

New Foundations

by Graeme D Robertson (email: GraemeDR@nildram.co.uk)

Published in Vector Vol.20 No.1, July 2003

The foundations of APL should be extended to incorporate features of computer algebra systems, and extended beyond that to encompass **arrays of functions and operators**, in true APL style. This would push back the boundaries of VALUE ERRORS and SYNTAX ERRORS and open up a whole new world of mathematical analysis to APL programmers.

1. Functions returning functions

You might have noticed that in Dyalog APL it is possible to write a function that returns another function.

```
▽ FN←F00
[1] FN←{ω*2}
▽
```

This says that F00 returns a function called FN, which can then be applied to an array argument to give an array result. (A scalar is a rank zero array.)

```
      F00
▽FN
      (F00)3
9
```

This technique can be employed recursively to produce some very deep syntax. Here are three simple functions which combine to produce some unusual syntax, never imagined in [1].

```
▽ f←top
[1] f←middle
▽
      f←a middle w
[1] f←bottom
▽
      f←bottom w
[1] f←{ω*2}
▽
```

To call the function `top` you need to employ syntax like this:

```
      ((2(top)3)3)4
16
```

How do we distinguish between a function (with array arguments) which returns a *function*, and an operator (with array operands) which returns a *function*? For example, how can we be sure that the primitive function `Replicate (/)` is not interpreted as an operator with an array left operand? Well, simply inserting a pair of parentheses in a way which is legal for operators but not for functions is one sure way to decide.

```

5 5 5
  (3/)5
SYNTAX ERROR
  (3/)5
  ^

```

This proves that Replicate is interpreted as a function. Whereas

```

  (+/)5
5

```

proves that Reduce (/) is interpreted as an operator. So there is a clear difference.

Let us introduce some words in order to describe the syntax of `top` more precisely.

Definition: The **valency of a function** is **niladic**, **monadic** or **dyadic** depending upon whether it has zero, one (right) or two (one right and one left) **arguments**.

Axiom: A non-niladic function may be **ambivalent** - *either* monadic *or* dyadic depending contingently upon the presence or absence of a left argument.

Definition: The **potency of an operator** is **nihilistic**, **monistic** or **dualistic** depending upon whether it takes zero, one (left) or two (one left and one right) **operands**.

Axiom: An operator is **unipotent** - *exclusively* nihilistic, monistic *or* dualistic depending upon whether it is defined to take zero, one or two operands.

Now we can say that Replicate is a dyadic function and Reduce is a monistic operator which returns a monadic derived function. In the case of `top` above we can describe it as a niladic function which returns a dyadic function which returns a monadic function which returns another monadic function.

2. Operators returning operators

By their action on functions, operators usually *derive* new functions (ie return derived functions) which then usually take array arguments and *return* array results. Operators could conceivably take array operands and still return derived functions. The Composition operator (\circ) Form II of Dyalog APL takes an array left operand (and a function right operand). It might seem hard to imagine an operator which only takes array operands and still returns a useful function, however, the dyadic Circle function (\odot) could have been defined as a monistic operator, but it was not. It was defined as a dyadic function.

```

  (1∘).1
SYNTAX ERROR
  (1∘)0.1
  ^
  1∘.1
0.09983341665

```

If operators were allowed to derive arrays, or if functions were allowed to return operators, would APL grammar, as constructed in [1], disappear completely into mud and mist? Do we want a few simple tight rules, or should we encourage every conceivable expression? Well, we need *some* rules in any language before any informative statements can be constructed. Then, given the rules, not every expression will be possible. So what is possible?

First, what are the rules? The two basic rules of first generation APL parsing are:

Rule 1: The **right argument of a function** is the evaluated result of the entire expression to its **right** (taking into account any special ordering dictated by parentheses).

Rule 2: A function **left operand of an operator** is the function derived from the longest possible operator sequence to its **left** (taking into account any special ordering dictated by parentheses).

Rule 1 leads to the slogan “APL works from right to left”. However, operators mirror functions in that they capture their left argument with priority. These rules and definitions are sufficient to nail down a powerful grammar while still leaving a lot of flexibility for constructions using the different parts of speech - arrays, functions and operators.

Now consider the *dot syntax* in Dyalog APL. This is a particularly convenient notation because it mimics Visual Basic for Applications - almost exactly. The trouble with the notation, from an APL point of view, is that the Dot (.) of *dot syntax* has been given no proper place in APL grammar. Is it a function or an operator or what?

