Operators

KENNETH E. IVERSON
IBM Corporation

Although operators, which apply to functions to produce functions, prove very useful in mathematics, they are absent from most programming languages. This paper illustrates their simplicity and power in terms of the operators of APL, and examines related constructs in other programming languages.

Key Words and Phrases: APL, operator, derived function, operator syntax, function composition.
CR Categories: 3.15, 4.20

1. INTRODUCTION

The notion of an *operator,* defined as an object which applies to a function or functions to produce a related function, is widely used in mathematics. Familiar examples include the derivative operator (and related operators such as the gradient, divergence, and curl of vector calculus), the tensor product, composition, and various transforms such as Laplace and Fourier. Because a small number of functions and operators can be combined to produce a much larger set of *derived* functions, the use of operators can lead to great economy in the number of distinct functions required, and also to the recognition of structure in the set of functions employed.

If we note that the verbs (or at least the *imperative* verbs) of natural language play a role similar to the functions of mathematics, then we can also identify the use of operators in natural language -- *adverbs* (and other devices such as inflection) modify *verbs* to produce, in effect, new related verbs.

Despite the utility of operators in mathematics and in natural languages, programming languages have largely avoided their use. The notable excep-

Author's address: T.J. Watson Research Center, P.O. Box 218, Yorktown Hts., NY 10598.

tion is APL, and we will here examine the notion of operators in the context of APL, illustrating their utility in terms of operators now implemented, contrasting the operator syntax and function syntax, and exploring new extensions of the use of operators. Finally, we will comment on the relationship of operators to certain constructs found in other programming languages.

Although the term *operator* as used here conflicts with its use in elementary mathematics and in programming languages as a synonym for *function*, we will use it in accordance with its definition in the International Dictionary of Applied Mathematics [1]. The term functional sometimes suggested as an alternative is inappropriate because, although its domain is functions, its range is the real numbers.

All APL notation used will be defined briefly in context; readers may find more detail on the language in [2], and discussions of its design and development in [3–5]. Except for the extra-lingual symbol ←→ used to denote equivalence of the expressions surrounding it, the expressions used may be executed directly on an APL computer.

Three generally useful functions will be defined immediately: `*` denotes the *power* function (as in `A*B` for `A` to the power `B`), `ρ` denotes the *shape* function, yielding the shape of its argument (a single element `N` when applied to a vector of `N` elements, a two-element vector `M,N` when applied to an `M` by `N` matrix, etc.), and `ι` denotes the *integer* function, yielding a vector of the first `N` non-negative integers when applied to a non-negative argument `N` (as in `ι4` for `0 1 2 3`). For example, these functions may be used to provide an expression for the terms of a polynomial with coefficents `C` and argument `X` as follows: `C×X*ιρC`. Thus if `C←3 1 4 2` and `X←2`, then:

```
      ρC ←→ 4
     ιρC ←→ 0 1 2 3
   X*ιρC ←→ 1 2 4 8
 C×X*ιρC ←→ 3 2 16 16
```

2. PRESENT OPERATORS

We will first illustrate the utility of operators by the three major APL operators, *reduction* (denoted by `/`), *scan* (denoted by `\`), and *inner product* (denoted by `.`). The first two are *monadic,* applying to a single argument, and the inner product is *dyadic,* applying to two arguments. The two arguments of the inner product are normally functions, but a null left argument (denoted by `∘`) can also be used to produce a degenerate case called the *outer product.*

2.1 Reduction

The reduction operator applied to the function *plus* (as in `+/`) produces the *summation* function commonly denoted by the uppercase sigma in mathematics. For example:

```
    +/ 1 2 3 4 ←→ 10
```

More generally, f/ produces a function equivalent to placing the function f between the elements of its argument. Thus, `×/ 1 2 3 4 ←→ 1×2×3×4 ←→ 24`. Used in conjunction with common dyadic functions such as *plus, times, and, or, exclusive-or, maximum,* and *minimum* (denoted by `+,×,∧,∨,≠,⌈`, and `⌊`) the reduction operator therefore produces a number of useful functions commonly denoted in mathematics by miscellaneous symbols such as sigma, pi, and max. For example:

```
X ← 3 1 4 2                  B ← 1 0 1 0

⌈/X ←→ 4                     ∧/B ←→ 0

⌊/X ←→ 1                     ∨/B ←→ 1

-/X ←→ 4                     ≠/B ←→ 0
```

More interesting examples of reduction can be framed in conjunction with other operators yet to be defined. For the moment, we will cite only two further examples relevant to polynomials. Since `C×X*⍳⍴C` yields the terms of a polynomial with coefficients *C*, the reduction `+/C×X*⍳⍴C` yields the polynomial itself. Times reduction can be used similarly to express a polynomial in terms of its vector of roots *R*. Thus: `×/X-R`.

