in [Control Panel][Windows Regional and Language Options].) Dot is also used for the primitive *inner* (`.`) and (irrational) *outer* (`∘.`) *product* operators. Normally, once a symbol or token has been used to represent an operator then it must always represent an operator (see APL Linguistics in *Vector Vol. 2 No. 2 p118* for some discussion of this). Therefore the dot in dot-syntax should be assumed to be an operator, given no other evidence to the contrary. (Remember, however, that the *reduce* operator (`/`) and the *replicate* function (`/`) unfortunately exemplify such a contradiction, albeit tolerated. Can you think of another?)

As a rule of thumb, the meaning of a dot (the big dot • here) may be (partially) interpreted by the class of the token to its left. In the special *'extra-APL'* case of a decimal point, the symbol immediately to the left clearly must be a numeric digit or space. The interpretation of the class 2 case is outlined in `Module 11`. As we have seen above, the class 9 case includes various classifications of right 'operand'.

• Larg Class	Syntax (Grammar)	Semantics (Meaning)
(0)	*D*•...	**Decimal number < 10**
(0)	*D*..*D*•...	**Decimal number**
(0)	•*D*...	**Decimal < 1**
1		Label
2	..*RefArr*•...	**See Module11**
2.1		Variable
2.2		Field
2.3		Property
3	..*f*•...	**Inner Product**
(3)	..∘•...	**Outer Product**
3.1		Canonical Function
3.2		Dynamic Function
3.3		Derived Function
3.6		External Functions & Methods
4.1		Canonical Operator
4.2		Dynamic Operator
(9)	..*Ref*•...	**Dot syntax**
9.1		Namespaces
9.2		GUI Object
9.3		Instances of Classes
9.4		Classes
9.5		Interfaces
9.6		.NET Classes?

3.3.3.2 What conclusions about the tokens involved can be drawn from the syntax of the statement `#.A.B[K]` or `#.A5.B` or `#.A .5` or `B.0` or `#.5`?

3.3.3.3 Ask for the next module on the **Session Object**. How did you get on with Module 3? Was it easy to follow? ☺.