APL 82, W. H. Janko, W. Stucky (ed.),
APL Quote Quad, Vol. 13 No. 1,
© 1982 by ACM, Inc.

PRACTICAL USES OF A MODEL OF APL

Kenneth E. Iverson
I.P. Sharp Associates,
Box 418, Exchange Tower,
2 First Canadian Place,
Toronto, Ontario
Canada M5X 1E3

Arthur T. Whitney
11033 80th Avenue
Edmonton, Alberta
Canada T6G 0R2

ABSTRACT

This paper discusses the use of a general model of APL (written in APL) which allows convenient definition of new operators and functions and experimentation with their use. Use of the model is illustrated by a number of functions and operators, some of which have been previously discussed, and some (such as the operator *til*) which are new. Details of the model itself are not treated.

The effective use of new facilities introduced into APL systems is often long delayed, not only because of a programmer's tendency to cling to familiar ways, but also because the abstract, formal treatment necessary to the specification of a new facility often makes its assimilation and use seem unduly difficult. Working models of new facilities available before their actual introduction can be very helpful in teaching their use, and very effective in speeding their widespread application.

However, the development of an accurate model for each new facility can, especially in the case of new operators, be burdensome, and we have found that a general model of the APL interpreter, developed primarily for use by the language designers in the design and modelling of new extensions, can also be useful in providing specific models for the enlightenment of users.

This paper discusses such uses of the general model, employing illustrations executed by the version available on the I.P. Sharp Associates APL system. Except for information necessary for understanding its use, details of the model will not be discussed here.

The model incorporates a number of facilities proposed in earlier papers [1 2 3]; the more fundamental among them are illustrated below. The model, invoked by executing the function *ΔAPL*, accepts character input and parses and executes the expression entered. 0-origin indexing is used throughout the paper.

```
      ΔAPL

      A←1 2 3                          ← may be used to assign
      MP←+.×                           names to functions and
      DOT←.                            operators as well as to
                                       variables [1 3]

      A MP A
14
      A + DOT × A
14

      CF←'α+÷ω' ∇ 'ωΔω'                The function
                                       definition operator ∇
      A CF A                           may be used to define
2 2.5 3.33333333                       ambivalent functions [3]
      CF A
2 2.5 3.33333333

      CF/A                             Operators apply to
1.42857143                             defined and derived
                                       functions as well as
      CF\A                             to primitives
1 1.5 1.42857143
      +.×⌿ 3 2 2⍴⍳12
118 131
518 575

      Q←×∇-                            The operator ∇ applies to
      A Q A                            functions as well as to
                                       character vectors
1 4 9
      Q A
¯1 ¯2 ¯3
      R←*∇'10*ω'
      3 R A
3 9 27
      R A
10 100 1000
```

© 1982   ACM 0-89791-078-8/82/007/0140 $00.75

```
      ⊣←'α'∇''                         An assignment can be
                                       made to a graphic
                                       name (⊣ is now the
                                       function lev which
                                       returns its left argument)

      C1←2 2ρ'BARE'⊣C2←2 2ρ'BEAR'

      C1
BA
RE
      C2
BE
AR
      x̄←+.×                            New composite symbols
                                       can be used

      A x̄ A
14
```

Assignment of values to graphic symbols is, of course, intended for use in language design, and is not proposed for general inclusion in the language. It is, nevertheless, convenient for use in experimentation by programmers as well as by language designers.

The use of new composite symbols is possible because the model receives and handles *arbitrary input* (⎕*ARBIN* in IPSA APL), that is, all terminal inputs including backspace and attention are transmitted directly to it.