The derived function f/ applies to matrices or higher-dimensional arrays by applying to each of the vectors along the last axis of the array. Thus if *M* is a matrix, `+/M` produces row sums, `⌈/M` produces row maxima, etc. The related operator `⌿` behaves similarly, producing derived functions that apply along the *leading* axis. Thus `+⌿M` and `⌈⌿M` produce *column* sums and maxima.

2.2 Outer Product

The outer product of tensor analysis produces the ordinary product (×) of each element of its left argument with each element of its right. For vectors *A* and *B* their tensor outer product is therefore a matrix which is, in effect, a multiplication table for *A* and *B*. The tensor product, denoted by `∘.×`, is a special case of the APL outer product. Thus:

```
      I∘.×I←1 2 3 4
 1  2  3  4
 2  4  6  8
 3  6  9 12
 4  8 12 16

      1 2 1∘.×1 3 3 1          0 1 2∘.+0 1 2 3
1 3 3 1                  0 1 2 3
2 6 6 2                  1 2 3 4
1 3 3 1                  2 3 4 5
```

The last two examples above can be used to illustrate the use of the outer product in determining the product of two polynomials. If the arguments C and D are the coefficients of two polynomials, then the table `C∘.×D` contains all elements to be summed to produce the coefficients of the product. The corresponding exponent for each product element is given by the table `0 1 2 ∘.+ 0 1 2 3`, which shows that sums should be taken down the counter-diagonals, yielding `1 5 10 10 5 1`.

Functions other than × can be used in the outer product, as in `∘.+` for an addition table, `∘.⌈` for a maximum table, and `∘.*` for a power table. An obvious use for such tables is to exhibit the properties of unfamiliar functions. For example, `|` denotes the *residue* function, `!` denotes the *binomial coefficients,* and `<` denotes the *less-than* function (which yields 1 if the relation holds and 0 if it does not). The corresponding tables appear as follows:

```
      I←0 1 2 3 4

      I∘.|I              I∘.!I              I∘.<I
 0 1 2 3 4          1 1 1 1 1          0 1 1 1 1
 0 0 0 0 0          0 1 2 3 4          0 0 1 1 1
 0 1 0 1 0          0 0 1 3 6          0 0 0 1 1
 0 1 2 0 1          0 0 0 1 4          0 0 0 0 1
 0 1 2 3 0          0 0 0 0 1          0 0 0 0 0
```

Three less obvious uses of outer products will now be illustrated: incidence matrices for plotting functions, balancing of parentheses in an expression, and determination of a sieve for prime numbers.

If

```
      X←1 2 3 4 5 6 7
      V←(X-3)×(X-5)
      R←8 7 6 5 4 3 2 1 0 ¯1
```

then V equals `8 3 0 ¯1 0 3 8` and is the result of applying a quadratic function to X, and R comprises the range of values in V. Hence:

```
      R∘.=V                          ' *'[R∘.=V]
 1 0 0 0 0 0 1                    *     *
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 0 1 0 0 0 1 0                     *   *
 0 0 0 0 0 0 0
 0 0 0 0 0 0 0
 0 0 1 0 1 0 0                      * *
 0 0 0 1 0 0 0                       *
```

The *equals* table on the left gives a plot of the quadratic in which 1's represent points on its graph; use of this table to index the two-element character vector consisting of a space and an asterisk produces the more pleasing plot on the right. Use of `R∘.≤V` produces a bar chart instead of a graph, and use of the monadic absolute value function `|` in an expression such as `TOL≤|R∘.-V` produces a table which identifies all points in *V* which lie within a specified tolerance of points in *R*.

