APL SYNTAX AND SEMANTICS

Kenneth E. Iverson

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

This paper presents a working model of APL syntax and semantics that incorporates explicit representations of functions, operators, and syntax, thus providing a basis for the clear and explicit statement of extended facilities in the language, as well as a tool for experimentation upon them. Use of the model is illustrated in the treatment of the syntax of operators, and in the discussion of a number of new or recently-proposed facilities including indirect assignment, the operators *axis, derivative, inverse,* and *til,* and the functions *link,* and *from.* The entire model is included in an appendix.

The model is expressed in SHARP APL as extended in [1] but, because it uses few special features (enclose, disclose, close composition, and dual) it should translate easily into other systems (such as NARS [2] and APL2 [3]) that provide some form of enclosed arrays.

We will begin with the overall behaviour of the model as seen in the definition and use of the two outer functions *APL* and *S* (the "stack manager" that applies to the left stack *L* of the expression to be evaluated, and a right stack *R* of intermediate results), and continue with the tabular definition of syntax, and the representations of functions and operators. 0-origin indexing is used throughout, and enclosed arrays are normally displayed within enclosing vertical bars, as determined by the setting □*PS*← ¯1 ¯1 0 ¯3 (see Reference [4]).

The function *APL* accepts literal input and executes the expression entered, using definitions of extended functions and operators already provided. For example:

```
    APL
    Q←3 2 4⍴⍳24                    Reduced indent
                                   within the model.
    ,⍤1 2 Q                        Ravel along
 0  1  2  3  4  5  6  7            axes 1 2
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
```



```
    F←,⍤1 2                        Assign the name F
                                   to this ravel
    F Q
 0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23

    ⍎→                             An expression preceded by ⍎ is executed in raw
                                   APL; this exits
```

The function *APL* and its main supporting function are defined as follows:

```
APL;Z;X;NORE;□PS
NORE←<< 0 2 3 5 7 ⍴0×□PS← ¯1 ¯1 0 ¯3
L1:→(∧/' '=X←⍞,⍞←'    ')/L1
→('⍎'≠1↑X←(+/∧\' '=X)↓X)/L2
⍎1↓X
→L1
L2:Z←(X←TK X) S ''
→('←'=''⍴,>1↑(N X)/X)/L1
→L1,0⍴□←⍎((0=⍴⍴>Z)/'>'),'>Z'

Z←L S R
Z←⍎,>ACT[''⍴AC R]
```

The main action is the application of the stack manager *S* to an empty right stack and a left stack of enclosed individual **tokens** (names, primitives, constants, etc.) produced by the tokenizing function *TK*. The function *S* simply executes one of a set of actions represented in the vector *ACT*, selection from *ACT* being determined by the *action and classes* function *AC*, which in turn depends on the syntax table *ST*. The following display of *ACT* appears in enclosing bars because of □*PS*←¯1 ¯1 0 ¯3:

```
      ⍪,⍤<ACT

|(>''⍴Z)S(1↓Z←L LE 4=1↑,⍴⍤>⍴⍤>R),R|
|L S((>>R[0]) LS R[2]),3↓R|
|L S(2↑R),(R[2]RE R[,3]),4↓R|
|L S(1↑R),(R[1]RE R[,2]),3↓R|
|L S(1↑R),(R[2]RE R[1 3]),4↓R|
|L S(1↑R),(R[2]RE R[,1]),3↓R|
|⍎'R[1]',(2<⍴R)/'R NOT CLEAR⋄'|
```

In order to provide convenient tracing of the execution we incorporate in $\underline{S}$ three uses of a *trace* function $\underline{TR}$ as follows:

```
Z←L S R
Z←⍪,>ACT[''⍴' ACS' TR AC 'R: ' TR R],
      0⍴'L: ' TR L
```

The display produced by $\underline{TR}$ consists of its left argument followed by its right, except that the number of rows displayed is limited by the magnitude of the trace control variable $T0$; if $T0$ is positive, the display is suppressed (except for a blank line) if the left argument of $\underline{TR}$ begins (as it does in the first occurrence of $\underline{TR}$) with a space. Finally, the display of a function is limited to its primary part, the body and axes. Thus:

```
      APL
    ⍎T0←1
    (⍋4)+5
L:  |(| |⍋| |4| |)| |+| |5|
R:
L:  |(| |⍋| |4| |)| |+|
R:  ||5||
L:  |(| |⍋| |4| |)|
R:  |||⍎<<(>>⍺)+>>⍵|| ||⍎<<⍋>>⍵||| ||5||
L:  |⍋| |4|
R:
L:  |⍋|
R:  ||4||
L:
R:  |||⍎<<(>>⍺)⍋>>⍵|| ||⍎<<⍋>>⍵||| ||4||
L:
R:  |⍎| |||⍎<<(>>⍺)⍋>>⍵|| ||⍎<<⍋>>⍵||| ||4
      ||
L:
R:  |⍎| ||0.25||
L:
R:  ||0.25|| |||⍎<<(>>⍺)+>>⍵|| ||⍎<<+>>⍵|
      | ||5||
L:
R:  |⍎| ||0.25|| |||⍎<<(>>⍺)+>>⍵|| ||⍎<<+
      >>⍵||| ||5||
L:
R:  |⍎| ||5.25||
5.25
```

The five segments of this example beginning with `L:  |⍋| |4|` illustrate the recursive use of $\underline{S}$ to handle parenthesized expressions. The details of the representations of the functions ⍋ and + (whose first lines appear in the displays of the right stack) may be ignored for the moment.

