Module12: Dynamic Programs

Dynamic Programming (of functions and operators) is an exciting alternative method of program specification to canonical function definition. Dynamic Programs have advantages and also some disadvantages with respect to the usual (canonical) form of programming. Advantages include; clarity for short algorithms, dynamic creation of small localised programs for in-line application, and *more direct control over the power of pure APL notation*. Disadvantages include decreased semantic density, missing features (such as line labels, branching (→), control structures, ⎕*PATH*, ⎕*CS* and ⎕*MONITOR*), partially implemented features (such as ⎕*STACK*, ⎕*STATE*, ⎕*REFS* and ⎕*AT*) and limited and less intuitive tracing facilities. Some of the gaps narrow with each new version of Dyalog.

In practise, perhaps the most serious disadvantage relates to the limitations encountered in tracing code, particularly in programs that have been defined inside canonical programs. Other irritations relate to the inability to use DFns directly as callbacks or the difficulty of calling non-result returning functions without causing a *VALUE ERROR*. The last point is particularly significant in GUI/OLE programming where one has no control over the shyness of the supplied methods. This limitation can be circumvented using *execute* with a dummy result, as in `sink←⍎'foo rarg⋄0'` when *foo* itself returns no result.

§ 12.1 Direct Definition

In Dyalog APL it is possible to directly define an ambivalent function using function specification or *direct function assignment*. Thus a name is given to the result of a *function expression* via the assignment arrow (←) with a name (eg *f*) on its left and a function (eg +) on its right. This syntax implies that assignment be in the class of dualistic niladic operator (where the right operand function may be ambivalent), if assignment were to be formally classified.

```
f←+    ⍝ Session statement (1)
```

A monadic call to *f* will apply the *prefix* function *identity* or conjugate (+) and return the Rarg.

```
f 5
 5
```

A dyadic call to *f* will add the left to the right argument via *infix* function *plus* (+).

```
3 f 5
 8
```

Reference to left and right arguments may be totally elided in statement `f←+` because the definition (+) is ambivalent and unambiguous in its argument(s).

Some more complex function expressions can be expressed by means of operators. For example,

```
f←↓∘⌽∘↑ ⍝ Used monadically on a vector of vectors
```

However, if one wished to express an algorithm involving left and right arguments in **arbitrary ways**, then the limitation of an assigned function expression to the form

{left arg} (function expression) right arg

is too restrictive

This restriction ***could*** be alleviated by invoking symbols ⍺ and ⍵ to represent implicit left and right arguments to an assigned function. This interpretation of ⍺ and ⍵ originates from the models of **direct definition** employed by I.P.Sharp in 2nd generation SharpAPL and from earlier APL publications.

Then the ambivalent definition in statement (1) above ***could*** be accomplished by **two** specifications.

```
f←+⍵      ⍝ Overwrites any monadic fn definition
f←⍺+⍵     ⍝ Overwrites any dyadic fn definition
```

Subsequently, the dyadic form of *f* above **could** be replaced by a new dyadic function '*under*'

```
f←⍵÷⍺     ⍝ Divide rarg by larg (cf larg over rarg)
```

And such a function ***could*** be called without having to give it an explicit name.

```
      4(⍵÷⍺)3
0.75
```

Bear in mind that any name given to an ambivalent function has to be sufficiently general a term as to be suitable for both the monadic and the dyadic context. (The APL primitives actually change their names in the different contexts: *eg* `⊨5≡+5` is read as "it is necessarily true that five matches *identity* five" whereas `⊨8≡3+5` is read as ".. eight matches three *plus* five".)

The functions dyadic `catRow` and monadic `justRow` ***could*** then be defined by direct function definition:

```
      catRow←↑(↓⍺),↓⍵                 ⍝ Catenate Rows
      justRow←(-+/¨' '∘=∘⌽¨⍵)⌽¨⍵       ⍝ Right Justify Rows
```

§§ 12.1.1 Programming DFns

Programming DFns (dynamic functions) is very like this, except that the essential function definition must be surrounded by braces (`{}`) thus, dynamic functions ***can*** be defined by:

```
      catRow←{↑(↓⍺),↓⍵}               ⍝ Catenate Rows
      justRow←{(-+/¨' '∘=∘⌽¨⍵)⌽¨⍵}     ⍝ Right Justify Rows
```

12.1.1.1 Define a square root DFn, `sqrt`, such that

```
      sqrt ⍳4
1 1.414213562 1.732050808 2
```

Consider the *rank* idiom - the **shape of the shape** of an array (`⍴⍴Arr`). `rr←⍴⍴` gives a `SYNTAX ERROR` because the left-most rho (`⍴`) cannot take a function Rarg – the right-most rho (`⍴`). However the token string `3∘` *is* consistent with a right-most rho (`⍴`) as in `3∘⍴` because jot is an operator.

We need to construct a genuine function so that function assignment can capture the derived rank idiom.

```
      rr←⍴∘⍴      ⍝ Assignment of a function expression
```

But function `rr` is ambivalent and involves a **reshape of shape** algorithm, so `⊨1 1≡2 rr ,3`. By calling `⍴∘⍴` the rank idiom it is clear that the dyadic application has been completely overlooked.

The (monadic) rank idiom may be captured in the dynamic function

```
      rr←{⍴⍴⍵}    ⍝ Assignment of a monadic dynamic function
```

In this case there is, as yet, **no dyadic form**. We could define a dyadic form

```
      rr←{⍺⍴⍴⍵}   ⍝ Assignment of a dyadic dynamic function
```

But this overwrites the previous definition, which means there is now **no monadic form**. We need a mechanism for assigning an ambivalent function.

