Co-operators. Dyalog '11. Boston, MA

Abstract

Phil has had a long interest in extending the language with user defined operators, thinks too little use is made of them and believes that many APLers are a bit afraid of them.

He will develop one or two of them before your eyes and perhaps demonstrate one or two more.

At least one of them is actually quite useful and he cherishes a hope that one day it will be implemented as a primitive.

Preamble

I've been developing user defined operators since a time when it was only possible to simulate them by executing (⍎) prepared expressions including a function name passed as left argument to another function.

So when IBM announced a mechanism for users to define their own operators in about 1983 I already had a queue of them awaiting proper implementation.

When Dyalog announced D-fns with lexical scope in 1997 I unashamedly switched to using that as my standard notation for all new functions and operators and embedding D-fns for most amendments to old ones. So without apology I'll warn you that everything you'll see here is in D-notation.

Application specific operators tend to be rare, often restricted to two or three in a large application while I currently have a collection of forty-four more-or-less general purpose ones.

All those I'm going to show have the feature that they expect to run in concert with other instances of themselves which is why I've called them "co-operators". Now there's a difficulty here analogous to non-infix constructs such as control structures and the largely deprecated index and axis brackets whose action-at-a-distance properties might be thought by purists to be foreign to APL. I hope to demonstrate that co-operators are a natural consequence of infix notation. I don't intend thereby to endorse the inclusion in APL of either brackets or control structures which are neither functions nor operators and most of which can be replaced by operators that are both a more natural part of the language and, I believe, far more lucid in use.

Functions used as operands in the following (f, g, h, j, k, l) merely return expressions that reflect how they were called. They are defined thus:

```
      ⍪/⎕cr¨'fghjkl'
  f←{⍺←⊢ ⋄ '(',⍺,'f',⍵,')'}
  g←{⍺←⊢ ⋄ '(',⍺,'g',⍵,')'}
  h←{⍺←⊢ ⋄ '(',⍺,'h',⍵,')'}
  j←{⍺←⊢ ⋄ '(',⍺,'j',⍵,')'}
  k←{⍺←⊢ ⋄ '(',⍺,'k',⍵,')'}
  l←{⍺←⊢ ⋄ '(',⍺,'l',⍵,')'}

      f 0 g 1
(f( 0 g 1 ))
      j/h¨⍳2
 ((h 0 )j(h 1 ))
      k\⍳3
0  ( 0 k 1 )  ( 0 k( 1 k 2 ))
      ⊢∘l\⍳4
0  (l 1 )  (l(l 2 ))  (l(l(l 3 )))
```

A couple of things I might use here that I should point out hoping I'm not preaching to the converted:

I'm using Dyalog 13.0 that introduces the left and right functions (⊣) & (⊢). It also

enables the four idioms (⊣/), (⊣⌿), (⊢/) & (⊢⌿) that mean respectively: first element or column array; first major cell; last element or column array; and last major cell.

If a D-fn or D-op is called without a left argument then (⍺) is undefined and can be assigned once with any value including the very useful (⊢). An expression that assigns (⍺) is ignored at run-time if the fn or op is called dyadically.

A useful way to access the items of a 2-vector individually is to use reduction and refer to them as (⍺) and (⍵) respectively:

```
      {(f ⍺)g(h ⍵)}/⍳¨1 2
 ((f 0 )g(h 0 1 ))
```

A useful way to apply a function to only the second item of a 2-vector is to use scan with a "monadic" D-fn:

```
      {j ⍵}\⍳¨1 2
 0  (j 0 1 )
```

but note this is *not* the same as j\ that applies the function dyadically.

Reduction is rank reducing but not generally depth reducing:

```
      ≡ (0 1)(2 3)(4 5)(6 7)
2
      ≡ +/ (0 1)(2 3)(4 5)(6 7)
2
```

so that a vector reduction often requires (⊃) to disclose its scalar result:

```
      ⊃{(f ⍺)g(h ⍵)}/⍳¨2 3
((f 0 1 )g(h 0 1 2 ))
```

Index origin is 0.