The "left evaluation" function $\underline{LE}$ handles the transfer of successive tokens from the input text to the righthand stack of intermediate results. Because the evaluated result in the right stack has no connection with the original names, the treatment of "side-effects" in expressions such as $(A\leftarrow 3)(A\leftarrow +)A\leftarrow 4$ is clearly defined. For example:

```
    ⍎T0←0
    (A←3)(A←⍋)A←4
0.75
    A
3
```

The example may be repeated with $T0$ set to `1`.

The function $\underline{LE}$ normally evaluates each token and transfers the evaluated result to the right stack, but if the first element on the right stack is an assignment arrow, the evaluation is suppressed. For example:

```
    A←'BCD'
    ⍎T0←1
    A←A,A
L:  |A| |←| |A| |,| |A|
R:
L:  |A| |←| |A| |,|
R:  ||BCD||
L:  |A| |←| |A|
R:  |||⍎<<(>>⍺),>>⍵||    ||⍎<<,>>⍵||
      | ||BCD||
L:  |A| |←|
R:  ||BCD|| |||⍎<<(>>⍺),>>⍵||    ||⍎<<,>>⍵|
      |       | ||BCD||
L:  |A|
R:  |←| ||BCD|| |||⍎<<(>>⍺),>>⍵||    ||⍎<<,
      >>⍵||        | †||BCD||
L:  |A|
R:  |←| ||BCDBCD||
L:
R:  ||A|| |←| ||BCDBCD||
L:
R:  ||BCDBCD||
L:
R:  |⍎| ||BCDBCD||

    ⍎T0←0
    A
BCDBCD
    ⍎→
```

A. SYNTAX

APL syntax questions may be characterized as *old* or *new*, the latter referring to the new questions raised by the general treatment of operators, and the former to old problems introduced by anomalies such as the treatment of brackets and semicolons in indexing, in axis operators, and in mixed output.

The old questions will here be treated as obsolescent, that is, nothing will be done to disturb existing definitions, either to invalidate their use in existing programs, or to extend them and encourage their use. The use of semicolons and brackets is therefore ignored in the model; in an actual implementation they could either be treated by established ad hoc mechanisms, or they could be eliminated by a "preprocessing" translation to equivalent normal expressions.

The new questions are addressed in the model by the *action and classes* function $\underline{AC}$, which examines the stack of intermediate results to determine what action is to be taken next. In the syntax proposed here, this function depends only on the first four elements of the intermediate results, and depends only on the *class* of each of these elements.

The classes and their numeric encodings are as follows:

0 Variable
1 Monadic operator
2 Dyadic operator
3 Function
4 Assignment arrow
5 Left filler (exhaustion of the left stack, denoted in a trace by ⍎)
6 Right filler (exhaustion of the right stack)

The encodings of the first four correspond to the valences of the entities represented (allowing 3 as the sum of the potential valences of a function). They also correspond to the ranks of the arrays whose enclosures represent the entities. Consequently the expression $\rho\ddot{}\circ\cdot>\rho\ddot{}>\omega$ occurring in the function $\underline{AC}$ determines the class of each element of the argument ω.

The syntax rules are, in effect, the manner in which the next action is chosen according to the classes of the intermediate result. This choice is made by the function *AC* (Action and Classes) in two steps:

1. The classes of the first four elements of *R* (completed by the filler code 6) are matched with the rows of the first four columns of the symbol table *ST*, each individual comparison being negated if the element of *ST* is negative; thus an entry ¯2 designates anything *except* a dyadic operator.

2. The first matching row selects the corresponding element of the last column of *ST* to be used (in *S*) as an index to the table of actions *ACT*. The classes are included in the result of the function *AC* only for use in tracing.

The proposed syntax table is defined as follows:

```
      ST

¯7  4 ¯7  6  1
 1  3  0 ¯7  3
 3  3  0 ¯7  3
 5  3  0 ¯7  3
 4  3  0 ¯7  3
 2  0  3  0  2
¯2  0  3  0  4
¯2 ¯1  2 ¯7  4
¯2 ¯1  1 ¯7  5
 5 ¯5 ¯7 ¯7  6
¯7 ¯7 ¯7 ¯7  0
```

However, it can be studied more easily in a display (produced by the function *SYNTAX*) which substitutes for each numeric code a more mnemonic representation, and appends the corresponding action chosen from the table *ACT*. Thus:

```
      ⎕PS←-1 1 0 0
      I←((-7 5 3 2 1),⍳7)⍳0 ¯1↓ST
      C←11 7↑'ALFDMVMDF←LR'[I]
      C,▼ACT[ST[;,4]]
A←AR   L S((>>R[0]) IS R[2]),3↓R
MFVA   L S(1↑R),(R[1]RE R[,2]),3↓R
FFVA   L S(1↑R),(R[1]RE R[,2]),3↓R
LFVA   L S(1↑R),(R[1]RE R[,2]),3↓R
←FVA   L S(1↑R),(R[1]RE R[,2]),3↓R
DVFV   L S(2↑R),(R[2]RE R[,3]),4↓R
DVFV   L S(1↑R),(R[2]RE R[1 3]),4↓R
DMDA   L S(1↑R),(R[2]RE R[1 3]),4↓R
DMMA   L S(1↑R),(R[2]RE R[,1]),3↓R
LLAA   ⍎'R[1]',(2<ρR)/'R NOT CLEAR⋄'
AAAA   (>''ρZ)S(1↓Z←L LE 4=1↑,ρ⍤>ρ⍤>R),R
```

