Two & Three Part Inventions

Abstract

Investigating the possibilities for generating a larger class of derived functions and operators through the agency of juxtaposition. For example, the difference between ...

```
      f g w       ⍝ which is f(g w)
```

and

```
      (f g)w      ⍝ which isn't
```

Sixteen two part & sixty-four three part syntactic permutations of arrays, functions & monadic & dyadic operators.

Earlier this year John Scholes[1] put forward three options for how to deal with "trains" of functions which currently generate SYNTAX ERROR in Dyalog APL.

1. Function arrays.

One of the things emerging aplers tend to try on the assumption that as nearly everything else seems to work this should as well ...

```
      1 2 3+4 5 6                  1 2 3+4                         1+2 3 4
5 7 9                        5 6 7                           3 4 5
      1 2 3(+×-)4 5 6              1 2 3(+×-)4                     1(+×-)2 3 4
5 10 ¯3                      5 8 ¯1                          3 3 ¯3
```

Partially implemented in Dyalog as named functions in arrays of namespaces or methods in arrays of instances.
Can be done without change to the interpreter through defined operators[2].

2. Forks and hooks.

SAX and J have these. Another case of intuitive expectation? ...

```
      1 2 3 4 5(>∨=)5 4 3 2 1  ⍝ (greater-than or equal)
0 0 1 1 1
      1 2 3 4 5(~<)5 4 3 2 1   ⍝ (not less-than)
0 0 1 1 1
```

Can be done without change to the interpreter through defined operators[3].

3. Currying.

Named after Haskell Curry, and implemented in a number of functional programming languages. We can specify less than the required number of arguments to a function to produce a resulting function whose valence is the number of arguments *not* already supplied. In APL terms this would normally entail the currying of a dyad with a left argument to produce a monad[4]. The most obvious example is ...

```
    increment←1+
    increment
1+
    increment 3
4
```

In his experimental FRE interpreter[5], John added two features not entirely related to closures. They were the ability to curry functions by the juxtaposition of an array and a function `(a f)` or of two functions `(f g)`.

The first is as above. An example of the second is ...

```
    index←⍳⍴
    index
⍳⍴
    index 'this'
0 1 2 3
```

Both these features can be viewed as the elision of the compose `"∘"` operator.
{
The third use of compose, which juxtaposes a function and an array `(f∘⍵)`

```
    decrement←-∘1
    decrement
-∘ 1
    decrement 4
3
```

cannot be elided as `(f ⍵)` would run as a monadic call to `(f)`.

```
    decrement←(- 1)
    decrement
¯1
    decrement 4
¯1 4
```

}

The derivations on the next page form a logical extrapolation of the FRE extensions deriving ...

Eight Arrays or immediate executions	0
Four Monadic Functions	1
Sixteen Ambivalent Functions	2
One Monadic Operator deriving monadic	3
Twenty Four Monadic Operators deriving ambivalent	4
Six Dyadic Operators deriving ambivalent	5
& Twenty One SYNTAX ERRORS	

The codes on the right appear in the derivation table below as "class".

The "*calling syntax*" of each derivation is how the *derived* entity would be called. The "*internal syntax*" is a dynamic function representation of how the interpreter would be expected to treat it after the *calling syntax* is satisfied. They are based on the table of "binding strengths" as defined by Jim Brown[6] and subsequently amended by Dyalog ...

Order of Binding Strengths in Dyalog

vector	binds adjacent data values to form a vector.
bracket	binds paired brackets to the entity on their left.
right-operand	binds a dyadic operator to a function or data on its right.
left-operand	binds an operator to a function or data on its left.
left-argument	binds a function to data on its left.
right-argument	binds a function to data on its right.

This will be seen to be insufficient for our purpose. This is partly because not all bindings are permitted to stand alone. The "array+function" binding described above is a case in point. This is merely *left-argument* binding but where, say, *left-operand* binding with a monadic operator produces an assignable function, *left-argument* binding does not. I am proposing that it should and, in fact, that all legal bindings be deemed to produce an assignable entity. The class of that entity will be determined by which of the four possible parameters, `(⍺)`, `(⍺⍺)`, `(⍵⍵)` & `(⍵)` it both requires and lacks.

Even this is insufficient as the juxtaposition of two functions or of anything to the right of a monadic operator is not in the bindings table. Hence an additional requirement is for a new binding, "*weak*" binding, which would bind any two entities that are allowed to be adjacent in a well formed expression but whose binding is not covered by any other. *Weak* binding would have to appear as the penultimate row in the table. This is for two reasons ...