First note the following features of dynamic function definition.

0. Let `⍵` represent Rarg and `⍺` Larg.

1. Any number of diamondized expressions (segments) may be included within the braces `{..⋄..⋄..⋄...}`.

2. The first expression (from left to right) that explicitly returns a result will terminate the function at that point and return that result.

3. All variables created in expressions in segments on the way to the final result-bearing segment are automatically shadowed prior to assignment.

4. A default left arg ⍺ may be provided simply by assigning ⍺ to a suitable default value. This assignment takes place only if `⊢(,0)≡⎕NC'⍺'` , *ie* if no left argument has been supplied.

Assignment of a default left arg by `⍺←…` provides a way to define an **ambivalent** function as long as a default ⍺ can be found which will provide the appropriate monadic form. The dyadic function has to have a natural monadic case such that the dyadic case with some specific Larg leads to the monadic case. This is possible for a few primitive functions. For example *divide* and *reciprocal* are such that

```
3 {⍺←1⋄⍺÷⍵} 4
0.75
```
and `{⍺←1⋄⍺÷⍵} 4` `0.25` , and also *power* and *exponential*

```
3 {⍺←*1⋄⍺*⍵} 4
81
```
and `{⍺←*1⋄⍺*⍵} 4` `54.59815003` , and *log* and *ln*

```
3 {⍺←*1⋄⍺⍟⍵} 4
1.261859507
```
and `{⍺←*1⋄⍺⍟⍵} 4` `51.386294361` , and ..

```
0 {⍺←⍞⍳⍴⍞⋄⍺↓⍵} 4
4
```
and `{⍺←⍞⍳⍴⍞⋄⍺↓⍵} 4` `4` , and *minus* and *negate*

```
3 {⍺←0⋄⍺-⍵} 4
¯1
```
and `{⍺←0⋄⍺-⍵} 4` `¯4` , and somewhat

```
3 {⍺←0⋄⍺+⍵} 4
7
```
and `{⍺←0⋄⍺+⍵} 4` `4` , but monadic *identity* actually applies to non-numeric data too and therefore `~⊧{⍺←0⋄⍺+⍵}Arr` `Arr` so the function could give a `DOMAIN ERROR`.

The beautiful design of the APL 1 primitive functions is an excellent model for the construction of user-defined functions. Primitive functions apply to arguments of various types and various ranks in meaningfully related ways, like much basic arithmetic notation applies unchanged in the complex domain. Thus in Sharp APL, and now in Dyalog APL version 11, *and* (∧) and *or* (∨) have been generalised to *lcm* and *gcd* because the Boolean cases follow as a natural consequence of the more general definitions of *lowest common multiple* and *greatest common divisor* (or highest common factor). Furthermore, the monadic and dyadic definitions of primitive functions are usually closely related in meaning, as in the classic case in arithmetic of *negate* and *minus* (-).

The above 4-point scheme for defining dynamic functions is not yet general enough even to model ambivalent primitive functions unless we explicitly use *execute* (⍎) in a construct such as

```
cross←{⍎⊃(b,~b←(,0)≡⎕NC'⍺')/'+⍵' '⍺+⍵'}
```

Even if we add the following 5th point, this limitation is still present.

5. DFns may be nested within dfns in the same way as canonical functions may be nested. *eg*

```
unwrap←{(⍵≠⎕AV[3+⎕IO]){⍺\⍺/⍵}⍵}⍝ Replace <LF> with blanks.
```

Without the ability to jump over diamondized segments, many algorithms become difficult to program. Nevertheless we already have a useful new form of function definition that yields some new idioms.

`{⍵}`	⍝ Function (**dex**) which returns the right argument
`{⍺}`	⍝ Function (**lev**) which returns the left argument
`{}`	⍝ Function (**sink**) which does not return any result

Another useful function idiom for converting a niladic form to a monadic form is simply `{niladic}`.

Note that the two forms of function assignment – assignment of a function expression and assignment of a dfn to a name – are not mutually exclusive, but may be combined into *hybrid* function expressions

```
withoutA←{⍵~'A'}¨      ⍝ Remove character A from each substring in a character array
{'[',⍵,']'}∘⍕          ⍝ Bracket a number
{'$',⍵,'.00'}¨∘⍕       ⍝ Dollarize integer dollars
{('F.C',⍕⍵)⎕WC'Circle'(0 0)⍵('FCol'(?3⍴255))}¨ ⍝ Draw circles of radii ⍵
```

or

```
{_←⎕NA'U4 kernel32|GetDriveTypeA <0T' ⋄ ⍵,GetDriveTypeA⊂⍵,':\'}¨
```

Beware of unreadable code wherein meaning can be lost due to essentially nameless proliferation of ⍺'s and ⍵'s from different contexts. Beware of dense strings of tokens without any context-relevant variable names – what Stephen Taylor has called ***semantic density***. It is easy to loose the fundamental meaning of an expression when there are no semantic clues in the form of well-chosen user-defined variable names.

```
{}11↑⍞{⍺←⎕A⋄⍵↑⍨↑⍨,/,¨∘.,\(⌈(⍴⍺)⊛⍵)⍴⊂⍺}1111
```

In the right dozes, dfns can clarify meaning

```
      ∇ Suggestions←howSpell TheWord;WD;Words
[1]     'WD'⎕WC'OLEClient' 'Word.Application'
[2]     Words←WD.GetSpellingSuggestions TheWord
[3]     Suggestions←{(Words.Item ⍵).Name}¨⍳Words.Count
      ∇
      howSpell'Helleo'
 Hello  Helle  Helloes  Heller  Hellion  Halloo  Hellos  Hallo  Hej
```

12.1.1.1 Show how the functions `{⍵⎕FAPPEND 1}` and `{⎕FREAD 1 ⍵}` may be used to append or read many file components in a single operation.