If `E←'3×((X-Y)×(P-Q)'`, then *E* is a 14-element character vector which may represent an expression to be parsed. To determine the balancing of parentheses we make the table `'()'∘.=E` whose first row identifies left parentheses and whose second identifies right. The expression `-⌿'()'∘.=E` then yields a vector with 1 for each left parenthesis and ¯1 for each right, and `+/-⌿'()'∘.=E` therefore yields the overall balance of parentheses. Thus:

```
      E
3×((X-Y)×(P-Q)

      '()'∘.=E
0 0 1 1 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 1

      -⌿'()'∘.=E
0 0 1 1 0 0 0 ¯1 0 1 0 0 0 ¯1

      +/-⌿'()'∘.=E
1
```

If `L←2+⍳11`, then `L∘.×L` is a multiplication table for the list of integers from 2 to 12, and `L∘.≠L∘.×L` is a three-dimensional array whose element in plane *I*, row *J*, and column *K* is 1 if `L[I]` differs from the product of `L[J]` and `L[K]`. Consequently, `C←∧/∧/L∘.≠L∘.×L` is a vector whose *I*th element is 1 if `L[I]` differs from all of the elements in the multiplication table, and the 1's in *C* therefore indicate which elements of *L* are prime numbers. Thus:

```
      L
2 3 4 5 6 7 8 9 10 11 12

      ∧/∧/L∘.≠L∘.×L
1 1 0 1 0 1 0 0 0 1 0
```

2.3 Inner Product

The matrix product of two matrices *M* and *N* is defined as a matrix whose *I*,*J*th element is obtained by summing the result of taking an element-by-element product between row *I* of *M* and column of *J* of *N*, expressible in APL as `+/M[I;]×N[;J]`. The *inner product* operator, denoted by a period, applies to two functions f and g (as in `+.×`) to produce a function analogous to matrix product, but with `f/M[I;]gN[;J]` replacing `+/M[I;]×N[;J]`. Consequently, the inner product operator is a generalization of matrix product, producing it in the special case `+.×`.

Applied to logical and relational functions and addition, the inner product produces a variety of useful comparison functions. For example, if *N* is a character matrix whose rows are names, and *V* is a vector of the same length as the rows, then *N*∧.=*V* produces a boolean vector which identifies the rows of *N* which agree with *V*. Similarly, the expression *N*+.≠*V* gives counts of the disagreements between *V* and each row of *N*, and *TOL*>*N*+.≠*V* identifies rows whose disagreements fall below a specified tolerance.

Applied to boolean right arguments, the derived function +.× can be used to produce sums over all subsets of the left argument specified by the columns of the boolean right argument. For example, if *S* is a list of yearly sales to a set of companies, the columns of the boolean matrix *S*∘.>*L* identify those exceeding certain limits specified by the list *L*, and *S*+.×*S*∘.>*L* produces a summary of sales falling in the various classes. Thus:

```
      S ← 125 1600 17 45 2000 65

      L ← 1000 300 100 30 0

      S+.×S∘.>L
3600 3600 3725 3835 3852
```

Similarly, the derived function ×.* with a boolean right argument produces *products* over subsets of the left argument. For example, if *B* is the boolean matrix of the first 2*N *N*-digit binary numbers (given by the expression *B*←(*N*⍴2)⊤⍳2**N*), then for *N*←4 we have:

```
      B
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

      R←2 3 5 7

      R×.*B
1 7 5 35 3 21 15 105 2 14 10 70 6 42 30 210
```

Combined use of sums and products over subsets can be seen in the following expression for the coefficients of a polynomial in terms of its roots:

```
      C←((-R)× *B)+.×(+/0=B)∘.=⍳1+⍴R.
```

Since *B* represents all subsets of a set of order ⍴*R*, the expression (-*R*)×.**B* gives all the products involved in Newton's symmetric functions, and it remains to sum subsets of *them* determined by comparing the number excluded from each subset (given by +/0=*B*) with the indices ⍳1+⍴*R* of the resulting vector of coefficients. Validity of the result can be checked for various values of X as follows:

```
      X←3.5

      C                                  R
210 ¯247 101 ¯17 1                2 3 5 7

      +/C×X*⍳⍴C                          ×/X-R
3.9375                            3.9375
```

The availability of matrices produced by certain outer products also leads to rather unexpected uses of the ordinary matrix product `+.×`. For example, since the columns of the matrix `M←(⍳N)∘.!⍳N` are, as shown in an earlier example, the successive vectors of binomial coefficients, and since the matrix product `M+.×C` is equivalent to a sum of the column vectors of M weighted by elements of C, it is easily shown that `D←(I∘.!I←⍳⍴C)+.×C` is the expansion (for `X+1`) of the polynomial with coefficients C. For example:

```
      C←3 1 4 2

      I∘.!I←⍳⍴C                      D←(I∘.!I←⍳⍴C)+.×C
 1 1 1 1                             D
 0 1 2 3                     10 15 10 2
 0 0 1 3
 0 0 0 1

      X←3

      +/C×(X+1)*⍳⍴C                  +/D×X*⍳⍴D
199                          199
```