A dot is used for the decimal point. It may be possible to argue that in this construct - integer part on the left and fractional part on the right - the dot is being used as a dyadic function, but clearly there are counter-arguments to suggest that no such thing has ever been intended. We shall not pursue that hypothesis. Dot (.) is also used for Inner and Outer Products. For Inner Product, at least, it is treated as a dualistic operator which returns a dyadic function which returns an array result.

Considered as a dualistic operator, Dot (.) can be regarded as changing its definition depending upon the class of operands it is given. We are used to great flexibility and (perhaps too much) tolerance in the design of APL. Consider the way a token changes meaning depending upon context - the way a function changes with the presence of a left argument, or the way the Index Generator (ι) or Factorial function (!) change the underlying algorithm depending upon the rank or type of the argument, or consider the way that the symbol = changes its meaning depending upon JML, or whether a symbol (eg \) was legally overstruck with another symbol (eg o) ... Many of these tolerances stem from an era when symbols were precious. Fully Unicode systems should make symbols less rare, less precious, and just ‘there for the taking’. The APL dialect ‘J’ might have been too quick to jettison extended character sets because Windows applications are catching up now.

The basic use of *dot syntax* is to form a full hereditary name of a namespace, exactly as is done, for example, in Excel or Word VBA (and in Dyalog .NET):

```
Application.WorkBooks.Workbook.Styles.Style
```

or

```
Application.Documents.Document.Paragraphs.Paragraph
```

These are the full ancestral names of objects which, naturally, inherit stuff from their parents.

A syntactically, almost imperceptibly different, new use of dot syntax is to form a namespace-qualified variable name, exactly as, for example, in Outlook VBA:

```
Application.MailItem.Recipients.User.Type
```

If, for example, Type in namespace Application.MailItem.Recipients.User is set to 3 (Enum olBCC) then the message will be sent to the recipient as a blank carbon copy. Type is the name of a *property* of an object, in VBA parlance. In APL it is syntactically synonymous with the name of a *variable* in a namespace.

Another example of a property(variable) of an object(namespace) is Count in a collection object:

```
Application.WorkBooks.Workbook.Worksheets.Count
```

The value of `Count` is read like a variable but it can only be changed by adding or deleting worksheets in some other way. `Count` is a sort of read-only variable. Special types of variables have long been a part of APL, at least since shared variables were released in APL SV by IBM in 1973.

But the syntax of VBA is more arbitrary than that of Dyalog APL. The `Item` *property* of a collection object takes an index argument as if it were a *method* (and returns an item object from the collection). The `Item` property cannot be set and can only be used with an argument. Furthermore VBA has the concept of a **default** method or property which is associated with a given object. For example, the `Worksheets` collection assumes that you mean

```
Application.WorkBooks.Workbook.Worksheets.Item(3)
```

when you write

```
Application.WorkBooks.Workbook.Worksheets(3)
```

because the default property of the `Worksheets` collection is `Item`. APL is more rigorous. Implicit defaults have to be made explicit when translating from VBA to APL.

Methods associated with objects in VBA are essentially *functions* in namespaces for APL. Consider the method

```
Application.Documents.Document.PrintOut
```

This method looks like a niladic function and causes the document to be printed, whereas

```
Application.Documents.Document.PrintOut(Range:=wdPrintCurrentPage)
```

makes the method appear as a monadic function. The basic contradiction in APL of having a name act contingently as a monadic function or a niladic function (a contradiction nevertheless exhibited by \rightarrow) is avoided in APL from VBA by insisting that the niladic case of a method is given the dummy argument of zilde (\emptyset). Such dummy arguments appear in many places in Dyalog APL code because so many VBA methods may be called with zero or more right arguments (employing *constructors* with *overloading* to determine the precise method to use). If a pair of empty APL parentheses were defined to be zilde inside the interpreter then the APL code might appear more legible and more compatible with VBA.

Definition: $() \equiv \emptyset$

We take the Dot of *dot syntax* to be a dualistic operator that can assume various forms. Let us remind ourselves of the various flavours of Dot.

We recognise the Dot of Inner Product to be a dualistic operator which takes two dyadic function operands and returns a derived dyadic function. In the case where the functions are $+$ and \times and the arguments are matrices, the derived function $+. \times$ follows the formula for matrix multiplication.

If the Dot has a namespace to the left and a namespace to the right, then Dot returns the child namespace as an object. If the Dot has a namespace to the left and an array to the right, then it returns the array as it would be found in the left namespace. These two cases are syntactically equivalent because namespaces and arrays are both essentially class 2 objects. If the Dot has a namespace to the left and a function (monadic or dyadic) to the right, then Dot returns that function as it would be found in the left namespace.