Function selection

It's often required to run only one of a number of functions according to some test. I'm going to develop an operator to do this. I'll call it "or" with syntax as:

```
      res ← a b c(f or g or h)w
```

This will apply function (f), (g) or (h) to array (w) according to whichever is the first of (a), (b) or (c) if any that is true and is equivalent to select statement:

```
 :Select 1
 :Case a ⋄ res ← f w
 :Case b ⋄ res ← g w
 :Case c ⋄ res ← h w
 :Else   ⋄ res ← w
 :EndSelect
```

There are two calls to the operator in the above example; although variable it'll always be one less than the number of functions that must conform to the length of the left argument.

When the rightmost instance runs it has the expression's left argument as its left argument (⍺):

```
     ⍺  ↔ a b c
```

the entire function expression to its left as its left operand (⍺⍺):

```
     ⍺⍺ ↔ (f or g)
```

the function to its right as its right operand (⍵⍵):

```
     ⍵⍵ ↔ h
```

and the expression's right argument as its right argument (⍵):

```
     ⍵  ↔ w
```

If there are no ones in (⍺) we just return the argument (⍵):

```
     ~∨/⍺:⍵
```

If only the last item of (⍺) is a one we run the right function (⍵⍵):

```
     </⍺:⍵⍵ ⍵
```

Otherwise we run the left derived operand (⍺⍺) with a shortened left argument (⍺):

```
     (¯1↓⍺)⍺⍺ ⍵
```

Putting them together:

```
 or←{
     ~∨/⍺:⍵
     </⍺:⍵⍵ ⍵
     (¯1↓⍺)⍺⍺ ⍵
 }
```

Let's try it:

```
      0 0 0 0(f or g or h or m)23 ⍝ none at all
23
      0 0 0 1(f or g or h or m)23 ⍝ only the last of 4
(m 23 )
      0 0 1 0(f or g or h or m)23 ⍝ only the last of 3
(h 23 )
      0 1 0 0(f or g or h or m)23 ⍝ only the last of 2
(g 23 )
      1 0 0 0(f or g or h or m)23 ⍝ only the last of 1
( 1 f 23 )
```

There's a problem when we get to (f). So far we haven't mentioned it but this presupposes that all the supplied operands are monads and indeed they must be either that or dyads composed with a left argument as: (larg∘f00). The last line calls the left operand dyadically assuming it is another derived function containing another call to (or). But in this case it's merely (f) and we've called that as a dyad as well. We've made no provision actually to run the leftmost function as a monad.

Another line after [2] to deal with the case where the leftmost item of (⍺) is a one should finish it. If there are more than two items left then (⍺⍺) must be derived so we run our present last line. If there are only two items: we know (⍺⍺) & (⍵⍵) are supplied functions; at least one item of (⍺) is on because otherwise we should have skipped out at line [1]; if the second is on then so is the first because otherwise we should have skipped out at line [2]; so the first must be on; so run the left function (⍺⍺).

```
 or←{
     ~∨/⍺:⍵
     </⍺:⍵⍵ ⍵
     2=⍴⍺:⍺⍺ ⍵
     (¯1↓⍺)⍺⍺ ⍵
 }
      0 0 0 0(f or g or h or j)23
23
      0 0 0 1(f or g or h or j)23
(j 23 )
      0 0 1 0(f or g or h or j)23
(h 23 )
      0 1 0 0(f or g or h or j)23
(g 23 )
      1 0 0 0(f or g or h or j)23
(f 23 )
```

Sequential constraints

Suppose we want to check that (w), which can be any array, has particular type, rank, depth, shape and range and we can't devise a single test for this because we could always find an argument that would break one of the tests with a domain, rank or some other error. We want only to test subsequent properties if all previous are true.

```
 :If    okType  w
 :AndIf okRank  w
 :AndIf okDepth w
 :AndIf okShape w
    res←okRange w
 :Else  res←0
 :EndIf
```