The action and class codes produced by *AC* are displayed if the trace control is set to a negative value. For example:

```
      TO←¯1
      APL
    A←3×4
L: |A| |←| |3| |×| |4|
R:
 ACS|0| |6| |6| |6| |6|
L: |A| |←| |3| |×|
R: ||4||
 ACS|0| |0| |6| |6| |6|
L: |A| |←| |3|
R: |||A<<(>>α)x>>ω|| ||A<<x>>ω||| ||4||
 ACS|0| |3| |0| |6| |6|
L: |A| |←|
R: ||3|| |||A<<(>>α)x>>ω|| ||A<<x>>ω||| ||
      4||
 ACS|0| |0| |3| |0| |6|
L: |A|
R: |←| ||3|| |||A<<(>>α)x>>ω|| ||A<<x>>ω||
      | ||4||
 ACS|4| |4| |0| |3| |0|
L: |A|
R: |←| ||12||
 ACS|0| |4| |0| |6| |6|
L:
R: ||A|| |←| ||12||
 ACS|1| |0| |4| |0| |6|
L:
R: ||12||
 ACS|0| |0| |6| |6| |6|
L:
R: |A| ||12||
 ACS|6| |5| |0| |6| |6|
    AT0←0
    A→
```

The proposed syntax is that embodied in the table *ST*, the selection function *AC*, and the list of actions *ACT*. Experiments with different syntax rules can be made by changes in any or all of these. Such changes can affect the number of elements *R* examined, or can even affect the classes and number of elements produced by the actions. Thus, an action could produce 0 or 2 or 3 results rather than 1 as proposed here, and the results could be operators as well as functions and variables. The table *ST* may be compared with the syntax table of [5], which covers the obsolescent syntax, but not the syntax of operators.

As stated in the introduction, parentheses are handled by an immediate recursive application of the model to the enclosed subexpression. With this premise, the remaining detailed syntax rules can be read directly from the display produced by the function *SYNTAX*. However, the new features that extend the syntax to operators can be summarized as follows:

> Operators take precedence over functions and have long left scope; that is, an operator applies to the result produced by the entire operator sequence to the left of it.

B. REPRESENTATION OF FUNCTIONS AND OPERATORS

The primary definition of a function concerns the specification of what result it produces when applied to an individual array of the lowest rank upon which it is properly defined. However, the complete definition of a function also concerns certain *attributes* which determine the effects of applying various operators to the function. For example, the axis (or axes) of application is an attribute of a function which determines how the function applies to a higher-rank array, an attribute which is modified by an axis operator, as in $\phi[I]A$; the identity element of a function is an attribute that determines the result of the reduction operator in certain cases, as in +/⍳0 or ×/⍳0.

The representation of functions adopted in the present model accommodates thirty attributes (of which 25 are actually used). The cases used are apparent from the following display of *PF*, the enclosed array (of shape 5 2 3) representing the *prototype function:*

```
      ⎕PS← -1 1 0 3
      >PF
|DBODY|        |MBODY|        |∘|
||7.237E75||  ||7.237E75||  ||7.237E75||

|LOPARG|       |ROPARG|       |∘|
|||α|||        |||ω|||        |||ω|||

||||           ||||           |∘|
|IDF A¨Δ|      |IDF Δ¨A|      |IDF Δ|

|DCASE A¨Δ|    |DCASE Δ¨A|    |∘|
|A¨Δ Δ|        |Δ¨AΔ|         |ΔΔ|

|VARIANT0|     |VARIANT1|     |∘|
|A¨Δ⍣¯1|        |Δ¨A⍣¯1|        |Δ⍣¯1|
```

The significance of each of the positions in *PF* will be made clear in the discussion of the corresponding attribute.

The first plane of >*PF* is the primary definition, that is, the bodies of the dyadic and monadic cases, and the application axes. Bodies are represented in the direct definition form defined in [6], with three modifications:

1. A leading ⍝ indicates that what follows is to be executed in raw (i.e., conventional) APL rather than in the APL of the model. Comments are normally allowed in any segment of a direct definition, but because of the special use of the symbol ⍝ they are excluded from use in the model.

2. A label is assigned a vector value consisting of the indices of all segments from the location of the label to the end of the definition.

3. A name is localized only if it is *immediately* adjacent to an assignment arrow (and the mechanism for declaring globals is therefore not used).

The three axes accommodated are in the order left dyadic axis, right dyadic axis, and right monadic axis. The specification of axes is extended to include *negative indexing* (in which ¯1 denotes the ultimate axis, ¯2 the penultimate, etc.) and *complementary* indexing, in which a leading infinite value (denoted by the constant ¯) designates all axes *except* those in the vector following it. Thus, ¯ 2 4 ¯1 denotes all axes except 2,4, and the final axis, and ¯ alone denotes all axes. It may be noted that the axes specified in the prototype function are all of the latter type, making the standard, or *default* axes of application unbounded.