The model also incorporates a definition from [1] which illustrates the notion that an operator may produce an ambivalent derived function - specifically, the monadic derived function produced by reduction is supplemented by a dyadic case in which the left argument specifies the width of the "window" over which reduction is applied. For example:

```
      P←13 11 7 5 3 2
      2 -/ P                           Pairwise differences
2 4 2 2 1
      (4+/P)÷4                         Running Averages
9 6.5 4.25
      ∧/2 ≥/P                          Test for decreasing
1                                      sequence
```

A. SOME NEW FUNCTIONS

We will further illustrate the use of the model in simple cases by defining a few functions (most of which have been defined in earlier papers [1 2]) for use in later examples:

```
      ∩←'(L∊ω)/L←,α' ∇ ''        Set intersection

      ~←'(~L∊ω)/L←,α' ∇ ~        Set difference.
                                 Note that the monadic
                                 definition of ~ is preserved.

      'ABACUS' ∩ 'SCAB'
ABACS

      'ABACUS' ~ 'SCAB'
U
```

```
      ∪←'' ∇ '((ιρω)=ωιω)/ω←,ω'    Nub (set
                                   of distinct elements)

      ∪ 'ABACUS'     Distribution [1]
ABCUS

      ⍦←'' ∇ '(∪ω)∘.=ω'

      ⍦ 'ABACUS'
1 0 1 0 0 0
0 1 0 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1

      ⍦ ι3                         ⍦ applied to any vector
1 0 0                              of distinct elements
0 1 0                              produces an identity matrix
0 0 1

      ⎕←W←R⊥⍦ιρR←8 4 5 2
40 10 2 1                          The weights associated
                                   with the radix R

     +/W×1 2 3 4
70
      R⊥1 2 3 4
70
```

B. ENCLOSED ARRAYS

Enclosed arrays provide a good example of a notion whose formal treatment makes it seem unduly difficult. For example, a graphic display of enclosed arrays is very helpful to the beginner, but the definition of a suitable display is recursive and somewhat complicated. However, its behaviour is immediately evident from a model. The definition of display, (i.e., of the function ⍒) used here, is a modification (due largely to P.K. Wooster) of the definition in [2], and may be illustrated as follows:

```
      ⎕PS←0 0 ¯3 ¯3

      ⎕←Q←2 2ρ(<A),(<(<C1),<C2),<A∘.×A
        |¯¯¯¯¯¯¯¯¯¯|
|¯¯¯¯¯| ||¯¯|  |¯¯||
|1 2 3| ||BA|  |BE||
|_____| ||RE|  |AR||
        ||__|  |__||
        |__________|

|¯¯¯¯¯|
|1 2 3|  |¯¯¯¯¯|
|2 4 6|  |1 2 3|
|3 6 9|  |_____|
|_____|
```

The last two elements of the system variable ⎕*PS* determine the row and column spacing between enclosed elements, injecting "box" decorations in one of the spaces if the arguments are negative. The first two elements control the vertical and horizontal positioning of the elements, ¯1 denoting top (or left) justification, 0 centering, and 1 bottom (or right) justification.

For example:

```
      ⎕PS←¯1 1 0 3

      Q
1 2 3    BA    BE
         RE    AR
1 2 3      1 2 3
2 4 6
3 6 9
```

In later examples it will be convenient to use two further functions. The first is the boolean *match* (called *idem* in [3]), denoted by ≡, which compares its left and right arguments and yields a scalar 0 unless they match in every respect. We will also define its monadic case to yield 1 if its argument is *simple*, that is, contains no enclosed elements.

The second function is a variant of *link*, defined as follows:

```
      ;←'(<α),Δω' ∇ 'ω ◇ <ω ◇ →≡ω'
```

In words, ;*X* encloses *X* if *X* is simple, and *X*;*Y* catenates the enclose of *X* to *Y*, first enclosing *Y* if it is simple. For example:

```
      ⎕PS←0 0 ¯3 ¯3

       C1;C2
 __    __
|  |  |  |
|BA|  |BE|
|RE|  |AR|
|__|  |__|

      C1;C2;A
 __    __    _____
|  |  |  |  |     |
|BA|  |BE|  |1 2 3|
|RE|  |AR|  |_____|
|__|  |__|

       ;/C1
 __    __
|  |  |  |
|BA|  |RE|
|__|  |__|

       Q≡2 2ρA;(C1;C2);A∘.×A
1
```

The Δ occurring in the definition of ; is a "self-reference" to the function being defined [3]; since it is used monadically, it refers to the monadic case and therefore encloses its argument if it is simple. It should also be noted that this use of the semicolon as a function need not conflict with its use as a separator in bracket indexing.