I'll define an operator so that we can replace the seven lines of the If-statement with a single expression and call it "and".

```
      res←okType and okRank and okDepth and okShape and okRange w
```

and characterise our derived function as:

```
      (t0 and t1 and ... tm and tn)
```

At the first (rightmost) call to (and) we get:

```
      αα ↔ (t0 and t1 and ... tm)
      ωω ↔ tn
```

We can only run (tn) if all previous tests return true. So we run (αα) and see what we get. As we run (αα) we are actually calling (and) again with a progressively shorter derived left operand until it is merely (t0) which is the first actual test to run and the only one to be run as (αα). All tests are given the same argument (ω) and are expected to return one for true, zero for false. If (αα) returns true we run (ωω) which is (t1) then (t2) and so on coming back up through the calling levels. If all return true, eventually we get to (tn). At any point if any one of the tests returns false we merely return zero from there and from all the other levels. And here it is:

```
 and←{
     αα ω:ωω ω
     0
 }
```

At my first attempt to write this I automatically assumed that this couldn't possibly cover all cases and that more was needed so it grew to about six lines before I realised the first draft really was all we need.

```
      {(⍬≡0/⍵)∧(1=⍴⍴⍵)∧(0∧.≤⍵)}23 45 67     ⍝ conditional expressions conjoined with (∧)
1
      {(⍬≡0/⍵)∧(1=⍴⍴⍵)∧(0∧.≤⍵)}'try this'   ⍝ we can't test range if type is wrong ...
DOMAIN ERROR
      {(⍬≡0/⍵)∧(1=⍴⍴⍵)∧(0∧.≤⍵)}'try this'
                        ∧
      {⍬≡0/⍵}and{1=⍴⍴⍵}and{0∧.≤⍵}23 45 67   ⍝ conditional functions conjoined with (and)
1
      {⍬≡0/⍵}and{1=⍴⍴⍵}and{0∧.≤⍵}'try this' ⍝ ... so we don't
0
```

You can probably see that it would be more efficient to do this the other way around:

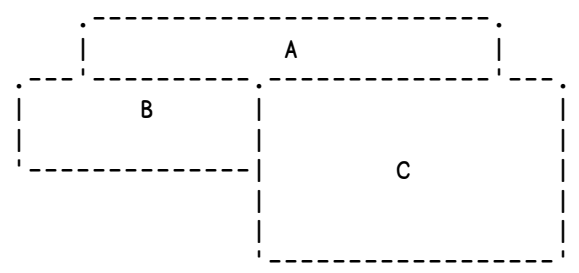
```
 if←{
     ωω ω:αα ω
     0
 }
```

specifying the tests in reverse order. If all are true we still traverse the entire tree but we skip out on the way down rather than on the way back if any prove false.

```
      {0∧.≤⍵}if{1=⍴⍴⍵}if{⍬≡⍵/⍵}12 34 56 78 90
1
      {0∧.≤⍵}if{1=⍴⍴⍵}if{⍬≡⍵/⍵}'try this'
0
```

Function arrays

When I'd been doing APL for a few months and had come to expect it to be capable of matching whatever I intuitively suspected it to be I came across a problem to calculate the minimum size of the rectangle surrounding three rectangles: A B C; where A is above B & C which are mutually adjascent:



What I wrote was:

```
      (⍴A)(+⌈)(⍴B)(⌈+)(⍴C)
```

and it seemed to me that it aught to work, functions applying separately to corresponding items of the shapes. Of course it was a syntax error and its future implementation as a syntax contruct has since been precluded, at least in J, by the invention of "forks" and other function "trains".

It was some time later that I recognised this as an example of vectors (or arrays) of functions that turned out to be a hot topic at a number of APL conferences during the eighties and hasn't gone away yet. It was longer still before I worked out how to do it with a defined operator.