The operator ⍣ (which will be discussed further in Section C), applied in the form *F*⍣∘, produces a function that selects any desired section of the representation of a function *F*.

```
      APL
    D←+⍣∘
    D 0
||⍝<<(>>α)+>>ω||  ||⍝<<+>>ω||  |∘|
||||              ||||          ||||
    D 0 1
|||| |||| ||||
    +⍣∘ 0 0 0
||⍝<<(>>α)+>>ω||
```

As seen in the foregoing, a single index selects a plane (the bodies and axes), two select a plane and a row, and three select a given element. Since a variable is represented by a double enclosure, the last display above shows that the dyadic definition of a function + is the (raw) double enclosure of + applied to the double disclosure of the arguments.

A monadic operator must be defined for two cases, a valence 0 argument (variable) and a valence 3 argument (function); a dyadic operator must be defined for four cases, two for each of its arguments. A monadic operator is therefore represented by an enclosed two-element vector, and a dyadic operator by an enclosed 2-by-2 matrix. For example:

```
     ¨
     ∘
|⍝α CONST ω|      ||
|⍝α AXIS ω◇ω←⊃ω|  |⍝α COMP ω|

     ⌿
||  |⍝ RED ω|

     ∇
|⍝α DEL ω|  |∘|
|∘|         |∘|
```

An example of the detailed definition of an operator may be seen in the function *DEL* used in the direct definition operator ∇ above. Thus:

```
R←A DEL W
R←>PF
R[0;0; 0 1]←A,W
R[2;0; 0 1]←(<<LOC>>A),<<LOC,>>W
R←<R
```

Briefly, the result of ∇ is the prototype function with the bodies replaced by the arguments of ∇, and with the local names (in row 0 of plane 2) replaced by the names to be localized, as determined by applying the function *LOC* to each of the arguments of ∇. For example:

```
      APL
    F←'B×B←A ←α+ω'∇'C*C←D ←÷ω'
    2 F 5
49
    A
7
    F 2
0.7071
    D
0.5
    F⍣∘0 0
||B×B←A ←α+ω||  ||C*C←D ←÷ω||  |∘|
    F⍣∘2 0
|||B|||  |||C|||  |∘|
    B
VALUE ERROR
LE[2]⍝ B
      ^
```

The details of other operators may be examined in a similar manner by displaying the supporting functions *AXIS*, *COMP*, etc. It may be noted that, although *some* of the definitions of operators must resort to functions in raw APL, some of the definitions may also be expressed in terms of operators defined only in the model. For example, the inverse operator *con* (denoted by ⊂) is defined as follows:

```
      APL
    ⊂
||  |ω⍣I⍣ω⍣(⍎I←''⊃4 1 2)|
```

C. AN AUXILIARY DEFINITION OPERATOR

An auxiliary definition operator, denoted by ⍒ and used earlier in the form F⍒∘I to display position I of the representation of a function F, is introduced for the purpose of modelling, and is not proposed as an operator to be incorporated in the language in its present form. Two further cases of it will be used in subsequent sections:

a) If F and G are functions, then F⍒G produces a function whose representation (of shape 10 2 3) is the catenation of the representations of F and G, as shown by the function ⍷11 that produces it:

```
⍷11 ◇ <(>α),[0]>ω
```

b) If I is a vector whose two elements are enclosed indices (full or abbreviated), and if H←F⍒G, then H⍒I is the function defined by replacing the element (or sub-array) of the representation of F selected by the index >I[0]. Thus F⍒G⍒(2ρ<0 1) is the function F with its axes replaced by the axes of G. If H is a simple function (whose representation has shape 5 2 3), then H⍒I is equivalent to H⍒H⍒I.

D. OPERATOR ARGUMENTS

Derived functions (resulting from the application of an operator) are represented in the general form presented in Section B; thus, the body and axes of ,⍤1 2 (ravel along axes 1 and 2) would appear as in the last two lines of the following example:

```
      APL
    F←,⍤1 2
    Q←2 3 4⍴⍳24
    F Q
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
    F⍒∘ 0
||α F ω||  ||F ω||   ||∘||
||1 2||  ||1 2||  ||1 2||
```

The definition of a derived function depends upon the arguments of the operator which produced it, as well as upon the arguments to which it is applied; the arguments of the operator are referred to in the body by the names ⍺⍺ and ⍵⍵, and are stored in locations 1 0 0 and 1 0 1, that is, in the locations denoted by *LOPARG* and *ROPARG* in the prototype function ⍷⍷. In the function F, the location 1 0 0 (that is, F⍒1 0 0) is the ravel function itself. Thus,

```
    17↑ F⍒ 1 0 0 Q
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

The following example illustrates the important fact that the arguments of an operator are bound at the time of its execution, and that subsequent reassignments to the names to which it applied do not affect the derived function produced:

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

Because of the binding at execution time, the inverse function (in location 4 1 2, and denoted by Δ*¯1 in ⍷⍷) must also appear explicitly in the representation of a function and cannot, in general, consist of a reference to the name of the inverse function. Since this inverse function must likewise contain (in its location 4 1 2) an explicit inverse function, the scheme seems to imply an infinite regression of explicit functions. However, because the inverse of the inverse is the original function, this implied difficulty can be handled in the simple manner shown in the following definition of an inverse operator:

```
    ⊂
|| |ω⍒I⍒ω⍒(⍬I←(<''),<4 1 2)|
```

The definition may be read as follows: ω⍒I produces a function in which the whole of the argument function ω (as selected by the empty first element of I) is replaced by element 4 1 2 of the same function, thus yielding the function inverse to ω. The further expression ω⍒I⍒ω therefore "catenates" the inverse of ω with ω itself, and the final application of ⍬I therefore inserts in location 4 1 2 of the inverse function ω, the original function. For example:

```
    L←* ⊂
    P←L ⊂
    L 3
1.099
    P⍒∘ 0 0 1
||⍴<<*>>ω||
```

The analogous problem of explicitly representing successive derivatives of a function does not yield to the method applied for inverses, but can be handled by using the derivative location (3 1 2) to represent a *dyadic* function whose left argument K determines the order of the index; the index K will appear as the right operator argument (in location 1 0 1, and referred to by ⍵⍵) and will be incremented on successive applications of the derivative operator.

E. SOME NEW FUNCTIONS

The convenience of the function representation employed will be illustrated by showing the formal definitions of some new functions:

```
    ⊢⍒∘0        Dex (⊢) - monadic is the identity function.
||ω||          ||ω||          |∘|
||7.237E75||  ||7.237E75||  ||7.237E75||
```

```
    ⊣⍒∘0        Lev (⊣) - monadic has no explicit result.
||α||          ||||           |∘|
||7.237E75||  ||7.237E75||  ||7.237E75||
```

```
    Z←⊢'ABC'
    Z
ABC
    Z←⊣'ABC'
SYNTAX ERROR
IS[1]⍝ NO RESULT∘
              ^
```

In the case of more complex functions, the axes of application may be seen even though the detailed definition of the body is subordinated in raw APL functions:

```
      APL
    {⍒∘0
||⍴<<(>>α)FR2>>ω||  ||⍴<<FR1>>ω||  |∘|
||¯1||              ||7.237E75||   ||¯1||
```

In what follows we will use ⊃ to denote a form of the *link* function that encloses its left argument and catenates it to the right, first enclosing the right argument if it is simple.

The foregoing function (called *from*) is defined briefly as follows: I{A is equivalent to inserting >I[J] before the Jth semicolon of an expression of the form A[;;;...]. For example:

```
    M←4 4⍴⍳16
    (2 1⊃3){M
11 7
    (<2 1){M
 8  9 10 11
 4  5  6  7
    I←3 2⍴1 0 2 1 3 0
    I
1 0
2 1
3 0
    I{M
4 9 12
```

The final case of a simple array to provide "scattered" indexing results from the definition of the left axis of application.

The monadic case of { is the cartesian product. For example:

```
    { 2 1⊃4⊃6 7
2 4 6
2 4 7

1 4 6
1 4 7
```

The relation between the monadic and dyadic cases of the function { may be seen in the definition of the function FR2.

F. SOME NEW OPERATORS

We will discuss only two operators, the first (to be denoted by }) because it is both powerful and relatively unknown, and the second (denoted by ⍤) because it motivates a number of attributes provided for in the representation of functions.

The first operator was introduced in [7] under the name *til*. It is defined as follows:

```
    }
|∘| |∘|
|∘| |'(Gω)Fα'∇'ωΔω'∇α∇(1 0 0⊃⍳0)∇ω∇(1 0 1⊃
      ⍳0)|
```

The main function is seen in the expression `'(Gω)Fα'∇'ωΔω'`; the rest simply inserts the function arguments α and ω in the "operator argument" locations 1 0 0 and 1 0 1.

The utility of *til* is discussed in [7]; the main point is that $\alpha\ F\}G\}H\ \omega \leftrightarrow (G\alpha)\ F\ (H\omega)$.

As defined in [1], the operator ⍤ applied to one function and one array produces a monadic function resulting from providing the array as one argument to the dyadic function. For example, `10⍤⍟` is the base 10 logarithm function, and `⋆⍤¯1` is the square root function. As remarked in [7], two interesting points arise:

1. Each of the monadic functions $A⍤F$ and $F⍤A$ may themselves possess inverses and derivatives; provision is made for these attributes in the locations labelled $A⍤\Delta\star^{-}1$, $\Delta⍤A\star^{-}1$, $A⍤\Delta$, and $\Delta⍤A$ in the prototype function PF.

2. Because a derived function is ambivalent, provision is made (in locations 3 0 0 and 3 0 1 of the representation of a function F) for representing the dyadic cases of the functions $A⍤F$ and $F⍤A$. The dyadic case of the selection function $I⍤\{$ is particularly important, being defined as follows:

   The result of $B\ I⍤\{A$ is the array A with B merged into the portion selected by I. This function obviates indexed assignment. In order to obtain the effect of indexed assignment of A, one would write $A\leftarrow B\ I⍤\{A$. Other dyadic selection functions may be treated analogously.

G. INDIRECT ADDRESSING

In the normal execution of an APL expression, each of the vector of tokens of the expression (placed in the left stack L in the model) is "evaluated" and the result of the evaluation is transferred to the stack of intermediate results (R in the model). However, a token which immediately precedes an assignment arrow must be exempted from this rule, and must be transferred "without evaluation". For example:

```
    A←'ABC'
    ⌽A
CBA
    A←5
    A
5
    ABC
VALUE ERROR
```

Parentheses, however, imply that the enclosed expression is to be evaluated, and the result transferred to the stack of intermediate results. In an expression such as (A)←5 this rule conflicts with the stated rule for assignment and, in conventional APL, such an expression is treated as a syntax error.

The conflict can be resolved by prescribing an order for the application of the two rules. In the present model, the rule for parentheses is applied first, with the obvious and convenient consequences illustrated by the following sequence:

```
      APL
    A←'ABC'
    (A)←5
    A
ABC
    ABC
5
```

Since the result of evaluating an APL expression may be an array of enclosed names, the notion can be extended as shown by the following example:

```
    N←<⍤⍳>'ABCD'
    N
|A| |B| |C| |D|
    M←4 3⍴⍳12
    (N)←M
    A
0 1 2
    D
9 10 11
    ⎕→
```

The detailed definition of this *indirect assignment* may be seen in the function IS:

```
W←A IS W;X;N1;B1
⍎(NORE=W)/'NO RESULT∘'
→(1∈A∈<⍤⍳>A)/L0
→0,0⍴A IS1 W
L0:A←(,A),[0.5],(>>W) NUC1(⍴⍴A)+⍳⍴⍴>>W
L1:→(0=1↑⍴A)/0
X←A[0;]
A← 1 0 ↓A
→(1∈N1∈<⍤⍳>N1↔>X[0])/L2
→L1,0⍴N1 IS1 X[1]
L2:→L1,0⍴N1 IS X[1]
```

The auxiliary function *IS*1 is simple assignment, except for the fact that it handles assignments to graphic symbols as well as to names that are legitimate in raw APL. The function *NUC*1 encloses the "nuclei" determined by the axes specified by its right argument.

Some interesting consequences of the definition of *IS* are illustrated by the following sequence in which *J* is assumed to be predefined (as shown):

```
        APL
      B←<4 5 6
      B
|4 5 6|
      (<'C')←B
      C
4 5 6
      ⍝⎕PS←-2/1 3
      J
|¯¯¯¯¯¯¯¯¯¯¯| |¯¯¯|
||¯¯¯|  |¯|| |BCD|
||ABC|  |B|| |___|
||___|  |_||
|           |
||¯|    |¯||
||C|    |D||
||_|    |_||
|___________|

      (J)←J
      ABC
ABC
      D
D
```

Since expressions of the form used for indirect addressing proposed here are invalid in conventional APL, their introduction would produce no conflict. Their use would, however, conflict with a different proposed use of parentheses to the left of assignment to extend the use of indexed assignment to selection functions other than "bracket" indexing [3]. It should be noted that the dyadic "merge" function *I*¨{ discussed in Section F illustrates a general scheme for using the operator ¨ together with any selection function to provide the effect of indexed assignment. It should also be noted that the explicit result of the expression *B I*¨} *A* is the entire merged entity, whereas the explicit result of *A*[*I*]←*B* (or of corresponding extensions to other functions) is simply *B*.

H. INDEX ORIGIN

A number of people (among whom Professor Penfield is perhaps the most persuasive) have long maintained that any benefits provided by the choice of index origin in APL are outweighed by the burden of controlling its effects. It is, of course, futile to propose that the present use of index origin be changed in any way; however, in the design of any new functions and operators one may choose to exclude dependence upon index origin, just as the choice was made in the design of APL\360 [8] to exclude dependence on index origin in the definition of the residue function, even though the earlier definition in [9] included it.

Problems due to index origin appear to be magnified in the case of operators. For example, in *G*←*F*∘*I*, are the axes used in the application of *G* to depend upon the origin in effect at the time of specifying *G* or at the time of applying *G*? Or should it perhaps depend upon the index origins localized within the definitions of *F* and *G* as well?

In any case, the present proposal is to adopt a fixed index origin for all new functions and operators and to make this origin *zero.*

I. IDIOSYNCRACIES OF THE MODEL

For practical reasons the model has not been made as general as it could be, and any person using or modifying it should perhaps be aware of some of the limitations and peculiarities, and some of the reasons for them. Thus:

1. Except for the name *APL*, the names used within the model all incorporate underscored letters or digits; the names that a user may safely employ should be formed from the simple alphabet only.

2. Some of the definitions of functions and operators are couched in expressions in raw APL, some in the extended APL provided by the model, and some in a mixture of the two. The choice of one or the other is rather arbitrary, except for the application of the following criteria:

   a) Some of the underlying functions *had* to be expressed in raw APL in order to obtain a working model.

   b) Illustrations of both uses were included as guides for anyone attempting to add further definitions.

   c) Use of raw APL leads to more efficient execution of the model.

   d) Use of the extended functions was very helpful in exercising the model and ensuring its correct behaviour.

3. The main criteria applied in the design of the model were clarity and flexibility; increased efficiency can, if required, be attained by rewriting a number of the auxiliary functions.

4. The definitions of the primitive functions provided in the model are incomplete in the sense that many of the meaningful attributes are left unspecified. However, the discussion and the examples (such as the inverse specified for the function ⋆) should provide sufficient guidance for completing the definitions as desired.

5. The prototype function *PF* shown in Section B shows some attributes which have not been discussed. They should be considered as tentative.

For example, positions `1 0` show the argument names used, and an operator for changing them would allow a choice of the argument names to be used in the direct definitions. Similarly, positions `2 0 0` and `2 0 1` may be used to directly specify the names local to the dyadic and monadic cases.

Positions `2 1` provide for the specification of identity functions (as a generalization of identity elements) for the monadic function itself (Δ) and for each of the derived monadic cases *A*¨Δ and Δ¨*A*. Positions `4 0 0` and `4 0 1` provide for possible inclusion of *variants* of the type discussed in [10].

In specifying the inverse of any function it should be remembered that the specification is *formal* in the sense that it merely determines the function that results from the application of the inverse operator; the function may in fact be only a partial inverse (as in ¯1○ω and 1○ω) or it could even be a function that is not inverse at all. Similar remarks apply to derivatives.

ACKNOWLEDGEMENTS

I am indebted to a number of my colleagues at I.P. Sharp Associates, particularly to Arthur Whitney for material adapted from his earlier model, and to Roland Pesch for comments arising from his use of the model.

REFERENCES

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

2. Carl Cheney, APL*PLUS Nested Arrays Reference Manual, STSC Corporation, 1981.

3. *APL Language Manual*, Form number SB21-3015, IBM Corporation, 1982.

4. P.K. Wooster, "Improved Display for Enclosed Arrays and a New System Variable □*PS*", Technical Supplement 37, *I.P. Sharp Newsletter*, Vol. 10, Number 2, 1982

5. Third Working Draft of the International Standard for the Programming Language APL, International Standards Organization, ISO TC97/SC5/WG6-N28, 1982.

6. K.E. Iverson and P.K. Wooster, "A Function Definition Operator", APL Quote Quad, Vol. 12 Number 1, ACM 1981.

7. K.E. Iverson and A.T. Whitney, "Practical Uses of a Model of APL", *APL Quote Quad*, Volume 13, Number 1, ACM September 1982.

8. A.D. Falkoff and K.E. Iverson, *APL\360*, IBM Corp., November 1966.

9. K.E. Iverson, *A Programming Language*, Wiley, 1962.

10. K.E. Iverson, *Operators and Functions*, Research Report RC 7091, Research Division, IBM Corp., 1978.

APPENDIX

See the body of the paper for functions *APL*, *S*, *DEL*, *D*11, and *IS*, for variables *ST*, *ACT*, and *PF*, for operators∘., +, and ∇, and for descriptions of variables *P* and *T*0.

VARIABLES