C. THE DISPOSITION OF AXES

The effect of an axis operator in an expression such as *F*⍤1 0 *T* is simply to apply the function *F* to the subarrays along the indicated axes of the argument *T*, and the axes resulting from applying *F* to each subarray are placed last in the overall result. As simple as these rules are, their effects can be bewildering until the user develops some familiarity with their application in specific instances. Such familiarity can be gained by applying some simple function (such as ravel or scan) in a number of cases:

```
      ⎕←T←3 2 4ρι24
 0  1  2  3
 4  5  6  7

 8  9 10 11
12 13 14 15

16 17 18 19
20 21 22 23
```

```
       ,⍤1 2 T
 0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
```

The results of the ravelled 2-by-4 subarrays are placed along the last axis.

```
      ,⍤0 1 T
 0  4  8 12 16 20
 1  5  9 13 17 21
 2  6 10 14 18 22
 3  7 11 15 19 23
```

The results of the ravelled 3-by-2 subarrays are placed along the last axis.

```
      ,⍤0 T
 0  8 16
 1  9 17
 2 10 18
 3 11 19

 4 12 20
 5 13 21
 6 14 22
 7 15 23
```

Although the ravel of each vector along axis 0 produces no change, the placement of the result along the last axis effects an overall transposition.

```
      +\⍤0 T
 0  8 24
 1 10 27
 2 12 30
 3 14 33

 4 16 36
 5 18 39
 6 20 42
 7 22 45
```

Because the axis resulting from the scan of each vector is placed last, the result differs (by a transposition) from the result of +\[0]*T*. Compare with the preceding result.

Although the definition of the axis operator used here is adopted from [2], the representation of the right argument *T* differs slightly; the older definition would be satisfied by replacing *F*⍤*I* by *F*⍤(;*I*). In other words, an enclosed right argument is treated as in [2], but a simple argument is treated as a single entity and specifies (in the dyadic case of the resulting function) both the left and right axes of application. For example:

```
      C1
BA
RE
      C2
BE
AR
      C1,⍤1 C2
BABE
REAR
      C1,⍤0 C2
BRBA
AEER
```

D. COMPOSITION AND RELATED OPERATORS

For a single scalar argument *S*, the composition of two functions *F* and *G* (denoted by *F*⍤*G*) is easy to understand, since *F*⍤*G S* is equivalent to *F G S*. For example:

```
      ι⍤⌊ 3.6                         ι⌊ 3.6
0 1 2                           0 1 2
      +/⍤ι⍤⌊ 3.6                      +/ι⌊ 3.6
3                               3
```

However, much of the importance (and difficulty) of composition stems from its use with non-scalar arguments, in which case the entire composed function *F*⍤*G* is applied individually to each of the subarrays to which *G* (according to its axes of application) applies. For example:

```
      +/⍤ι⍤⌊ V← 2.5 3.6 4.9 6.4
1 3 6 15

      ⎕←L← <⍤ι⍤⌊ V
|¯¯¯| |¯¯¯¯¯| |¯¯¯¯¯¯¯| |¯¯¯¯¯¯¯¯¯¯¯|
|0 1| |0 1 2| |0 1 2 3| |0 1 2 3 4 5|
|___| |_____| |_______| |___________|

      +/⍤> L
1 3 6 15
```

Note that the expressions +/ι⌊ and <ι⌊ and +/> corresponding to the foregoing compositions would not even be valid for the given arguments. In some cases, the corresponding expressions are valid, but yield different results. For example:

```
      A←3 2 2ρ1 0 1 1 1 0 2 1 1 0 3 1
      A             ⌹A            ⍉⍤⌹ A          ⍉⌹ A
1 0           1  0          1 ¯1            1  1  1
1 1          ¯1  1          0  1           ¯1 ¯2 ¯3

1 0           1  0          1 ¯2            0  0  0
2 1          ¯2  1          0  1            1  1  1

1 0           1  0          1 ¯3
3 1          ¯3  1          0  1
```

The operator denoted by ¨ (called *with* in [2]) applies in two distinct ways. Applied to a function and a variable, it produces the monadic function obtained by supplying the variable as one argument of the (dyadic) function. For example:

```
      (EXP←10¨*) 1 2 3
10 100 1000
      (SQ←*¨2) 1 2 3
1 4 9
```