1. *Right-argument* binding must remain the weakest of all as it is the last thing to occur prior to execution of the function.
2. *Weak* binding cannot occur before any other binding except *right-argument* otherwise the meaning of a large class of currently well formed expressions would change.

Two and Three Part Inventions

```
  id  ← components       internal syntax                      calling syntax class
= aa  ← arr0 arr1      ⋄                                    ⋄ aa             0
+ af  ← arr0 fnc1      ⋄ {arr0 fnc1 ω}                      ⋄ af ω           1
= am  ← arr0 mop1      ⋄ {α←{ω} ⋄ α(arr0 mop1)ω}            ⋄ α am ω         2
  ad  ← arr0 dop1      ⋄ {α←{ω} ⋄ α(arr0 dop1 αα)ω}         ⋄ α αα ad ω      4
= fa  ← fnc0 arr1      ⋄                                    ⋄ fa             0
+ ff  ← fnc0 fnc1      ⋄ {α←{ω} ⋄ α fnc0 fnc1 ω}            ⋄ α ff ω         2
= fm  ← fnc0 mop1      ⋄ {α←{ω} ⋄ α fnc0 mop1 ω}            ⋄ α fm ω         2
  fd  ← fnc0 dop1      ⋄ {α←{ω} ⋄ α(fnc0 dop1 αα)ω}         ⋄ α αα fd ω      4
  ma  ← mop0 arr1      ⋄ SYNTAX ERROR                       ⋄
  mf  ← mop0 fnc1      ⋄ {α←{ω} ⋄ α(αα mop0 fnc1)ω}         ⋄ α αα mf ω      4
  mm  ← mop0 mop1      ⋄ {α←{ω} ⋄ α(αα mop0 mop1)ω}         ⋄ α αα mm ω      4
  md  ← mop0 dop1      ⋄ {α←{ω} ⋄ α(αα mop0 dop1 ωω)ω}      ⋄ α αα md ωω ω   5
  da  ← dop0 arr1      ⋄ {α←{ω} ⋄ α(αα dop0 arr1)ω}         ⋄ α αα da ω      4
  df  ← dop0 fnc1      ⋄ {α←{ω} ⋄ α(αα dop0 fnc1)ω}         ⋄ α αα df ω      4
  dm  ← dop0 mop1      ⋄ SYNTAX ERROR                       ⋄
  dd  ← dop0 dop1      ⋄ SYNTAX ERROR                       ⋄
= aaa ← arr0 arr1 arr2 ⋄                                    ⋄ aaa            0
+ aaf ← arr0 arr1 fnc2 ⋄ {arr0 arr1 fnc2 ω}                 ⋄ aaf ω          1
= aam ← arr0 arr1 mop2 ⋄ {α←{ω} ⋄ α(arr0 arr1 mop2)ω}       ⋄ α aam ω        2
  aad ← arr0 arr1 dop2 ⋄ {α←{ω} ⋄ α(arr0 arr1 dop2 αα)ω}    ⋄ α αα ad ω      4
= afa ← arr0 fnc1 arr2 ⋄                                    ⋄ afa            0
+ aff ← arr0 fnc1 fnc2 ⋄ {arr0 fnc1 fnc2 ω}                 ⋄ aff ω          1
+ afm ← arr0 fnc1 mop2 ⋄ {arr0 fnc1 mop2 ω}                 ⋄ afm ω          1
  afd ← arr0 fnc1 dop2 ⋄ {arr0(fnc1 dop2 αα)ω}              ⋄ αα afd ω       3