2.4 Scan

The scan operator, denoted by \, is closely related to the reduction operator, since it produces a vector of results defined as the reduction over all *prefixes* (i.e., initial segments) of its arguments. For example:

```
      I←3 1 4 2

      +\I                 ×\I                 ⌊\I
3 4 8 10           3 3 12 24           3 1 1 1
```

Uses of the scan are manifold. Just as `+/F G` is an approximation to the integral of the function F over the grid points G, so is `+\F G` an approximation to the integral function over the same set of points G. Used in conjunction with `⍳N` it produces the factorials `(×\1+⍳N)`, the triangular numbers `(+\⍳N)`, and, more generally, the successive figurate numbers `(+\+\⍳N` and `+\+\+\⍳N`, etc.). The plus scan can also be substituted for the plus reduction in the parenthesis-balancing example given earlier to yield the depth in parentheses in every part of the expression *E*. Thus:

```
+\-⌿'()'∘.=E←→0 0 1 2 2 2 2 1 1 2 2 2 2 1
```

Used in conjunction with a boolean argument vector *B* the scan can produce various useful vectors including: a solid segment of 1's beginning at the first 1 in the argument (∨\B), all zeros beginning with the first (∧\B), alternation of 1's and 0's produced by successive 1's (≠\B), and a lone 1 in the position of the first (<\B). Corresponding results can be obtained for the rows or columns of a boolean matrix argument. For example, if M←(⍳N)∘.=⍳N, then *M* is an identity matrix, U←∨\M is an upper triangle, and <\U is again an identity matrix.

3. SYNTAX OF OPERATORS

Because a function produces an explicit result which can serve as the argument of another function, it is possible to write sequences of such functions such as F G H X or (X×Y)+(P×Q). An appropriately defined syntax for functions can make such sequences unambiguous, and make it convenient to form complex sentences to serve a variety of purposes.

Since the result of an operator is a derived function which could serve as argument to another operator, a similar facility for writing unambiguous sequences of operators could be provided, as for example, in the expression +.×/ to yield reduction by inner product over the set of matrices formed by the planes of a three-dimensional argument.

Because present definitions and implementations of APL do not permit such sequences of operators, it has not been necessary to formulate a general syntax for operators. However, any extension of the use of operators should properly begin with such a formulation.

3.1 Position of Operators

Present operators in APL are, like functions, either *monadic* (taking one argument), or *dyadic* (taking two). Dyadic operators, like dyadic functions, appear *between* their arguments, as in A+.×B compared to A+B. Monadic operators, unlike monadic functions, appear to the *right* of their arguments rather than to the *left,* as in +/ and ×\ as contrasted with -X and |X. The position of operators is therefore a mirror image of the position of functions.

3.2 Parentheses and Order of Execution

Parentheses control the sequence of execution of functions in an APL expression in the usual manner. In the absence of parentheses, the order of execution is determined as follows: the right-hand argument of any function is the value of the entire expression to its right. The great convenience of this rule seems to arise from the fact that the application of a further transformation to any result *X* (either monadic, as in F X, or dyadic, as in P⌈X) is specified by simply prefixing the expression which produced *X* with the desired transformation.

In order to obtain the same convenience for sequences of operators, it appears necessary to impose an order of execution which, because the positions of operators and functions are mirror images, is a mirror image of the rule for functions. Thus we impose the following rule which applies in the absence of parentheses: the left argument of any operator is the result of the longest operator sequence to the *left* of the operator. For example, in the expression *M*×+.×/*TDA*, the longest operator sequence to the left of the reduction operator is +.×, and reduction is therefore applied to the inner product +.× to produce an inner product between subarrays of the argument *TDA*, between matrix planes if *TDA* is a three-dimensional array.

3.3 Precedence

There are no rules of precedence to distinguish among APL functions -- all behave alike as concerns order of execution. Likewise there is no precedence among APL operators. However, operators do take precedence over functions. Thus, in the expression *A*+.×*B* the inner product operator is executed first to produce the derived function +.× which is then applied to the arguments *A* and *B*. Consequently, the parentheses in *A*(+.×)*B* are redundant.