Applied to two functions, as in *F*¨*G*, it is equivalent to composition, with a final application of the function *GI* which is inverse to *G*. Thus, for a scalar argument *S*, we have *F*¨*G S* ↔ *GI F G S*. The extension to general arguments is like that of composition. For example:

```
      +\¨⌽V←1 2 3 4 5
15 14 12 9 5

      ⌈¨(*¨.5) V
1 4 4 4 9
```

```
      ⎕←M←3 4ρι12
0  1  2  3
4  5  6  7
8  9 10 11

      +/¨(*¨2⍤0)M                         Lengths of
                                          columns
8.9442719 10.3440804 11.8321569 13.3790882

      +/¨(*¨2⍤1)M                         Lengths of
                                          rows
3.74165739 11.22497216 19.13112647
```

In the foregoing discussion of the operators ¨ and ⍤, only the monadic case of the derived function was considered, that is, it was applied to a single argument. The dyadic cases of *F*⍤*G* and *F*¨*G* are similar, the function *G* being applied (along appropriate axes) to both the left and right arguments. For example:

```
      A  +⍤÷ B←1+A←1 2 3
1.5 0.83333333 0.58333333
      A+¨÷B
0.66666667 1.2 1.71428571
      C ×⍤(+/⍤0) C←3 3ρι9
81 144 225
```

The composition in the expression *A F*⍤*G B* (referred to in [2] as *F on G*) applies the dyadic (case of the) function *F* to the results of the monadic function *G*; the composition in the expression *A F*⍥*G B* (to be referred to as *F upon G*) applies the *monadic* function *F* to the result of *dyadic G*, as in *F A G B*. The difference may be illustrated as follows:

```
      C1                   C2
BA                   BE
RE                   AR
      C1 ,⍤, C2
BAREBEAR
      C1 ,⍥, C2
BABEREAR
      A <⍤÷ B←⌽A←1 2 3 4
0 0 1 1
      A <⍥÷ B
|¯¯¯¯| |¯¯¯¯¯¯¯¯¯¯| |¯¯¯| |¯|
|0.25| |0.66666667| |1.5| |4|
|____| |__________| |___| |_|
```

We will conclude this section with a rather complex expression involving operators, the purpose being to illustrate how to "read" such expressions. If:

```
      A←2 3 4 5ρι120
      V←(I←1);(J←2 0);(K←3);(L←4 2 0)
```

then it is possible to write a function of *V* such that the ravel of *A* indexed by its result is equivalent to *A*[*I*;*J*;*K*;*L*]. Thus:

```
      A[I;J;K;L] ≡ (,A)[>∘.+¨>/V×¨>R⊥⊎ιρR←ρA]
```

An understanding of the expression in the brackets can be gained by executing it piece-by-piece:

```
      ⎕←Z←R⊥⊎ιρR←ρA
60 20 5 1
      V
|¯| |¯¯¯| |¯| |¯¯¯¯¯|
|1| |2 0| |3| |4 2 0|
|_| |___| |_| |_____|
      ⎕←Y←V ×¨> Z
 __    _____    __    ______
|  |  |     |  |  |  |      |
|60|  |40  0|  |15|  |4  2 0|
|__|  |_____|  |__|  |______|

      ∘.+¨>/Y
 ____________
|            |
|119 117 115 |
| 79  77  75 |
|____________|
```

The expression for Z (whose details were shown in Section A) yields the weights appropriate to the successive axes in indexing the ravel of the argument A. The vector Y is obtained by multiplying each element of the list of indices (V) by the appropriate weight, and the final result shown is the application of the plus-outer-product (`∘.+`) between each of the disclosed elements of Y.

E. THE DEFINITION OF OPERATORS

A dyadic operator can be defined in terms of expressions involving its two arguments (to be denoted by α̲ and ω̲) and the two arguments of the derived function produced (to be denoted by α and ω). For example, if the dyadic operator ×̄ were defined by the expression `'(α̲α)×ω̲ω' ∇ 'ωΔω'`, then `3.5 ⌊×̄⌈ 6.5` would yield `21`, and `3.5 ⌈×̄⌊ 6.5` would yield `24`.