To do it using an operator ("fv" for "function vector") we need syntax as:

```
      res ← a b c(f fv g fv h)x y x      ⍝ vector-vector
      res ← a b c(f fv g fv h)⊂x         ⍝ vector-scalar
      res ← (⊂a) (f fg g fv h)x y z      ⍝ scalar-vector
      res ←      (f fv g fv h)x y z      ⍝ monadic-vector
```

The number of calls to the operator is one less than the number of functions which must be the same as the length of the vector(s) (⍺) and/or (⍵) so that the functions are distributed to corresponding items of the argument(s). This implementation will not attempt to cover the case where a vector of functions is applied to one or between two scalar arguments. The length of the argument(s) will determine the behaviour of the operator as it did with (or) above.

Let's give it a try. At the first (rightmost) call above (f fv g fv h) we have:

```
      ⍺⍺ ↔ (f fv g)
      ⍵⍵ ↔ h
```

So we can run (⍵⍵) on/between the last of the argument(s) its being a supplied operand function but (⍺⍺) is still a derived function so we supply it with the remainder of the argument(s) missing its/their last item. And we'll distinguish between the monad and dyad:

```
 fv←{
     m←0                          ⍝ assume a dyad
     ⍺←m←1                        ⍝ override if a monad.
     m:(⍺⍺ ¯1↓⍵),⊂⍵⍵⊃⊢/⍵
     (⊃⍺⍺/¯1↓¨⍺ ⍵),⍵⍵/⊃¨⊢/¨⍺ ⍵
 }
```

It doesn't matter that we've assigned a value to (⍺) because we're not going to use it in the monadic case anyway.

```
      (f fv g fv h)1 2 3
(f 1 ) (g 2 )  (h 3 )
      1 2 3(f fv g fv h)4 5 6
( 1 f 4 ) ( 2 g 5 )  ( 3 h 6 )
```

These look OK but the spacing is suspect:

```
      ]disp 1 2 3(f fv g fv h)4 5 6
.→-----------------------------------.
|           .→--------. .→--------. |
| ( 1 f 4 ) |( 2 g 5 )| |( 3 h 6 )| |
| -   -   - '+--------' '+--------' |
'∈-----------------------------------'
```

The leftmost result is not enclosed. In fact we can see that by the time we are down to (α) and/or (ω) being 2-vectors the expressions should be symmetrical so we make our present code conditional on their not being 2-vectors and add the cases where they are:

```
 fv←{
     m←0
     α←m←1
     n←2≠⍴ω
     m∧n:(αα ¯1↓ω),⊂ωω⊃⊢/ω
     n>m:(⊃αα/¯1↓¨α ω),ωω/⊃¨⊢/¨α ω
     m>n:(αα 0⊃ω)(ωω 1⊃ω)
     m⍣=n:(αα/0⊃¨α ω),(ωω/1⊃¨α ω)
 }
      ]disp (f fv g fv h)4 5 6
.→---------------------------.
| .→-----. .→-----. .→-----. |
| |(f 4 )| |(g 5 )| |(h 6 )| |
| '+-----' '+-----' '+-----' |
'∈---------------------------'
      ]disp 1 2 3(f fv g fv h)4 5 6
.→-----------------------------------.
| .→--------. .→--------. .→--------. |
| |( 1 f 4 )| |( 2 g 5 )| |( 3 h 6 )| |
| '+--------' '+--------' '+--------' |
'∈-----------------------------------'
      ]disp 1 2 3(f fv g fv h)4
LENGTH ERROR
fv[4] m∧n:(αα ¯1↓ω),⊂ωω 0⊃⍵ω
                          ^
```