3.4 Valence

The *valence* of an APL object is defined as the number of arguments to which it applies, a dyadic function or operator having valence 2, a monadic function or operator having valence 1, and an array having valence 0. The symbol for an APL primitive function is ambivalent, the valence of the function it represents being determined by context. Thus the symbol - represents the dyadic function *subtraction* in the expression *A*-*B*, but the monadic function ***negation*** in the expressions -*B* and *A*×-*B*; the symbol ÷ represents ***division*** in *A*÷*B*, and ***reciprocal*** in ÷*B*, etc. This property of primitive symbols applies equally to the names of user-defined functions, although not all implementations permit ambivalent user-defined functions.

Ambivalent operator symbols are not used because, due to the precedence given to operators, every monadic use of an ambivalent operator would require enclosing parentheses. Moreover, the use of the jot (∘) in the essentially monadic outer product shows how monadic cases of dyadic operators can be defined when desired.

3.5 Domains

As used thus far, an operator applies only to functions, and its domain does not include objects of valence 0. However, it is clear that an *axis* operator, which specifies the axis or axes of an argument to which a function is to be applied, should apply to two arguments, one being a function and the other a number. Likewise, a *K*th derivative operator should apply to a function and a number. There seems, in fact, to be no good reason to exclude variables from the domain of operators.

The reduction operator (denoted by /) applies, as we have seen, to a dyadic function such as *plus* to produce a monadic derived function such as *summation.* It also applies to a boolean vector *U* (as in *U*/) to produce a *compression* function which applies to its right argument to select those elements corresponding to the 1's in *U*. For example, if `U← 1 0 1 0 1` and `V← 1 2 3 4 5`, then `U/ V` yields `1 3 5`. Similar remarks apply to the scan operator \; when applied to a boolean argument it produces the *expansion* function illustrated by the following example: if `U←1 0 1 0 1` and `V←1 2 3`, then `U\V` yields `1 0 2 0 3`. Further implications of such use of variables as well as functions as arguments of operators are discussed in Iverson [5, pp. 129–130].

3.6 Ambivalent Derived Functions

Each of the derived functions discussed thus far has been of fixed valence, being either monadic (as in `+/` or `U/` or `+\`) or dyadic (as in `+.×` or `∘.×`). However, there is considerable advantage to be gained from assuming that derived functions are, like primitive and user-defined functions, ambivalent, their valence in application being determined by context.

Thus, meanings could be attached to the (dyadic) expression `A G/ B` (for both a function *G* and a boolean vector *G*) as well as to the (monadic) expression `G/ B`. For boolean *G*, the derived functions `G\` and `G/` may be defined as the functions *mesh* and *mask,* discussed in [6, pp. 19–21] and illustrated below:

```
A←1 2                    B←3 4 5               G←1 0 1 0 1

A G\ B ←→ 3 1 4 2 5

A←1 2 3 4 5              B←10 9 8 7 6          G←1 0 1 0 1

A G/ B ←→ 10 2 8 4 6
```

For the case of a dyadic function *G*, the expression `A G/ B` produces reductions by *G* over a moving window of width *A*. For example, if `B←7 5 3 2 1`, then:

```
2 +/ B ←→ 12 8 5 3          3 +/ B ←→ 15 10 6
2 -/ B ←→ 2 2 1 1           2 >/ B ←→ 1 1 1 1
```

The last two examples suggest the manner in which a left argument of 2 can be used to obtain pairwise differences, pairwise comparisons, pairwise ratios, etc. Formal definitions of `A G/ B` and `A G\ B` are given in [7] for both positive and negative values of the left argument *A*.

A useful definition for a monadic case of the inner product f.g has also been proposed [8]. Since the determinant is a monadic function defined as the function f (alternating sum) applied over the results of applying a function g (product) over certain subsets of the elements of the argument, and since `-/V` yields the alternating sum of the elements of *V*, the determinant of *M* can be represented as `-.×M`. The *permanent* would therefore be represented as `+.×`, and other useful cases (such as `∨.∧`) of this generalized determinant can be identified.