Simple idiomatic algorithms may be expressed neatly, for example in

```
sortVec←{⍵[⍋⍵]}
getParent←{(-1++/∧\⌽⍵≠'.')↓⍵}
trimCVec←{(~(∧\' '=⍵)∨(⌽∧\' '=⌽⍵))/⍵}
justLeft←{(+/∧\' '=⍵)⌽⍵}
getPath←{'\',⍨(-(⌽⍵)⍳'\')↓⍵}
```

but more complex algorithms deserve more space. Consider, for example, the marvellous Box-Mueller algorithm from Professor Tony O'Hagan, which deserves to be implemented as a new APL primitive function *plus or minus* (±):

```
±←{⍵⍴↑(⌈(×/⍵)÷2){(⊂(¯2×⊛⍺{(?⍺⍴⍵)÷⍵}⍵)*0.5)×¨1 2∘.○⊂2×⍺{(?⍺⍴⍵)÷⍵}⍵}¯1+2*31}
```

The dyadic form might be such that `⍺±⍵ ↦ ⍺+±⍵` , *ie* it might have the ambivalent definition

```
±←{⍺←0 ⋄ ⍵⍴↑(⌈(×/⍵)÷2){ .. }¯1+2*31}
```

Clearly we need to break this up if we want to be able to read and understand the function easily.

§§ 12.1.2 MultiLine DFns

In order to make a long complicated dynamic function definition more readable (and more writable) it is necessary to break it into manageable comprehensible chunks.

6. You may break a line in a dfn at any diamond (`⋄`), after a left brace (`{`) or before a right brace (`}`).

It is not possible to enter a multi-line dfn in the APL session (although **Shift+Enter** as opposed to **Enter** could be defined as *continue* (↵) as opposed to *enter* (⍎).) You may enter a multi-line dfn in the editor as a stand-alone dfn, or as part of a larger canonical function.

For example, you could define a function to determine the mean value of a numeric vector in the session

```
      mean←{(+/⍵)÷⍴⍵}                    ⍝ arithmetic mean
```

or as a 1 line function within a canonical function

```
      ∇ Variation
[1]     Nos←,⍞                           ⍝ input numbers
[2]     mean←{sum←+/⍵⋄num←⍴⍵⋄sum÷num}    ⍝ arithmetic mean
[3]     Nos-mean Nos                     ⍝ difference from average
      ∇
```

or as a multi-line function within a canonical function

```
      ∇ Variation
[1]     Nos←,⍞                           ⍝ input numbers
[2]     mean←{sum←+/⍵                    ⍝ total
[3]           num←⍴⍵                     ⍝ number of numbers
[4]           sum÷num}                   ⍝ arithmetic mean
[5]     Nos-mean Nos                     ⍝ difference from average
      ∇
```

or as a stand-alone **multi-line dynamic function**

```
      ∇ mean←{sum←+/⍵                    ⍝ total
[1]         num←⍴⍵                       ⍝ number of numbers
[2]         sum÷num}                     ⍝ arithmetic mean
      ∇
```

Note that the final comment will be lost unless it is placed inside the outermost brace.

12.1.2.1 Trace each of the above functions, using some arbitrary set of numbers for input. Check for global variables left in the workspace.

As well as completely empty lines or lines consisting entirely of diamonds or comments, it is also possible to have lines containing nothing but a single left brace `{`, or a left brace followed by a right brace `}{`, or a single right brace `}`. The following function has a valid header line and a valid closing line:

```
      ∇ compress←{                       ⍝ remove multiple blanks
[1]         (~' '⍷⍵)/⍵
[2]     }
      ∇
```

shown in `⎕VR` form. Alternatively, the function below is shown in `⎕CR` form:

```
to←{⎕IO←0                                ⍝ Sequence ⍺ .. ⍵
    from step←1 ¯1×-\2↑⍺,⍺+×⍵-⍺          ⍝ step default is +/- 1.
    from+step×⍳1+0⌈⌊(⍵-from)÷step+step=0  ⍝ ⍺ thru ⍵ inclusive.
}
```

12.1.2.2 Trace the line

```
      Eigen ?10 10⍴1000
```

where dfn `∇Eigen∇` is to be found in the distributed workspace ..\WS\MATH.DWS. Compare this function with canonical function `∇EV∇` in §§ 9.3.3

§§ 12.1.3 Guards and Error Guards

Imagine that dyadic execute (`⍎`) was defined to take a Boolean Larg (`BSc`) and a character string Rarg (`CVec`) whereby the character string was executed if the Boolean were `1`, *ie* `⍎←{⍺←1⋄⍎⍺/⍵}`, then this is something like a *guard* (`BSc:Expr`) in dynamic programs. A guard, signified by a single colon (`:`), is

neither a primitive function nor an operator but a new ungrammatical symbol, only available within a dynamic program, with the following meaning:

```
      BSc : ...                          :If BSc ⋄ ... ⋄ :End
```

An expression (returning *BSc*) to the left of the colon (:) does not need to be surrounded by parentheses and an expression to the right is not surrounded by quotes, as would be the case with the *execute* model.