= ama ← arr0 mop1 arr2 ⋄                                    ⋄ ama            0
  amf ← arr0 mop1 fnc2 ⋄ {α←{ω} ⋄ α(arr0 mop1 fnc2)ω}       ⋄ α amf ω        2
= amm ← arr0 mop1 mop2 ⋄ {α←{ω} ⋄ α(arr0 mop1 mop2)ω}       ⋄ α amm ω        2
  amd ← arr0 mop1 dop2 ⋄ {α←{ω} ⋄ α(arr0 mop1 dop2 αα)ω}    ⋄ α αα amd ω     4
= ada ← arr0 dop1 arr2 ⋄ {α←{ω} ⋄ α(arr0 dop1 arr2)ω}       ⋄ α ada ω        2
= adf ← arr0 dop1 fnc2 ⋄ {α←{ω} ⋄ α(arr0 dop1 fnc2)ω}       ⋄ α adf ω        2
  adm ← arr0 dop1 mop2 ⋄ SYNTAX ERROR                       ⋄
  add ← arr0 dop1 dop2 ⋄ SYNTAX ERROR                       ⋄
= faa ← fnc0 arr1 arr2 ⋄                                    ⋄ faa            0
+ faf ← fnc0 arr1 fnc2 ⋄ {α←{ω} ⋄ α fnc0 arr1 fnc2 ω}       ⋄ α faf ω        2
+ fam ← fnc0 arr1 mop2 ⋄ {α←{ω} ⋄ α fnc0 arr1 mop2 ω}       ⋄ α fam ω        2
  fad ← fnc0 arr1 dop2 ⋄ {α←{ω} ⋄ α(fnc0 arr1 dop2 αα)ω}    ⋄ α αα fad ω     4
= ffa ← fnc0 fnc1 arr2 ⋄                                    ⋄ ffa            0
+ fff ← fnc0 fnc1 fnc2 ⋄ {α←{ω} ⋄ α fnc0 fnc1 fnc2 ω}       ⋄ α fff ω        2
+ ffm ← fnc0 fnc1 mop2 ⋄ {α←{ω} ⋄ α fnc0 fnc1 mop2 ω}       ⋄ α ffm ω        2
  ffd ← fnc0 fnc1 dop2 ⋄ {α←{ω} ⋄ α(fnc0 fnc1 dop2 αα)ω}    ⋄ α αα ffd ω     4
= fma ← fnc0 mop1 arr2 ⋄                                    ⋄ fma            0
+ fmf ← fnc0 mop1 fnc2 ⋄ {α←{ω} ⋄ α fnc0 mop1 fnc2 ω}       ⋄ α fmf ω        2
= fmm ← fnc0 mop1 mop2 ⋄ {α←{ω} ⋄ α fnc0 mop1 mop2 ω}       ⋄ α fmm ω        2
  fmd ← fnc0 mop1 dop2 ⋄ {α←{ω} ⋄ α(fnc0 mop1 dop2 αα)ω}    ⋄ α αα fmd ω     4
= fda ← fnc0 dop1 arr2 ⋄ {α←{ω} ⋄ α(fnc0 dop1 arr2)ω}       ⋄ α fda ω        2
= fdf ← fnc0 dop1 fnc2 ⋄ {α←{ω} ⋄ α fnc0 dop1 fnc2 ω}       ⋄ α fdf ω        2
  fdm ← fnc0 dop1 mop2 ⋄ SYNTAX ERROR                       ⋄
  fdd ← fnc0 dop1 dop2 ⋄ SYNTAX ERROR                       ⋄
  maa ← mop0 arr1 arr2 ⋄ SYNTAX ERROR                       ⋄
  maf ← mop0 arr1 fnc2 ⋄ {α←{ω} ⋄ α(αα mop0 arr1 fnc2)ω}    ⋄ α αα maf ω     4
  mam ← mop0 arr1 mop2 ⋄ {α←{ω} ⋄ α(αα mop0 arr1 mop2)ω}    ⋄ α αα mam ω     4
  mad ← mop0 arr1 dop2 ⋄ {α←{ω} ⋄ α(αα mop0 arr1 dop2 ωω)ω} ⋄ α αα mad ωω ω  5
  mfa ← mop0 fnc1 arr2 ⋄ SYNTAX ERROR                       ⋄