4. NEW OPERATORS

In order to indicate the directions in which the development of operators in APL may lead, we will now consider three groups of new operators, the first two of which are adapted from mathematics. The first concerns composition and application (in the lambda-calculus sense), and the second concerns operators corresponding to dyadic functions in the sense commonly represented by f+g and f×g, etc. The third group concerns axis operators. Further operators such as the *derivative,* the *dual,* and the *domain* are discussed in [7] and [9].

Expressions are more readable if the operator symbols are easily distinguishable from the function symbols. This distinction is easily made in present APL because of the small number of operator symbols used (the brackets and the symbols . / \ ⌿ ⍀). This ease of distinction can be maintained by adopting a small number of *classes* of symbols for operators, those incorporating an overbar (¯) or a dieresis (¨), making exceptions only for certain mnemonically-suited cases, such as the delta (Δ) for the derivative operator.

4.1 Composition and Application

Composition, denoted by the dieresis alone, is defined as in mathematics so that *F*¨*G* produces a monadic function based upon the monadic functions represented by *F* and *G*, but is extended to also produce a dyadic case based upon the dyadic function represented by *F*:

```
  F¨G B ←→ F(G B)
A F¨G B ←→ (G A) F (G B)
```

The definition of composition given above must be understood as applying to an argument *B* of the lowest *rank* (i.e., dimensionality) appropriate to the function *G*. For example, if *G* is matrix inverse (denoted by ⌹ and applying to each of the matrices along the last two axes of its argument), if *F* is the *transpose* (denoted by ⍉ and defined to reverse the order of the axes of its argument), and *B* is a rank 3 array of shape 5 4 4, then ⍉¨⌹ *B* differs from ⍉⌹*B*, since the former transposes the inverse of each of the five 4-by-4 matrices of *B* to form a result of shape 5 4 4, whereas the latter inverts each of the same matrices to form a result of shape 5 4 4 which is *then* transposed to produce a result of shape 4 4 5. In other words, it is the resulting derived function *F*¨*G* which is applied to each subarray of appropriate rank, rather than the function *F* being applied to the result of applying *G* to the entire array. Similar remarks apply to other derived functions.

For any *dyadic* function *F* there exist two (often very useful) related *monadic* functions obtained by assigning a fixed value to the left argument and leaving the *right* argument as argument (to be called a *right case of F*), and by assigning a fixed value to the right argument (a *left* case). If *V* is a variable and *F* is a dyadic function, then the compositions *V*¨*F* and *F*¨*V* may be given the following useful definitions:

```
V¨F X ←→ V F X                    F¨V X ←→ X F V
```

For example, the monadic exponential function 2*X is a right case of the dyadic power function *, and the square function X*2 is a left case. The compositions with a variable provide for convenient production of such monadic cases, as in 2¨* and *¨2.

If E is a character vector (such as 'X+A÷Y') which represents an expression, then E can, as in the lambda calculus, also represent a *function* if the names of the arguments are identified. If these are identified by a second character vector C (such as C←'X Y'), then an operator applied to E and C can produce a function. The composition operator ¨ will be used in this way, and will then be referred to as *application.* Thus if A←6, then 5 E¨C 2←→8, and if A←6 and X←10, then E¨'Y' 3←→12.

We will also define an abbreviated form of application, using a jot as right argument and assuming that the left and right arguments in the expression involved are represented respectively by the fixed symbols α and ω. For example:

```
        3 'α+÷ω'¨∘ 5 ←→ 3+÷5

'α+÷ω'¨∘/3 2 4 5 ←→ 3+÷2+÷4+÷5
```

The last example shows that reduction of a vector V by the dyadic function 'α+÷ω'¨∘ yields the continued fraction represented by V.

4.2 Scalar Operators

An overbar applied to a primitive function symbol will be used to denote an operator related to the function in the manner illustrated by the following:

```
F+̄G X ←→ (F X)+(G X)
F×̄G X ←→ (F X)×(G X)
```

Use of these operators in conjunction with composition will be illustrated by some familiar identities of the calculus, using the monadic operator Δ for the derivative operator:

```
F+̄GΔ ←→ FΔ+̄(GΔ)

F×̄GΔ ←→ FΔ×̄G+̄(GΔ×̄F)

F¨GΔ ←→ FΔ¨G×̄Δ)
```

If, instead of adopting notation of the form S={x | Px} for sets, we recognize that the set is characterized by the proposition (i.e., function with range 0 1) P, and work directly in terms of the proposition, then the logical operators ∧̄,∨̄,<̄, etc., provide the operators appropriate to the use of sets.