Since either or both of the arguments α̲ or ω̲ may be either a variable or a function, and since definitions must sometimes be provided for two or more of the four possible cases, each definition will be associated with an indication of the case it defines. For example, if

```
     ×̄←(1 1;0 1) ∇ ('''(α̲α)×ω̲ω''∇''ωΔω''';
           '''''∇''α̲¨×⍥ω̲ω''')
```

then `⌊×̄⌈` behaves as described above, and `3×̄⌈ 6.5` yields `21`.

In general, the definition of a dyadic operator requires as the left argument of ∇ a vector whose Kth element is the enclosed representation (using `0` to denote a variable and `1` to denote a function) of the case represented by the Kth element of the right argument. In the definition of a monadic operator, each case is indicated by a single quantity, and the left argument of ∇ is therefore one of `0`, or `1`, or `0;1`, or `1;0`.

We will conclude this section with the definition of a new dyadic operator denoted by ⍨ (and called *til* as an abbreviation of *tilde*):

```
     ⍨←1 1 ∇ '''(ω̲ω)α̲α''∇''ωΔω'''
```

This operator is of great practical interest because it subsumes a number of important special cases arising from its use with the identity function ⊢ (defined as `⊢←'ω'∇'ω'`), and with a sequence of functions, as in F⍨G⍨H:

a) Since

```
     α F⍨⊢ω ↔ (⊢ω) F α ↔ ω F α
```

the function F⍨⊢ is the *commutation* of F.

b) Since

```
     α F⍨G⍨H ω ↔ (Gα) F (Hω)
       F⍨G⍨H ω ↔ (Gω) F (Hω)
```

the operator *til* provides a generalization of the operators illustrated above, for instance by the operator ×̄, applying an arbitrary dyadic function F to the results of two monadic functions G and H. Moreover, the use of the identity function for either G or H provides cases which might be called "left composition" and "right composition":

```
     F⍨G⍨⊢ω ↔ (Gω) F ω
     F⍨⊢⍨Hω ↔      ω F (Hω)
```

For example, if the propositions P and Q are defined by `P←2¨|` and `Q←3¨<`, then:

```
      ∧⍨P⍨Q V←1 2 3 4 5 6 7
0 0 0 0 1 0 1

      ∨⍨P⍨Q V
1 0 1 1 1 1 1

      +⍨P⍨Q V
1 0 1 1 2 1 2
```

Finally, the function CF (defined in the introduction by `'α+÷ω'∇'ωΔω'`) may now be defined by `CF ← +⍨⊢⍨÷` or, since + is commutative, by `CF ← +⍨÷`, and the expression for indexing the ravel of an array A (given at the end of Section D) can now be defined by:

```
Q←>⍥(∘.+¨>/)⍥(×¨>⍥¯1⍨(⊥⍨ρ⍨(ʉ⍥ι⍥ρ⍥ρ)))
```

F. MERGE AS A DYADIC CASE OF SELECTION

If S is a dyadic selection function such that I S A selects from A an array determined by the "indices" (or other parameters) provided by the variable I, then the derived function I¨S is a derived selection function which is monadic, and selection from A is obtained by the expression I¨S A. For example, if `M←3 3ρι9`, then `0 0¨⍉` is a monadic selection function such that `0 0 ¨⍉` selects the diagonal vector `0 4 8` when applied to M.

Consider a definition of the dyadic case of such a derived function which returns the entire right argument, but with the elements of the left argument substituted for the selected elements of the right argument. The dyadic function therefore provides a *merge* of its arguments. For example:

```
      M←3 3ρι9
      M                                  (10+ι3) 0 0 ¨⍉ M
0 1 2                               10  1  2
3 4 5                                3 11  5
6 7 8                                6  7 12
```

This definition of a dyadic case will be extended to other selection functions in the same manner. Consider, for example, the following indexing function, defined in terms of the function Q defined at the end of the preceding section:

```
      S←'(,ω)[ωQα]'∇''
      ⎕←I←?4 2ρ3
                                               I S M
1 2                                            5 6 0 3
2 0
0 0
1 0
```

The function ⌷ (called *from*) is incorporated in the model using the definition of *S* above, but adopting the merge definition for the dyadic case of the derived function *I*¨⌷. Thus:

```
      (10+ι4) I¨⌷ M
12 1   2
13 4  10
11 7   8
```

A different approach to the treatment of merge functions may be found in [4].