A dfn may then be written as a series of segments each with an opening guard that determines whether or not the rest of the segment is executed. The first segment to be executed may then return the final result. For example, by analogy with the (atypical) circle function (○), we could call functions by number:

```
      ∇ fn←{
[1]         ⍺=1:+⍵  ⍝ identity
[2]         ⍺=2:-⍵  ⍝ negate
[3]         ⍺=3:×⍵  ⍝ signum
[4]         ⍺=4:÷⍵  ⍝ reciprocal
[5]         ⍺=5:*⍵  ⍝ e to power
[6]         ⍺=6:⍟⍵  ⍝ natural log
[7]     }
      ∇
       5 fn 2 fn 3 ↳ *-3 ↳ 0.04978706837
```

12.1.3.1 Write a single line dfn which discloses (once) an array if it is scalar and enclosed.

Hint: .. rank zero and depth of magnitude greater than one.

Imagine ⎕*TRAP* had been defined dyadically with the error numbers on the left and the execute cutback expression on its right: this is something like an ***error guard*** (*NVec::Expr*) in dynamic programs.

A error guard, signified by a double colon (::), is neither a primitive function nor an operator but a new grammatical **pair of symbols** (going in an unfortunate **J** direction), only available within a dynamic program and with the following meaning:

```
      NVec :: ...                        :Trap NVec ⋄ ... ⋄ :End
```

The expression (returning *NVec*) to the left of the double colon (::) does not need to be surrounded by parentheses (making :: impossible to interpret even as a dualistic niladic operator) and the expression to the right is not surrounded by quotes, as would be the case with the ⎕*TRAP* model.

7. Use *guard* (:) to replace *branch* (→) or *:If*, and *error guard* (::) to replace ⎕*TRAP* or *:Trap*

The expression to the left of an error guard evaluates to a vector of error numbers. The expression on the right of the error guard is evaluated in the event that one of these errors is generated by subsequent lines (or segments of a line). For example the following function will return ⎕*DM* in the event of any error in the second segment.

```
      cover←{0::↑⎕DM⋄⍎⍵}
      cover ?3 3⍴3
DOMAIN ERROR
cover[] cover←{0::↑⎕DM ⋄ ⍎⍵}
                       ^
```

Note that the trap is not set when executing the expression immediately to the right of an error guard, making trap loops less likely. As with ⎕*TRAP* it is possible to have a hierarchy of traps set, or a series of

traps performing different functions. The following example attempts to tie a file, and depending on the error, performs a different alternative, each alternative still being covered by the traps above.

```
open←{
      0::0
      22::⍵ ⎕FCREATE 0
      24 25::⍵ ⎕FSTIE 0
      ⍵ ⎕FTIE 0
}
```

In this case, the last line is the first to be evaluated. If a `FILE NAME ERROR` (`22`) occurs then an attempt is made to create the file. If any error occurs at this point then the function returns `0`.

§ 12.2 Extended direct Definition

§§ 12.2.1 Programming DOps

In canonical form, programming operators is very much like programming functions. Only the header line is slightly different with parentheses round the effective derived function. Likewise, programming DOps is similar to writing DFns but there is no header line to distinguish the two sub-classes. One clue as to the program class when examining a dynamic program is given by the colour of the braces. It is possible to select in [Options][Colours][Syntax][Element] either D-Op (dyadic) or D-Op (monadic) – what we call dualistic and monistic to distinguish from functional form (see *Vector Vol.2 No.2 p118*). Pairs of braces can take any of three different colours, one for dfn, one for monistic dop and one for dualistic dop.

How does the interpreter know what class a program is? The left operand in a dop is represented by the double-symbol `⍺⍺` and the right operand in a dualistic dop is represented by the double-symbol `⍵⍵`. If the dual symbol `⍵⍵` exists within the braces (not counting its presence in sub-braces) then the program within the braces must be a dualistic operator as only a dualistic operator has a right operand. If there is an `⍺⍺` but no `⍵⍵` then the program must be a monistic operator, and if there is no `⍵⍵` then the program is class 3 (a function), and the braces are coloured accordingly.

8. Use `⍺⍺` to represent the left operand and `⍵⍵` to represent the right operand of a dynamic operator.

So we could define the primitive monistic *commute* operator (⍨) to be

```
comm←{⍺←⍵ ⋄ ⍵ ⍺⍺ ⍺}
```

First if there is no left argument to the derived function then it is taken to be the same as the right argument. (Try primitive *commute*, `+⍨4 ↦ 8` .) Then the function left operand (`⍺⍺`) is passed the arguments `⍺` and `⍵` in the reverse order - `⍺` becomes the right argument and `⍵` the left, exactly as required by the definition of commutation.

```
4 -comm 3 ↦ 4-⍨3 ↦ 3-4 ↦ ¯1
10 ⋆comm 3 ↦ 10⋆⍨3 ↦ 3⋆10 ↦ 59049
```

Or we could cover the **J** grammatical concept of ***hook*** with the dualistic operator `hook`:

```
hook←{⍺←⍵⋄⍺ ⍺⍺ ⍵⍵ ⍵}
4 ⋆hook- 3 ↦ 4⋆∘-3 ↦ 4⋆-3 ↦ 0.015625
```

DOps can be multi-line, and they can have guards, just like functions.

```
pow←{                    ⍝ Explicit function power.
      ⍺=0:1
      ↑{⍵}∘⍺⍺/(⍳⍺),⊂⍵
```

```
}
```

The derivative operator from elementary calculus takes a single monadic function and returns a monadic derived function, the gradient function. So the operator is monistic and the derived function is monadic. The derivative of a function, $f(x)$ is $f'(x)$ where $f'(x)=df/dx \approx (f(x+dx)-f(x))/dx$, or to a better approximation $f'(x)=df/dx \approx (f(x+dx)-f(x-dx))/2dx$. This is clearly what we have on line [2] below. ⍺⍺ represents the function operand $f(x)$ and ⍵ represents the function argument (x). The operator ∆ then models the derivative operator d/dx.

```
      ∇ ∆←{    ⍝  derivative operator
[1]         dw←⍎'1E¯6'
[2]         ((⍺⍺ ⍵+dw)-⍺⍺ ⍵-dw)÷2×dw
[3]     }
      ∇
```