  mff ← mop0 fnc1 fnc2 ⋄ {α←{ω} ⋄ α(αα mop0 fnc1 fnc2)ω}    ⋄ α αα mff ω     4
  mfm ← mop0 fnc1 mop2 ⋄ {α←{ω} ⋄ α(αα mop0 fnc1 mop2)ω}    ⋄ α αα mfm ω     4
  mfd ← mop0 fnc1 dop2 ⋄ {α←{ω} ⋄ α(αα mop0 fnc1 dop2 ωω)ω} ⋄ α αα mfd ωω ω  5
  mma ← mop0 mop1 arr2 ⋄ SYNTAX ERROR                       ⋄
  mmf ← mop0 mop1 fnc2 ⋄ {α←{ω} ⋄ α(αα mop0 mop1 fnc2)ω}    ⋄ α αα mmf ω     4
  mmm ← mop0 mop1 mop2 ⋄ {α←{ω} ⋄ α(αα mop0 mop1 mop2)ω}    ⋄ α αα mmm ω     4
  mmd ← mop0 mop1 dop2 ⋄ {α←{ω} ⋄ α(αα mop0 mop1 dop2 ωω(ω} ⋄ α αα mmd ωω ω  5
  mda ← mop0 dop1 arr2 ⋄ {α←{ω} ⋄ α(αα mop0 dop1 arr2)ω}    ⋄ α αα mda ω     4
  mdf ← mop0 dop1 fnc2 ⋄ {α←{ω} ⋄ α(αα mop0 dop1 fnc2)ω}    ⋄ α αα mdf ω     4
  mdm ← mop0 dop1 mop2 ⋄ SYNTAX ERROR                       ⋄
```

```
   mdd ← mop0 dop1 dop2 ⋄ SYNTAX ERROR                        ⋄
   daa ← dop0 arr1 arr2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 arr1 arr2)⍵}     ⋄ ⍺ ⍺⍺ daa ⍵     4
   daf ← dop0 arr1 fnc2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 arr1 fnc2)⍵}     ⋄ ⍺ ⍺⍺ daf ⍵     4
   dam ← dop0 arr1 mop2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 arr1 mop2)⍵}     ⋄ ⍺ ⍺⍺ dam ⍵     4
   dad ← dop0 arr1 dop2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 arr1 dop2 ⍵⍵)⍵} ⋄ ⍺ ⍺⍺ dad ⍵⍵ ⍵  5
   dfa ← dop0 fnc1 arr2 ⋄ SYNTAX ERROR                        ⋄
   dff ← dop0 fnc1 fnc2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 fnc1 fnc2)⍵}     ⋄ ⍺ ⍺⍺ dff ⍵     4
   dfm ← dop0 fnc1 mop2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 fnc1 mop2)⍵}     ⋄ ⍺ ⍺⍺ dfm ⍵     4
   dfd ← dop0 fnc1 dop2 ⋄ {⍺←{⍵} ⋄ ⍺(⍺⍺ dop0 fnc1 dop2 ⍵⍵)⍵} ⋄ ⍺ ⍺⍺ dfd ⍵⍵ ⍵  5
   dma ← dop0 mop1 arr2 ⋄ SYNTAX ERROR                        ⋄
   dmf ← dop0 mop1 fnc2 ⋄ SYNTAX ERROR                        ⋄
   dmm ← dop0 mop1 mop2 ⋄ SYNTAX ERROR                        ⋄
   dmd ← dop0 mop1 dop2 ⋄ SYNTAX ERROR                        ⋄
   dda ← dop0 dop1 arr2 ⋄ SYNTAX ERROR                        ⋄
   ddf ← dop0 dop1 fnc2 ⋄ SYNTAX ERROR                        ⋄
   ddm ← dop0 dop1 mop2 ⋄ SYNTAX ERROR                        ⋄
   ddd ← dop0 dop1 dop2 ⋄ SYNTAX ERROR                        ⋄
= already implemented in Dyalog production system.
+ already implemented in Dyalog FRE system.
```