```
      M
Z1←DE DEX Q;SC;SG;FS;J;Z;AR;□TRAP
□TRAP←'∇ 8 C →L0',0ρZ←>DE[2]
⍎(>>''ρΦAR←>DE[1]),'←''''ρΦQ',(ρZ) SLB Z
⍎(1≠ρQ)/(>>''ρAR),'←''''ρQ'
SC←¯1Φ⍳ρSG←>''ρDE
L0:→(0=ρSC)/L4
SC←1↓SC,0ρFS←>SG[''ρSC]
→(∧/' '=FS)/L0
→('ρ'=1↑FS←(J←'→'=1↑FS)↓FS)/L1
→J↓L2,L3,0ρZ←(TK FS) S ''
L1:→J↓L2,L3,0ρZ←⍎1↓FS
L2:→L0,0ρZ1←Z
L3:→L0,SC←,>>Z
L4:□TRAP←'∇ 6 C →0,0ρZ1←NORE'
Z1←Z1

      CN
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFG
HIJKLMNOPQRSTUVWXYZ0123456789¯.A
```

FUNCTIONS

```
AC◇(1↑(∧/(0>Z)≠(|Z)=(ρZ←0 ¯1↓ST)ρX)/ST[;4]
  ),X←4↑(,ρ⍤0>ρ⍤1>ω),4ρ6

Z←F AXIS I;J
Z←>''ρPF,0ρJ←(<<'αFω'),(<<'Fω'),(<<'∘')
Z[0;;]← 2 3 ρJ,<⍤1>3ρ>>I
Z[1;0;0]←F
Z←<Z

CF◇0≠ρω,0ρ□PS←1◇''◇(⍎(α/'<,∇'),',>1↑W'),(~
  α)CF 1↓ω

CQ◇(1↑L)CF(L←'∧/□VI ω'QNAω)ENCω←ω

Z←A COMP B;Q;J
Q←>''ρB,0ρZ←>PF
Q[0;1;]←3ρ<⌊/0ρJ←(<<'(Gα)F Gω'),(<<'F Gω')
Z[1;0; 0 1]←A,<Q
Z[0;;]← 2 3 ρJ,(<<'∘'),(>B)[0;1;]
Z←<Z

Z←A D10 B;I;J;P;SA;SP;U
→(('∘'∈B),~1∈B∈<⍤0>B←>>B)/L4,L5,0ρSA←ρA←>A
I←SP EIX>''ρB,0ρU←,(-SP←ρ>PF)↑A
→((1∈J∈<⍤0>J←>''ρΦB),∨/SA≠SP)/L2,L0
L2:U←,(-SP)↑A←A,[0] A
L0:P←,(-SP)↑(-1↑SP)Φ[0] A
J←U[J←SP EIX J]
⍎1Φ'JJ←',1↑(x(x/ρI)-x/ρJ)Φ' ><'
P[I]←J
→0,0ρZ←<((- 2 0 0 ×SP)↑A),[0] SPρP
L4:Z←>PF
Z[0;0;1]←<<'ω{F'
Z[1;0;0]←<<A
→0,0ρZ←<Z
L5:Z←(,A)[(ρ>PF)⊥¯3↑B]