The derivative of $3x^4$ with respect to x is the function $12x^3$ for all x, for example for x=3, 4 and 5.

```
      {3×⍵*4}∆ 3 4 5 ↳ {4×3×⍵*3}3 4 5 ↳ 324 768 1500
```

Symbolically, one could write

```
      {a×⍵*n} ∆ ↳ {a×n×⍵*n-1}
```

You can see some other examples of dynamic operators in the ..\WS\DFNS.DWS workspace. You can download the latest version of this useful workspace, as well as an article DFNS.PDF by John Scholes, from [Download Zone] of www.dyalog.com. You can also find examples in the *Language Reference* and in the 7.3 or 8.1 new release Help files, downloadable from [Document Download Zone].

One particularly useful operator is *memo* which remembers the result of a function as applied to any particular argument. If called again identically through *memo* then the result is not recalculated, but just returned directly from memory. Functions without side-effects are suitable for memoization. A second example of an operator created by Phil Last, who like John Scholes is a prolific author of fabulous D programs, is *else* which, depending on Boolean ⍺, applies the left operand or the right operand to the derived function argument ⍵.

```
else←{           ⍝ Condition f else g ...
     ⍺:⍺⍺ ⍵      ⍝ True: apply left operand.
     ⍵⍵ ⍵        ⍝ False: apply right operand.
}
```

Join the Dyalog dynamic functions mailbox group dfns@dyalog.com for discussions and live examples and issues relating to general dynamic programming, led by the main protagonists.

§§ 12.2.2 Idioms and Utilities

The entire Finnish APL Idioms list, maintained by Veli-Matti Jantunen, with over 500 entries, has been rewritten as dynamic programs. Every canonical idiom has a dynamic counterpart.

12.2.2.1 Give the following idioms meaningful names. Ask yourself if the word you have chosen reads well in the context of its use. Add some more of your favourite phrases…

```
{⍵[⎕AV⍋⍵;]}
{⍵/⍳⍴⍵}
{(+/⍵)÷⍴⍵}
{↑(-⍳⍴⍵)↑⍤¯1⍵}
{⎕AV[(⎕AV⍳⍵)-48×⍵∊⎕A]}
```

Beware of illegibility like

```
{
      mlr←⌽3⍴⍵⍵,⍳(aa←⍺⍺)/m←0
      ⍺←(aa←{⍵}∘aa)/m←1
      l r←-1↓r⌊|{⍵+r×0>⍵}(mlr⌊r←3⍴(⍴⍵),⍴⍺)[⍸m×⍳3]
      ↑aa⍲(⊂[l↑⍳⍴⍺]⍺),[-0.1-⍳1]⊂[r↑⍳⍴⍵]⍵
}
```

or inscrutable (until §12.3) one-liners like

```
{'*'∨.≠(⍴⍵)↑⎕←'*+'/⍕¨⍵{(⍺+.=⍵),⍺{+/⊃⌊/+/⍷⍨(⊂∪⍺)∘.=⍷⍨(⍺≠⍵)∘/⍷⍨⍺ ⍵}⍵}⍣⍵:∇ ⍵}
```

§§ 12.2.3 Object.Object..Object.Operator Rationale

DFns may be space qualified either by name, or without a name. For example, given function

```
plus←{⍺+⍵}
```

then

```
3 #.plus 4 ↦ 7
```

and

```
3 #.{⍺+⍵} 4 ↦ 7
```

In this respect dfns are just like user-defined functions, object methods or primitive functions. All of the rules stated in Module 11 apply to dfns as they do to other functions.

The same rules also apply to space-qualified dops. If we had a *natural log* dfn (`ln`) in `#`

```
ln←{⍟⍵}
```

and a derivative operator `∆` in `#.A.B`, then whilst *in* any other space (see `⎕SE.CurSpace`) we could find the derivative of *ln*(*x*) at any points *x*, say at 6 and 7:

```
#.ln #.A.B.∆ 6 7 ↦ ÷6 7 ↦ 0.1666666667 0.1428571429
```

This all seems quite natural and useful.

```
h←f ñ_D.Ô                    ⍝ Run operator(s) Ô in space(s) ñ_D with operand...
```

```
h←f ñ_D.Ô g                  ⍝ h is space-array of derived functions...
```

The problem from the point of view of rational APL grammar is that any attempt to argue that the *dot* of dot syntax should be considered to be an operator is now confronted with the situation where *an operator has an operator operand*. This introduces entirely new APL grammar whose implications have been explored in New Foundations in *Vector Vol.20 No.1*. Either you can accept the pragmatic rationale for `Object.Operator` syntax given above or you may seek a deeper justification elsewhere. (Note that in advanced mathematics, the first derivative operator (d/d*x*) applied to the first derivative operator (d/d*x*) gives the second derivative operator (d^2/dx^2) so there is certainly a precedent in mathematics.)

§ 12.3 Recursion

§§ 12.3.1 Recursive Functions

Most problems that can be solved with iteration can also be solved with recursion. One advantage of recursion is that the program often looks more like the original formula.