Of the two part derivations ...
`(aa)` & `(fa)` execute immediately.
`(af)` uses *left-argument* binding to derive a monadic function.
`(am)` & `(fm)` are already allowed and use *left-operand* binding to derive an ambivalent function.
`(ad)` & `(fd)` use *left-operand* binding to derive a monadic operator.
`(da)` & `(df)` use *right-operand* binding to derive a monadic operator.
`(ff)`, `(mf)`, `(mm)` & `(md)` use *weak* binding to derive an entity of the higher degree of the pair in the order; fnc, mop & dop.
`(ma)` is a SYNTAX ERROR because it already has a right argument so aught to run but has no left operand so cannot.
`(dm)` & `(dd)` are SYNTAX ERRORS because a dyadic operator can never be followed by another operator which must thereby lack a left operand.

The three part derivations are illustrative of higher order compositions but all are strict applications of the full range of bindings on the two part derivations using the correct precedence. Given the equivalence of arrays and functions as operator operands the actual number of identifiably different cases is all but halved. Extending to four part derivations would add nothing as the internal & calling syntax is determined by the limit of at most four missing parameters; `(⍺)`, `(⍺⍺)`, `(⍵⍵)` & `(⍵)`.

Examples ...

```
 id ← e.g.  ⋄ use       → result

 af ← 1+    ⋄ af 3      → 4

 am

 ad

 ff ← ⍳⍴    ⋄ ff 'this' → 0 1 2 3

 fm ← +/    ⋄

 fd ← f00 ⍣ ⋄ 3 fd w    → f00 f00 f00 w

 fd ← ∧.    ⋄ x = fd y  → x∧.=y

 mf ← /⍳    ⋄ + mf 5    → 10

 mm ← ¨⍨    ⋄ ⍴ mm ⍳4   →  1  2 2  3 3 3

 md ← /⍣    ⋄ + md 2+y  → reduce last 2 dims

 da ← ⍣¯1   ⋄ 2 ⊥ da 9  → 1 0 0 1

 df ← .≠    ⋄ x ∨ df y  → x∨.≠y
```

Notes.

1. Impromptu presentation

Flipdb Moot, San Quírico d'Orcia, Italia, May 2007.

2. Dynamic operator implementing vector of functions.

```
  fv←{ ⍝ function vector
      m d←112358314594370 774156178538190
      ⍺←m
      e←2∊⍴⍵
      e<⍺≡m:(m ⍺⍺ ¯1↓⍵),⊂⍵⍵⊃⊖⍴⌽⍵
      e<⍺≡d:(d ⍺⍺ ¯1↓⍵),⍵⍵/⊃⊖⍴⌽⍵
      e∧⍺≡m:⊃⍺⍺{(⍺⍺ ⍺)(⍵⍵ ⍺)}⍵⍵/⍵
      e∧⍺≡d:⊃⍺⍺{(⍺⍺/⍺),⍵⍵/⍵}⍵⍵/⍵
      d ∇ ⍺{⍺ ⍵}¨⍵
⍝           f0∇∇f1 x y            ↔ (f0 x)(f1 y)
⍝     a b c f0∇∇f1∇∇f2 x y z      ↔ (a f0 x)(b f1 y)(c f2 z)
⍝      (⊂a) f0∇∇f1∇∇f2∇∇f3 w x y z ↔ (a f0 w)(a f1 x)(a f2 y)(a f3 z)
⍝ a b c d e f0∇∇f1∇∇f2∇∇f3∇∇f4 ⊂x  ↔ (a f0 x)(b f1 x)(c f2 x)(d f3 x)(e f4 x)
  }
```

3. Dynamic operators implementing fork & hook

```
  fk←{ ⍝ fork
      a m d g←112358314594370 774156178538190 998752796516730 336954932572910
      ⍺←a
      ⍺≡a:g ⍺⍺(m ⍺⍺ ⍵)(⍵⍵ ⍵)
      ⍺≡m:⍺⍺ ⍵
      ⍺≡d:⊃⍺⍺/⍵
      ⍺≡g:⊃⍵⍵/⍵
      g ⍺⍺(d ⍺⍺ ⍺ ⍵)(⍺ ⍵⍵ ⍵)
⍝   f0∇∇f1∇∇f2 ω         ↔ (f0 ⍵)f1(f2 ⍵)
⍝ ⍺ f0∇∇f1∇∇f2 ⍵         ↔ (⍺ f0 ⍵)f1(⍺ f2 ⍵)
⍝   f0∇∇f1∇∇f2∇∇f3∇∇f4 ⍵ ↔ ((f0 ⍵)f1(f2 ⍵))f3(f4 ⍵)
⍝ ⍺ f0∇∇f1∇∇f2∇∇f3∇∇f4 ⍵ ↔ ((⍺ f0 ⍵)f1(⍺ f2 ⍵))f3(⍺ f4 ⍵)
  }

  hk←{ ⍝ hook
      ⍺←{⍵}
      ⍺⍺ ⍺ ⍵⍵ ⍵
⍝   f0∇∇f1 ⍵ ↔ f0   f1 ⍵
⍝ ⍺ f0∇∇f1 ⍵ ↔ f0 ⍺ f1 ⍵
 }
```

4. Limited use of currying arguments in APL

John has pointed out that the naming of stranded arguments using namelists in traditional functions would lend itself to a more extensive currying of right arguments. Unfortunately this is precluded for class methods due to "overloading" so it's difficult to see how it could be utilised for "normal" functions.

5. Function Results Edition

Dyalog Conference, Helsingör, Danmark, October 2006.

6. Binding Strengths

Brown, JA,"The principles of APL2", IBM Technical Report, TR 03.247, March 1984.
The order as defined in this report and implemented in APL2 was ...

right-operand	binds a dyadic operator to a function or data on its right.
bracket	binds paired brackets to the entity on their left.
vector	binds adjacent data values to form a vector.
left-operand	binds an operator to a function or data on its left.
left-argument	binds a function to data on its left.
right-argument	binds a function to data on its right.