Oops! We haven't accounted for scalar extension. Use laminate and split to force both to be the length of whichever is the vector:

```
 fv←{
     m←0
     α←m←1
     (A W)←↓α,[-0.1]ω          ⍝ laminate and split
     n←2≠⍴W
     m∧n:(αα ¯1↓W),⊂ωω⊃⊢/W
     n:(⊃αα/¯1↓¨A W),ωω/⊃¨⊢/¨A W
     m:(αα 0⊃W)(ωω 1⊃W)
     (αα/0⊃¨A W),(ωω/1⊃¨A W)
 }
      1(f fv g fv h fv j)2 3 4 5
 ( 1 f 2 )  ( 1 g 3 )  ( 1 h 4 )  ( 1 j 5 )
      1 2 3 4(f fv g fv h fv j)5
 ( 1 f 5 )  ( 2 g 5 )  ( 3 h 5 )  ( 4 j 5 )
      + fv - fv × fv ÷ 1 2 3 4
1 ¯2 1 0.25
      1 2 3 4 + fv - fv × fv ÷ 5 6 7 8
6 ¯4 21 0.5
```

Sequential operations

It's often the case that we have a vector of known length and need to assign the items to different names just so that we can run a fixed sequence of functions between them. To run functions (f), (g), (h) & (j) sequentially between items of (vec):

```
      (a b c d e)←vec
      res←a f b g c h d j e
```

This is another claimant for the use of trains of functions as an extension to reduction or "insert" as it's called in J.

```
      (+-×÷) / 1 2 3 4 5 ↔ 1 + 2 - 3 × 4 ÷ 5
```

In fact J implements this but rather than as a train the functions are combined with the (tie) conjunction (operator) (`) to form a list of gerunds which can then be used directly as the argument (operand) to insert:

```
   +`-`*`% / 1 2 3 4 5
0.6
```

In like wise we have to insert our operator ("fs" for "function sequence") between the functions:

```
      (f fs g fs h fs j) a b c d e
```

but we will have a derived function that operates like the combination of gerund list with insert rather than a "function array" to be applied under reduction. The effect should be the same. What we can't do is resuse the functions cyclically as J does because we are dependent on the length of the argument to terminate the sequence.

Note that the number of calls to the operator is as usual one less than the number of functions and that that is one less than the length of the argument. Also that all the supplied functions are dyads and that the derived function is monadic. At the first call we have:

```
      αα ↔ (f fs g fs h)
      ωω ↔ j
```

We can hive off the last two items and run (ωω) between them, catenating its result on to the remainder and calling (αα) on that:

```
 fs←{
     αα(¯2↓ω),ωω/¯2↑ω
 }
      1 f 2 g 3 h 4 j 5                ⍝ this is what we want
( 1 f( 2 g( 3 h( 4 j 5 ))))
      (f fs g fs h fs j)1 2 3 4 5      ⍝ and this is what we have so far
(f 1  ( 2 g( 3 h( 4 j 5 ))) )
```

It's correct from (g) onwards so as previously we have to make a special case of the leftmost function. We can identify it because when (αα) is that function there must be only two functions left in the derivation so there must be three items in the argument vector:

```
 fs←{
     3=⍴ω:(0⊃ω)αα⊃ωω/1↓ω
     αα(¯2↓ω),ωω/¯2↑ω
 }
      (f fs g fs h fs j)1 2 3 4 5
( 1 f( 2 g( 3 h( 4 j 5 ))))
      ((f fs g fs h fs j)1 2 3 4 5) ≡ 1 f 2 g 3 h 4 j 5
1
      (+fs-fs×fs÷)1 2 3 4 5
0.6
```

Forks

As mentioned above the function train has been designated in J as a combination of forks with perhaps another construct called a "hook". A fork is defined thus:

```
       (f g h)⍵ ↔   (f ⍵)g(h ⍵)      ⍝ monad
      ⍺(f g h)⍵ ↔ (⍺ f ⍵)g(⍺ h ⍵)    ⍝ dyad
```

This too can be emulated with operators except for one proviso.