It has always been possible to write recursive functions in APL by referring to the function itself within its own definition. Thus niladic `foo` defined by `⎕fx'foo' 'foo'` is infinitely recursive, as is monadic `foo` defined by `foo←{foo ⎕←⍵}`. The essential novelty in recursive dfns is the possibility of

writing *unnamed* recursive functions. This is implemented simply by using the symbol del (∇) (*function self)* inside a dfn to refer to the entire dfn itself. So the useless monadic infinitely recursive dfn `foo` above may be replaced by the equally useless `{∇ ⍵}` which may or may not be assigned an arbitrary name.

Many examples of useful recursive dfns are to be found in the supplied DFNS.DWS workspace. Here are a few examples from the workspace.

Power, as in x^y where y is a positive integer, is just repeated multiplication; $x \times x \times .. x$, y times. This can clearly be written with a looping solution, or in APL without a loop:

```
      ×/4⍴3 ↳  81
```

Alternatively it may be written as a recursive dfn:

```
      pow←{⍵=0:1 ⋄ ⍺×⍺ ∇ ⍵-1}
      3 pow 4 ↳ 3*4 ↳ 81
```

Factorial is a classic case of recursion where factorial of an integer, x, may be written as $x(x\text{-}1)(x\text{-}2)..1$ which translates directly into dfn

```
      {⍵=0:1 ⋄ ⍵×∇ ⍵-1} ↳ {!⍵}
```

with the added value of `1` for factorial zero which enables the function to end the recursion. An alternative program for factorial may be written

```
      {⍺←1 ⋄ ⍵=0:⍺ ⋄ (⍺×⍵)∇ ⍵-1} ↳ {!⍵}
```

This 'tail-recursive' form turns out to be faster because the segment containing the *function self* returns the result of self immediately as the result of the entire function whereas the first algorithm multiplies the result of *function self* by `⍵` before returning a result as the result of the entire function, making certain internal interpreter optimisations impossible. To be tail-recursive, the answer ultimately returned by the top-level call to the function must be identical to the value returned by the very bottom level call. The ultimate answer, 81, is not the same as the deepest level return value which was 1, so power is not a tail-recursive function but the second (faster) form of factorial is tail-recursive.

There are many examples of beautiful mathematical functions with an elegant recursive definition. The greatest common divisor may be programmed as

```
      {⍵=0:⍺⋄⍵ ∇ ⍵|⍺} ↳ {⍺∨⍵}
```

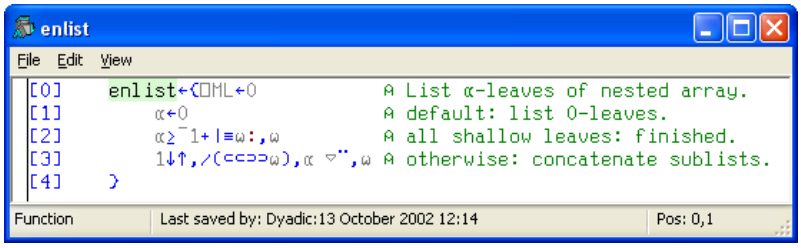
See also algorithms for prime factors or the ancient algorithm for identifying prime numbers first espoused by Eratosthenes of Cyrene who lived around 275-195 BC.

One well known recursive algorithm is that for obtaining the determinant of a matrix. The essence of the method is visible in the following multi-line dfn:

```
det←{⎕IO ⎕ML←0                                  ⍝ Determinant of matrix ⍵.
     1{                                         ⍝ initial accumulator.
          0 0≡⍴⍵:⍺                              ⍝ null matrix: finished.
          (⍺×⊃⍵)∇ 1 1↓⍵-⍵[;0]∘.×⍵[0;]÷⊃⍵         ⍝ accumulator ∇ sub-matrix.
     }⍵
}
```

Notice how the system variables are automatically localised. Notice also how subtraction and multiplication are at the core of the algorithm, prompting Iverson to propose that the dualistic monadic *dot* operator be introduced such that `{-.×⍵}` be the *determinant* of a matrix argument. He further calls `{+.×⍵}` the *permanent* function.

Nested arrays offer much scope for recursive functions. It was not a coincidence that the *each* operator was introduced at the same time as nested arrays. For example, a recursive definition of *enlist*, which may be expressed simply as `{⎕ML←1⋄∈ ⍵}`, is given in function `enlist` in the DFNS workspace.



12.3.1.1 Study the function `refs` below (also to be found in the DFNS workspace).

```
refs←{                                  ⍝ Vector of sub-space refs for ⍵.
     ⍺←⊢ ⋄ (⍴,⍺)↓⍺{                     ⍝ default exclusion list.
          1∊⍵=⍺:⍺                       ⍝ already been here: quit.
          ⍵,(↑∇∘⊃¨⍵/⍨⍵∊⍺),↓⎕NL 9)        ⍝ recursively traverse sub-spaces.
     }⍵                                 ⍝ for given starting ref.
}
```

Load distributed workspace WDESIGN.DWS and trace `refs #` with the tracer in [Options][Configure][Trace/Edit][Classic Dyalog mode]. Useful examples include:

```
(refs #).⎕wx
(refs #).⎕NL 2
(refs ⎕SE).(⍴∘⎕CR¨3↑↓⎕NL 3)
```

Aside: The functions `Legendre`, `Hermite` and `Laguerre` in the distributed MATH.DWS workspace represent the sets of (function) solutions to three commonly applicable differential equations. See, for example, http://www.efunda.com/math/Laguerre/index.cfm. These (infinite) sets of orthogonal functions are the eigenfunctions of the corresponding differential operator. A 'recurrence relation' relates each function in a set to neighbouring functions. Thus the three functions above may be replaced by recursive definitions.