For example:

P∧̄Q	Intersection
P∨̄Q	Union
P>̄Q	Difference
P≠̄Q	Symmetric difference
P⍲̄Q	Complement of intersection

4.3 The Axis Operator

The *shape* of an APL array (given by the expression ⍴A) is a vector which specifies the number of items along each of the axes of A, and the *rank* (given by ⍴⍴A) is the shape of this vector, that is, the number of axes. Thus the rank of a matrix is 2, of a vector is 1, and of a scalar (which has no axes) is 0.

Each APL function is defined upon arrays of specified rank, and is said to have the corresponding *rank*. For example, the familiar arithmetic functions are defined upon scalars (rank 0), reversal (denoted by ⌽ and defined to reverse the order of the elements of its argument) is defined upon vectors (rank 1), matrix inverse is of rank 2, and ravel (denoted by , and defined to produce a vector list of the elements of its argument in row-major order) is defined on any array and is said to have unlimited rank.

It is important to be able to extend functions to apply to arrays of higher rank, treating the argument as a collection of subarrays of rank appropriate to the particular function. Scalar functions (of rank 0) present no problem, but a vector function such as reversal might be applied to each of the arguments along the last axis of a matrix, thereby reversing the order of each of the rows, or along the first axis, reversing each of the columns.

In general, it is necessary to specify the axes to which a function applies. This facility has been provided in APL, in a limited and anomalous manner, by the use of indices enclosed in brackets. We will now consider an *axis* operator, to be denoted by ∘̈, which applies to a function left argument F and a vector of axis indices AI as right argument to produce a derived function which applies the function F along axes AI. A function used without the axis operator applies automatically to the last axes of its argument.

Many APL functions are *uniform* in the sense that the shapes of their results depend only upon the shapes of their arguments. Thus reversal, matrix inverse, and ravel (which produces a result of shape ×/S when applied to an array of shape S) are uniform, but ⍳ is not.

Any uniform function can be extended to apply to an array of higher rank by using the axis operator to specify the axes to which it applies, and decreeing that the axes of its individual results occur last in the overall result. This will be illustrated by showing the shapes resulting from applying the ravel function to various axes of an array *A* of shape 2 3 5 7:

```
      ρ,⍤2 3 A ↔ 2 3 35
  ρ,⍤0 1 2 3 A ↔ 210
      ρ,⍤1 3 A ↔ 2 5 21
    ρ,⍤1 2 3 A ↔ 2 105
      ρ,⍤0 1 A ↔ 5 7 6
        ρ,⍤0 A ↔ 3 5 7 2
```

We will now treat non-uniform functions by employing composition with two functions useful in the construction and use of general arrays or data structures, namely, *enclose,* and *disclose. Enclose,* to be denoted by <, is a function of unlimited rank which produces a *scalar encoding* of its argument *A*. The result is of rank 0, and bears no usable relationship to the original argument except that the inverse function *disclose* (>) restores it, that is, ><*A*↔*A*. The disclose function is of rank 0, that is, it is a scalar function. The disclose function applied to a *simple* scalar *S* (i.e., one not produced by enclosure), behaves as the identity function, that is, >*S*↔*S*.

For any function *F*, the composition <¨*F* produces a uniform function which is simply related to *F* and, because it is uniform, can be systematically extended to an array of any rank. For example, if *R*←<¨ι 2 3 4 5, then *R* is a four-element vector such that >*R*[0]↔0 1, and >*R*[1]↔0 1 2, etc. Finally, the composition <¨*F*¨> produces a scalar function, which applies to scalars to produce scalars. It is normally used to apply to arrays whose elements are themselves results of the enclose function.

5. OPERATORS IN OTHER LANGUAGES

We will now examine the use of operators in a few of the other major programming languages, including BASIC, FORTRAN, ALGOL, PL/I, and LISP. It appears that practicing programmers in these languages neither use operators nor think in terms of them. A reader conversant with any particular language (including those mentioned above) may find it instructive to program some of the problems treated by operators in the earlier sections of this paper.

There are three aspects of interest: inherent operator facilities, the constructibility of such facilities within the language, and the usability and actual use of such facilities by practicing programmers.