Any odd train longer than 3 constitutes a fork as follows with a simple fork on the right forming the right branch of a "larger" fork:

```
      (f g h j k) ↔ (f g(h j k))  ⍝ a 5-train
```

A derived function of five functions alternating with a dyadic operator parses thus:

```
      (f op g op h op j op k) ↔ ((((f op g)op h)op j)op k)
```

If we define our operator ("fk" for "fork") and we group our derived function in triples of functions each with two operators we see two forks as before but the "shorter" forms the left branch of the larger:

```
      (f fk g fk h fk j fk k) ↔ ((f fk g fk h)fk j fk k)
```

So we can never completely model a J odd train longer than three without non-redundant parentheses but we can easily model a simple fork:

```
       (f fk g fk h)⍵ ↔   (f ⍵)g(h ⍵)
      ⍺(f fk g fk h)⍵ ↔ (⍺ f ⍵)g(⍺ h ⍵)
```

At the first call we have:

```
      ⍺⍺ ↔ f fk g
      ⍵⍵ ↔ h
```

Taking the monad first we can run (h ⍵). But we also need to pass (⍵) to (⍺⍺) so we can apply (f) to it.

```
      ⍺⍺ ⍵(⍵⍵ ⍵)       ⍝ combined 2-vector argument
```

We need to distinguish the first from the second call. In this case the length of the arguments has no bearing on the calling sequence. We need another way to do it. The only way I've found so far is to introduce some "magic numbers" to flag the different internal calls. I'll use some Fibonacci related numbers. I'm sure there must be an entire literature on what to use:

```
 fk←{
     M←112358314594370 774156178538190 998752796516730 336954932572910
     h←⍵⍵               ⍝ first time  - label right operand
     f←⍺⍺ ⋄ g←⍵⍵        ⍝ second time - label left and right operands
     ⍺←⊢                ⍝ if a monad
     1≡⍺ 1:M ⍺⍺ ⍵(h ⍵)  ⍝ 1≡⍺ 1 identifies a first time monad
                        ⍝ - pass ⍵ and result of (h ⍵) to (f fk g) with flag M
     ⍺≡M:⊃{(f ⍺)g ⍵}/⍵ ⍝ M identifies a second time monad - ⍵ is from line above
                        ⍝ - separate it with {...}/ so the first part becomes ⍺
 }
      (f fk g fk h)23
((f 23 )g(h 23 ))
```

Well that's alright then. What about a dyad? We need a similar solution to the monadic except that we need also to pass (⍺) unchanged to the second call as it will also be the left arg to (f). And we need another flag "D" to identify it. If (⍺) isn't already (D) then make it so:

```
 fk←{
     (M D)←(112358314594370 774156178538190)(998752796516730 336954932572910)
     h←ωω
     f←αα ⋄ g←ωω
     α←⊢
     1≡α 1:M αα ω(h ω)
     α≡M:⊃{(f α)g ω}/ω
     α≠D:D αα(α ω)(α h ω)   ⍝ first dyad - pass D, (α & ω) and h-result
     α≡D:⊃{(⊃f/α)g ω}/ω     ⍝ D is a second dyad - f twixt orig α ω; g twixt f and h
 }
      23(f fk g fk h)45
(( 23 f 45 )g( 23 h 45 ))
      (+/ fk ÷ fk ⍴)1 2 3 4 5 6
3.5
      1 2 3 4 5 > fk ∨ fk = 5 4 3 2 1
0 0 1 1 1
```

Afterword

Until I wrote this, (fv) also used magic numbers, the same set as those above:

```
      112358314594370 774156178538190 998752796516730 336954932572910
```

I still have a probably futile hope that as I've just managed with (fv) I may find a way to implement (fk) without any.

Recapitulation

Co-operators:

```
      a b c(f or g or h)w          ⍝ select statement
           (f and g and h)w        ⍝ sequential testing
   [a[b c]](f fv g fv h)w[x y]     ⍝ function vector
           (f fs g fs h)w x y z    ⍝ sequential reduction
        [a](f fk g fk h)w          ⍝ fork
```

Phil Last. Hertfordshire. September 2011.