§§ 12.3.2 Recursive Operators

There are two distinct kinds of self reference for recursive D-Ops. The symbol `'∇'` may be used to refer to the current derived function - the operator bound to its operand(s). When the operands are functions, this is the most frequently used form of self reference. However, if the operands are arrays, we often need a recursive reference to the operator itself and then we would use the double symbol `'∇∇'`.

An example of the *first type* of recursion within a dop is given by the `while` operator. As long as the right operand function `⍹⍵` acting on the argument `⍵` returns `1` apply the derived function again to the result of the left operand function `⍺⍺` applied to `⍵`, otherwise return `⍵`.

```
while←{                 ⍝ Conditional function power.
     ⍹⍵ ⍵:∇ ⍺⍺ ⍵        ⍝ While ⍹⍵ ⍵: apply ⍺⍺ ⍺⍺ ··· ⍵.
     ⍵                  ⍝ Otherwise: finished.}
```

A fascinating example of the *second type* of recursive operator is given in function `kt` in ..\DFNS.DWS.

The most general second type of operator recursion involves a situation whereby the function operands of an operator change at each level of recursion. A simple example is given by

```
comp←{
     ⍺=0:⍺⍺ ⍵
     (⍺-1)⍺⍺∘⍺⍺ ∇∇ ⍵}
```

A potentially very useful example of operator recursion is given by the *function determinant* operator. Imagine you had a 2 by 2 matrix of functions (whose APL representation is as yet undefined),

$M(x) = \begin{pmatrix} p(x) & q(x) \\ r(x) & s(x) \end{pmatrix}$

then the function determinant is defined mathematically as

$det(M(x)) = p(x)s(x)-q(x)r(x)$

where multiplication and subtraction are now operators like

```
times←{(αα ⍵)×(⍺⍺ ⍵)}
minus←{(αα ⍵)-(⍺⍺ ⍵)}
```

and the essential recursive APL operator would contain a line something like

```
(αα times ⊃⍵⍵)∇∇ 1 1↓⍵⍵ minus ⍵⍵[;0]∘.times ⍵⍵[0;] divide ⊃⍵⍵
```

if `⍵⍵` was allowed to be a matrix of functions… For a larger size square matrix of functions, the recursive determinant operator would take different function args at each level.

12.3.2.1 The determinant of the function matrix of problem 11.3.2.1 is clearly `1`. Consider how you might express this in executable notation.

§§ 12.3.3 Biological Beauties

Much of the beauty of nature rests on self-similarity - the fact that patterns may be repeated at different size scales. All sorts of natural objects from crystals and sea shells to fern leaves and onions may be modelled by recursive functions. (See Stephen Wolfram's *New Kind of Science* for an extensive monumental computational analysis of self-similarity.)

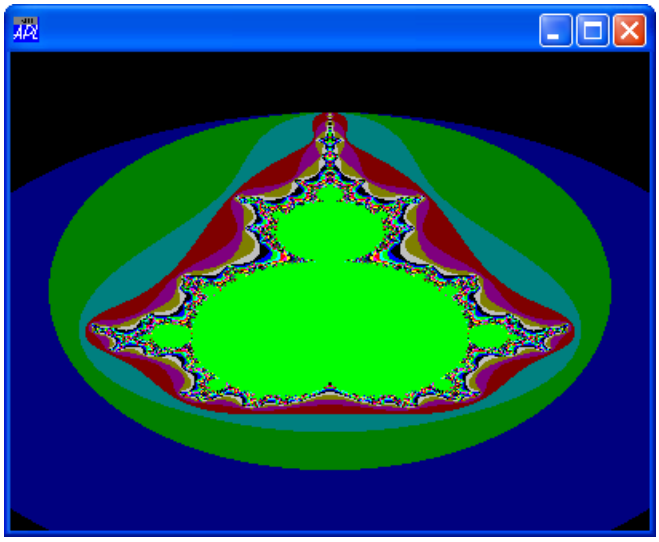
One of the first discoveries in this vast new subject was made by Gaston Julia in 1918 and developed and visualised via computer by Benoit Mandelbrot around 1975. They found infinite depth in simple iterative algorithms. We can capture in APL the Mandelbrot set using his algorithm, $Z=Z^2+C$. We can picture the set of points on the complex plane whose modulus never exceeds 2 under this iteration. These points are connected and produce a line on the plane whose dimension may be considered as not 1 but *fractional*.

The essential lines are the second, third and last in the recursive function, `square`, defined below. The second line calculates the square of a complex number and adds the position in the complex plane under consideration. The third line determines whether the modulus is greater than 2 (in which case it will diverge and therefore is outside the set. The last applies the function recursively.