5.1 Inherent Facilities

Although none of the cited languages appear to recognize operators as such, they do provide certain constructs which make it convenient to modify the application of functions. For example, a DO LOOP used to perform the equivalent of +/ can be modified to perform ×/ by substituting × for +, and the DO LOOP can therefore be construed as an operator acting upon some function embedded in the body of the loop. However, even in this simple case of reduction, it is also necessary to change the initialization of the loop according to the value of the identity element of the function.

The ability to pass arguments by name (as in ALGOL) would make it possible to use constructions such as DO LOOPS more like true operators, although in the case of reduction it would be necessary to include the identity element as an argument as well as the function name.

LISP, like any language based upon the lambda calculus, has the ability to define operators in the same manner that it defines functions, but makes no distinction between them.

5.2 Constructibility of Operators

We will now examine the possibility of constructing procedures which may be used directly in the manner of operators. We begin by defining a LISP procedure REDUCE which is analogous to the reduction operator. More specifically, the expression (REDUCE PLUS) produces a function which may be applied to an argument list as in the following example:

```
( (REDUCE "PLUS) "(1 2 3 4 5))
VALUE = 15
```

The definition of REDUCE (in LISP 370) follows:

```
(DEFINE "( REDUCE
   (LAMBDA (OP)
      (SUBST
         OP
         "OP
         "(LAMBDA (%G1098)
            (PROG (%G1099)
               (SETQ %G1099 (CAR %G1098))
               (MMAPC
                  (LAMBDA (%G1101)
                     (SETQ %G1099 (OP %G1099 %G1101)))
                  (CDR %G1098) )
               (RETURN %G1099) ) ) ) )
))
```

The parentheses surrounding REDUCE "PLUS in the expression ((REDUCE "PLUS) (1 2 3 4 5)) are important, since they imply that REDUCE "PLUS produces a result (a function) independently its eventual arguments, a result which can therefore be used as argument to a further operator. It appears to be impossible to achieve this same effect in PL/I or ALGOL.

In PL/I, for example, one might define composition by the procedure:

```
COMP: PROC (F,G,X) RETURNS(FIXED);
      DCL (F,G) ENTRY(FIXED) RETURNS(FIXED);
          X  FIXED;
      RETURN(F(G(X)));
      END;
```

which could then be applied as in COMP(NOT,NOT,X). However, this is not satisfactory since the arguments include the eventual argument of the composed function to be produced.

The following attempt to produce a function independent of its arguments reveals the source of the difficulty:

```
COMP:  PROC(F,G) RETURNS(ENTRY);
        DECLARE (F,G) ENTRY(FIXED);
        DUMMY:PROC(X) RETURNS(FIXED);
            DECLARE X FIXED;
            RETURN (F(G(X)));
           END;
        RETURN(DUMMY);
      END;
```

The block structure of PL/I ensures that this will not work because the names F and G will disappear upon completion of the procedure DUMMY.

Attempts to construct proper operators in ALGOL or PL/I would be further frustrated by the distinctions imposed by types, and by the necessity to refer to primitive functions indirectly, as in COMP(NOT,NOT) together with a definition of the procedure NOT rather than using (COMP(¬,¬) directly.

ACKNOWLEDGMENTS

I am indebted to several of my colleagues in the T.J. Watson Research Center, to Mr. A.D. Falkoff for critical reading of the manuscript, and to Messrs. P. Lucas, C. Alberga, M. Wegman, and M. Mikelsons for help in the discussion of PL/I and LISP.

REFERENCES

1. *The International Dictionary of Applied Mathematics,* Van Nostrand, Princeton, N.J., 1960.
2. *APL Language,* Form GC26-3847, IBM Corporation.
3. Falkoff, A.D., and Iverson, K.E. The design of APL. *IBM J. Res. Develop. 16,* 4 (July 1973).
4. Falkoff, A.D., and Iverson, K.E. The evolution of APL. Proc. of the ACM SIGPLAN Conference on the History of Programming Languages, June 1-3, 1978.
5. Iverson, K.E. The role of operators in APL. *APL Quote Quad 9* (1979).
6. Iverson, K.E. *A Programming Language.* Wiley, New York, 1962.
7. Iverson, K.E. Operators and functions, Res. Rep. RC7091, IBM Corporation, 1978.
8. Iverson, K.E. Two combinatoric operators. Proc. of ACM STAPL Conference, 1976.
9. Iverson, K.E. The derivative operator. *APL Quote Quad 9* (1979).

Received May 1978; revised January 1979