G. FUNCTION ATTRIBUTES

An APL function is defined not only by the result it produces for each argument in its domain, but by certain *attributes*, some of which (like the axes of application) are manifest in every use of the function, and some of which (like the *identity element* and the *inverse*) become manifest only upon the application of some operator. For example, the results of the functions ∧ and ⌊ are identical for boolean arguments, but differ in their identity elements, ∧/ι0 yielding 1, and ⌊/ι0 yielding infinity; the inverse of a function is manifested in the application of the dual operator.

The representation of attributes will be introduced into the representation of functions as follows: any segment (as delimited by diamonds) which begins with a right parenthesis represents an attribute; the first token following the parenthesis denotes the name of the attribute specified, and what follows (after the delimiting space) specifies the attribute.

For example, the name Δ⌿ denotes the identity element attribute, and the expression *MP*←'α+.×ω ◇)Δ⌿ ᵿι1↑ρω'∇'' defines *MP* as a matrix product function whose identity function produces an identity matrix of appropriate shape. Thus, if ρ*A* ↔ 5 4 4, then *MP* ⌿ 5 0 0↓*A* is a 4-by-4 identity matrix.

The following is a list of attributes (together with suggested names) which we have found useful and have incorporated into the APL model:

1. Axes (Δ⍤)

   The axes of application, which may be specified independently for the monadic and dyadic cases.

2. Scope (∼;)

   The names to be exempted from the localization otherwise applied to names which are assigned values within a function. Except for the inclusion of the name (∼;) for identifying the attribute, the scheme agrees with that proposed in [3].

3. Identity Element (Δ⌿)

   The identity elements associated with the primitive functions are (because limited to scalar functions) all constants; in the more general case they may, like the example of matrix product given above, be functions of the shapes of the arguments.

4. Merge (αα̲¨Δω)

   This attribute will indicate that the derived function α̲¨Δ (for a variable α̲) is a selection function which extends to a *merge* in the dyadic case in the manner discussed in Section F.

5. Inverse (Δ⍣¯1)

   This attribute gives the inverse function to be used when the dual operator is applied to the function. The name Δ⍣¯1 used above arises from the *power* operator ⍣ incorporated in the model (which applies its monadic left argument the number of times specified by the variable right argument). The result of *F*⍣¯1 is the inverse of *F*. For example, *⍣¯1 is equivalent to ⍟.

6. Inverses: left and right cases (α¨Δ⍣¯1) (Δ¨ω⍣¯1)

   If *V* is a variable and *F* is a function, then *V*¨*F* produces a monadic function (the *left case* of *F*), and *F*¨*V* produces a function, either or both of which may possess inverses. These attributes provide the appropriate inverses.

   For example, the inverses of +¨*V* and *V*¨+ are both (-*V*)¨+, the inverse of -¨*V* is +¨*V*, and *V*¨- is self-inverse.

7. Derivative (ΔΔ̈)

   The model incorporates the specification of derivatives for a few cases. For example, 1¨○Δ̈ is equivalent to 2¨○.

8. Variant (Δ:)

   The model provides a few cases of the variant operator discussed in [1]. For example, (ι:1) 3 is 1 2 3, and (ι:0) 3 is 0 1 2. This use of the colon need not conflict with the present use provided that (as proposed for the double use of the semicolon) the older usage is given priority. Thus the colon in *L*:*A B C* at the beginning of an expression will indicate that *L* is a label, and the use of the variant *L*:*A* would (at the beginning of an expression) have to be indicated by parentheses, as in (*L*:*A*) *B C*.

REFERENCES

1. Iverson, K.E., Operators and Functions, Research Report #RC7091, IBM Corp., 1978.

2. Bernecky, R., and Iverson, K.E., Operators and Enclosed Arrays, **APL Users Meeting**, I.P. Sharp Associates, 1980.

3. Iverson, K.E., and Wooster, P.K., A Function Definition Operator, **APL Quote Quad**, Vol.12, No.1, Sept. 1981 (Proceedings of APL81).

4. Pesch, R.H., Indexing and Indexed Replacement in APL, **APL Quote Quad**, Vol.12, No.1, Sept. 1981.