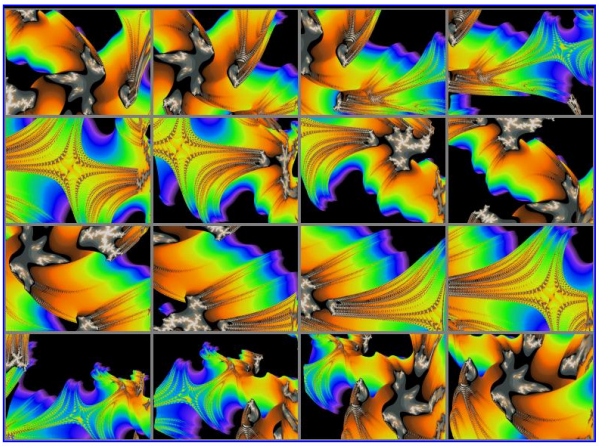
```
Mandelbrot;r;c;v;ADDR;BITS;Cu2;Zu2;x;y;⎕IO;cmap
 ⎕IO←1
 Xmin←¯2.5 ⋄ Xmax←1.5 ⍝ set X coord limits
 Ymin←¯1.5 ⋄ Ymax←1.5 ⍝ set Y coord limits
 r c←300 400 ⍝ number of rows and cols
 v←r×c       ⍝ number of pixels
 BITS←v⍴0    ⍝ initial colour black
 x←Xmin+((Xmax-Xmin)÷r-1)×0,⍳r-1 ⍝ X range
 y←Ymin+((Ymax-Ymin)÷c-1)×0,⍳c-1 ⍝ Y range
 Cu2←↑,x∘.,y  ⍝ r×c points (2 coords each) on complex plane
 Zu2←v 2⍴0    ⍝ zero initial value of Z at each point
 ADDR←⍳v      ⍝ address in bits vector
 cmap←↉2 2 2⊤0,⍳7
 cmap←(127×cmap)+(255×cmap)
 cmap[8;]←192
 'FRM'⎕WC'Form'('Size'(r c))('Coord' 'Pixel')('Picture' 'BMP' 2)('OnTop' 1)
 'BMP'⎕WC'Bitmap'('Bits'(r c⍴0))('CMap'cmap)
 1 ⎕NQ'FRM' 'Flush'
 square←{Zu2 Cu2 ADDR BITS←⍵
```

```
      Zu2←Cu2+((Zu2[;1]*2)-Zu2[;2]*2),[1.5]2××/Zu2   ⍝ Z←C+Z*2
      ⎕←+/~OUTu←~2<0.5*⍨(Zu2[;1]*2)+(Zu2[;2]*2)      ⍝ 2<|Z
      ∧/OUTu:                                        ⍝ all outside, QUIT
      ADDR←OUTu/ADDR                                 ⍝ remove outside addresses
      Zu2←OUTu⌿Zu2                                   ⍝ remove outside values
      Cu2←OUTu⌿Cu2                                   ⍝ remove outside points
      0=⍴ADDR:                                       ⍝ none to update, QUIT
      BITS[ADDR]←⎕←1+⌈/BITS                          ⍝ increment effective counter
      cbits←(r c)⍴256⊥⍉cmap[1+15|,r c⍴BITS;]          ⍝ recalculate colour from depth
      _←⍞'''BMP''⎕WS''CBits''cbits⋄0'                ⍝ set new colours
      ∇ Zu2 Cu2 ADDR BITS                            ⍝ recurr with subset
}
square Zu2 Cu2 ADDR BITS                             ⍝ go
```

The result is a beautiful picture, whose beauty is only limited by the graphical capability of the output medium.



This simple algorithm and the astonishing pictures that it can generate can be applied to the quaternionic (or the octonionic) domain with an identical mathematical algorithm. Prizes have been awarded to some fantastic 3D projections and 2D sections of quaternionic fractals. Many examples can be viewed on the internet, eg at http://www.lactamme.polytechnique.fr/Mosaic/images/JU.g2.0.16.D/display.html.



Such is the power and beauty of recursion by computer. From an examination of a number of recursive models we can extract a fundamental form which is at the heart of many of them. This recursive operator is named `CPA` from *Critical Path Analysis* where two distinct functions emerge from network analysis. The first is `FPA`, *Forward Path Analysis* where the network is analysed in a forward (time) direction.

Then there is the *BPA*, *Backward Path Analysis*, stage where the network is analysed from the other direction. These two functions together build a comprehensive description of the network.

For our examples below the recursive operator,

```
CPA←{ ⍵ Tree←(FPA CPA BPA) Trunk
      0=⍴trunks←⍺⍺ ⍵:''⍵⍵ ⍵
      trees←(⍺⍺ ∇∇ ⍵⍵)¨trunks
      ⍵ ⍵⍵ trees
}
```

reapplies the same function operands ⍺⍺ and ⍵⍵ at every level is replaced with the simpler operator

```
CPA2←{ ⍵ Tree←(FPA CPA BPA) Trunk
      0=⍴trunks←⍺⍺ ⍵:''⍵⍵ ⍵
      trees←∇¨trunks
      ⍵ ⍵⍵ trees
}
```

This operator can form the basis of the analysis of many nested structures in APL. *FPA* (or ⍺⍺) is a function which digs down one level into a structure, and *BPA* (or ⍵⍵) is a function which builds the final result, going backwards one level at a time.

So, for example, we could take ⎕*WN* as the *FPA* function which digs down into a GUI structure one level and use the simple construction {(⊂⍺),⊂⍵} for the *BPA*, backward pass construction. Thus

```
      ⍴⍴⍴⍴⍴⍴(⎕WN CPA{(⊂⍺),⊂⍵})'⎕se'
                  9   8     9   2     6   11     9   16     0   7
```

12.3.3.1 Consider the following *FPA* candidates and attempt to run an example:

```
{⎕CMD'Dir ',⍵}   ⍝ to examine the directory structure
{⊃¨⍵}            ⍝ to examine a nested array
⎕REFS            ⍝ to examin a function calling tree
⍵.⎕NL 9          ⍝ to examine namespace structure.
```

By means of this simple-looking operator many diverse subjects can be investigated and pictured (especially beautiful using OpenGL briefly discussed in Module 9): project plans, road systems, trees, leaves, lungs, veins and arteries, virtual particles, and other idealized and natural fractals of all sorts.

One of the many great things about APL is that it is usually imagination and not the programming language that is the limiting factor. With APL, what we can conceive we can achieve.

12.3.3.2 Please ask for the next module on **multi-threading** ☺.