The reason why Dot in this context works so much like an operator is because of Rule 2 above which ensures that the namespace to the left is constructed starting from the leftmost, and therefore most distant, ancestor. In other words, the parser ensures that namespace

$N1 . N2 . N3 . N4 . N5 . N6 . N7$

is interpreted according to explicit ordering

(((((N1.N2).N3).N4).N5).N6).N7

The main point of difference to notice at this stage between this Dot operator and all previous examples of operators in APL is the fact that the operator has two essentially class 2 operands and returns a derived class 2 object. In terms of the definition in [1] of parsing character, this flavour of Dot has character 0 0 0, whereas the Dot of Inner Product has character 2 2 2. (Parsing character was defined as the derived function valency catenated to the left operand valency (or zilde) catenated to the right operand valency (or zilde).) In the case of $N.f$ Dot has character 1 0 1 or 2 0 2. These different characters do not seem to conflict in the interpreter. The parser seems to cope.

However, **operators are also susceptible to dot syntax**. It is possible to have a namespace on the left and an operator on the right, in which case an operator is returned.

This totally confounds parsing character in [1]. It is, on reflection, not unknown in mathematics that an operator should act on an operator to produce another operator. For example, the second derivative operator is formed from the action of the first derivative operator on itself:

$$\frac{\partial}{\partial x}(\frac{\partial}{\partial x}) = \frac{\partial^2}{\partial x^2}$$

So now, apparently, dot syntax allows operators to take operators as operands and Dot derives new operators. For example, given namespace $\# . \alpha . b$ we can define and use a defined operator in relation to operator Dot.

```
# . \alpha . b . op ← { \alpha \alpha \omega }
*# . \alpha . b . op 1 3
2.718281828 7.389056099 20.08553692
```

In this example, the construction $\# . \alpha . b . op$ returns an operator. Dot can have a namespace to the left and an operator to the right, and return an operator.

The pictograms in [1] describing primitive syntax become arbitrarily tall at the function level because functions can return functions. They also become arbitrarily tall at the operator level because operators can return operators. Exhaustive encapsulation in pictograms of all possible primitive syntax becomes theoretically impossible.

3. Promotion of primitives + - × ÷ _ ! [] ▢ * ⊗ ⊛ ϕ , ρ +.× °.×

While on the one hand this new found freedom is frustrating to those who favour tight rules, on the other hand it reveals a panorama of possibilities to those who are happy to write unintelligible code.

Use the difficulty!

If something that was once clearly an operator ($f.g$) can now also be parsed consistently as something which looks like a classic function ($N1.N2$), then can the reverse be done? Can something which was once only known as a function, also be consistently used as an operator? Can we promote some primitive functions to useful operators?

One of the modern desires of A Programming Language is the possibility of arrays of functions. Such arrays are used all over the place in mathematics; tensors are arrays of functions of arbitrary rank whose form is invariant under a change of basis. (It is vital for all laws of physics that their form does not change with a change of coordinate system, therefore all laws are formulated as tensor equations.)

The first question is how to make a vector of functions. If we can promote Catenate (,) to an operator when it is applied to functions then we can use it to create **vectors of functions** in the same way as it is used to produce vectors of arrays.

$$VF \leftarrow \{\omega * 0\}, \{\omega * 1\}, \{\omega * 2\}, \{\omega * 3\}$$

We can promote Reshape (ρ) to operator level if the right argument is a function, or function array.

$$MF \leftarrow 3 \rho VF$$

We can promote indexing so that

$$(MF[1; 3]) 5$$

25

We can form the inner product of two such matrices by promoting $+ . \times$ to operator level in the case that at least one of the derived function arguments is a function array.

$$MF3 \leftarrow MF1 + . \times MF2$$

Note that Plus and Times in this situation are both promoted to operators. The promotion of Plus is so prevalent in mathematics that it is easy to overlook the fact that its definition has subtly changed. We move from adding numbers as in $3 + 4$, to adding functions as in $f(x) + g(x)$, to adding operators as in

$$\frac{\partial}{\partial x} + i \frac{\partial}{\partial y}$$

with hardly a thought as to the evolving definition of Plus in each case. This could be accommodated in APL if Plus can be promoted depending upon the class of the adjacent tokens. The following example would add operators together by promoting $+$ to an operator which acts on other operators to produce a new operator (left to right):

$$Op \leftarrow \{\alpha \alpha \omega\} + \{\alpha \alpha \alpha \omega\} + \{\alpha \alpha \alpha \alpha \omega\}$$

which is equivalent to

$$Op \leftarrow + / \{\alpha \alpha \omega\}, \{\alpha \alpha \alpha \omega\}, \{\alpha \alpha \alpha \alpha \omega\}$$

So we can now see that a number of primitives may be applied in many different ways to both functions and operators.

Vectors of operators are introduced in vector analysis. For example, the Gradient operator is defined as

$$\nabla \equiv \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$$

This monistic operator takes a scalar monadic function of a 3-vector (x, y, z) as its operand and returns a 3-vector of scalar gradient functions. The Divergence operator is written as a scalar product $\nabla \cdot V$ which is mathematical shorthand for $\nabla + . \times V$ in which Times is more like Composition. Divergence takes a 3-vector function and returns a scalar function:

$$V(x, y, z) = (f(x, y, z), g(x, y, z), h(x, y, z))$$

$$\begin{aligned} \nabla \cdot V(x, y, z) &\equiv \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \cdot V(x, y, z) \\ &\equiv \frac{\partial}{\partial x} f(x, y, z) + \frac{\partial}{\partial y} g(x, y, z) + \frac{\partial}{\partial z} h(x, y, z) \end{aligned}$$

The Curl operator takes a vector function operand and derives a vector function result. The Curl operator can be defined as the vector cross product of the vector Gradient operator with the vector function operand. The vector cross product is a very special type of anti-symmetric vector multiplication which only exists in 3 and 7 dimensions (because these are the remnant vector parts of the quaternion and octonion fields respectively). The inner product of the Gradient operator with itself is also a very significant operator.

Notice the use of a dot and its analogy with inner product. For Divergence, an inner product is applied between a vector operator ∇ and a vector function V . The functions $f(x, y, z)$, $g(x, y, z)$ and $h(x, y, z)$ are assumed to be scalar monadic functions which assign real numbers to each point of a region in the 3D space with coordinates x , y and z . So the function $V(x, y, z)$ is a monadic vector function of a vector which assigns a 3-vector of real numbers to each point in a 3D space (eg the velocity of a fluid at a point). Vector functions occur frequently in mathematical applications. A simple example is

$$V(x, y, z) = (xz, -xy^2, yz^2)$$

which might be programmed as

$$V \leftarrow \{ \omega[1] \times \omega[3] \}, \{ -\omega[1] \times \omega[2] \} * 2, \{ \omega[2] \times \omega[3] \} * 2$$

and used as a rank 1 1 function:

$$V \quad (1 \ 2 \ 3)$$

3 -4 18

An example of a *matrix operator* is the Dirac derivative which is a 4 by 4 matrix of partial derivatives with complex coefficients.

Ken Iverson has introduced in considerable detail [2] a typically brilliant attempt to implement vector analysis into APL. However, he does not have the luxury of native vector functions and operators nor the concept of promoted primitives to ease the transition from arithmetic to analysis. Nevertheless, it might be time to revisit this important work.

In order to implement arrays of functions and operators, it is probably necessary to considerably modify any existing APL interpreter. It will therefore be of interest to note that there are two GNU C++ libraries which would greatly assist with this non-trivial project to convert APL into a great algebraic (as well as a great arithmetic) language. The first is CLN - Class Library for Numbers [3]. This suite of programs would allow APL to incorporate easily many new types of numbers, including unlimited precision integers, rational numbers, unlimited precision floats and complex numbers, as well as many elementary functions that act on such numbers.

The second potentially very useful library is GiNaC - which stands for GiNaC Is Not A CAS [4,5]; where CAS means Computer Algebra System. (Not Germanium Sodium Carbide? - the recursive name itself, GiNaC, is worthy of an APL application!) GiNaC is a set of building blocks, based on CLN, for performing algebraic operations on functions and variables, such as forming derivatives or simplifying or expanding algebraic expressions. GiNaC was developed with the intention of replacing a Maple based system with a flexible modular system designed round a set of C++ programs with

algebraic capabilities that could be incorporated easily into larger C++ programs. This ‘larger program’ could and should be an APL interpreter.

References

- [1] G.D.Robertson, *APL Linguistics*, Vector **2.2** (1985) 118-126
- [2] K.E.Iverson, *The Derivative Operator*, Proceedings of **APL79** (1979) 347-354
- [3] <http://www.ginac.de/CLN>
- [4] <http://www.ginac.de>
- [5] C.Bauer et al, *Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language*, J.Symbolic Computation **33** (2002) 1-12