EIX◇QPP(Φ1,x\Φ1↓α)x⍤1>(<⍤0>ω),(ρ,ω)↓⍳⍤1>α

ENC◇(0≠ρα)◇''◇(<K↑ω),(K↓α)ENC(K←1⌈+/∧\(α≤1
  )∧α=1↑α)↓ω

FIX X;□PS;H;S
X←⍉>,<(0≠,ρ⍤0>,⍤1>X)/X←(>X[0]),(>X[1]),>X[2]
S←(1↓ρM)⌈ρH←(M[0;]),,∇(,(<';'),X),0ρ□PS←1
0 0 ρ□FX(S↑H),[0] 1 0 ↓((1↑ρM),S)↑M

FR1◇⍉X⍉QPPωx⍤1>1↓Φ1,x\ΦX←1+MAXω


FR2◇(,ω)[⍉(ρω)⊥⍉FR1(<⍤0>α)SUBω]

IQ◇Z∨≠\Z←ω=''''

W←A IS1 W
→(∨/,A∈'αωΔ',56↑,CN)/L0
→0,0ρP←((<A),W),[0](A≠,>P[;0])/P
L0: 0 0 ρ⍎A,'←W'

Z←L LB A;X;U;□PS
X←,⍤1>>⍤1↑⍤1>A←'<TAω' QNA SEG A,'◇',0ρ□PS←1
U←(0≠1⊥⍤0>X∈⍤1><52↑,CN)∧0=1⊥⍤0>~⍤1>X∈⍤1><62↑,CN
X←X,[0.5]<0ρU←U∧(<,':')≡⍤0>,⍤1>>⍤1↑⍤1↓⍤1>A
Z←(<,⍤1>∇⍤1>(2×U)↓⍤1>A),L,<, 0 1 ↓UΦX
```

```
Z←X LE Y;W;J;K
→((∧/')'=>W),Y,∧/,(W←''ρ(ΦX),<,' ')∈P[;0])
  /L0+ι3
→0,0ρZ←(<¯1↓X),±(2×1∈W∈<⍥>W)↓'<<W',0ρW←±>W
L0:→0,0ρZ←(<J/X),(¯1↓1↓X) S±''''''',('('≠>'
  'ρK←(~J←ΦNΦX)/X)/'UNMATED()'
→0,0ρZ←(<¯1↓X),<''ρ¯1↑X
→0,0ρZ←(<¯1↓X),(P[;1])[P[;0]ιW]

LOC◇((1ΦI∈<,'←')∧~I∈(<,'□'),<,' ')/I←TA ω

MAX◇0≠ρ,ω◇''◇(⌈/0,,>''ρω),MAX 1↓ω

N◇(~<\J)∧J←∨\0=+\-⍀'()'∘.=,1↑⍥>,¨>ω

NUC1◇(L/ι0)=1↑ω,X←(ω∈(0,-Y)∘.+ιY)/ω,0ρY←ρρ
  α◇α N2 Y|X◇α N2 (~(ιY)∈Y|X)/ιY

N2◇OAρ(<IA)ρ¨>X←I↑¨>((ιx/OA←N↓ρα)×I←x/IA←(
  N←-ρ,ω)↑ρα)↓¨><,α←(⍋((~B∈ω)/B←ιρρα),ω)⌽α

Z←α ONA ω
→(0≠1↑,0≠ρω)/3
→0,0ρZ←''
Z←(±α,0ρω←>''ρω),α ONA 1↓ω

OPP◇0≠ρ,ω◇0◇(>''ρω)∘.+OPP 1↓ω


Z2←Δ RE Y;F;G;J;A1;A2;NO;SH;Z1;DE;DEX;SAL
±(NORE←Y[0])/'NO RESULT∘'
→(∨/ 0 3 ∈ρ⍥>ρ¨>Δ)/L0,Z2←0ρρρJ←''ρ2←ρY
FIX(1↓SAL←(<(~J)↓<⍥>'αω') LB DE),<LOC DE←,
  >(,>Δ)[2⊥0≠ρ⍥>ρ¨>Y]
→0,0ρZ2←SAL DEX,Y
L0:G←''ρ1↓G,0ρF←''ρG←(>Δ)[1;0; 0 1]
FIX(1↓SAL←(<>¨>(>Δ)[1;1;(J= 1 1 0)/ι3]) LB
  >>(>Δ)[0;0;~J]),>(>Δ)[2;0;~J]
Z2←0ρNO←x/SH←ρA2←(>>Y[J]) NUC1>>(>Δ)[0;1;2
  -J]
A2←,A2
→(~J)/A1←L1
A1←(>>Y[0]) NUC1>>(>Δ)[0;1;0]
±(~∨/(SH=ρA1),(0=ρρA1),0=ρSH)/'~CNFRMABL∘'
NO←x/SH←,>((<SH),<ρA1)[(ρρA1)>ρSH]
A1←,A1
L1:→(0>NO←NO-1)/L2
Z1←SAL DEX(J/<''ρA1),<''ρA2
→L1,0ρA2←1ΦA2,0ρA1←1ΦA1,0ρZ2←Z2,Z1
L2:Z2←<<>>SHρZ2

Z←RED X;Y
Z←>''ρPF,0ρY←'G⍥2 1 2ω◇Z←>(E←ι0)ρA◇→Sρ3'
Y←Y,'◇Z←(>EρA←1↓A)G Z◇→,>(¯1*0>S←¯1↓ρ,A'
Z[0;0;1]←<Y←Y,'←Φ<⍥¯ 0 ω)↑1 2⊃0'
Z[1 2 ;0;1]←X,<<LOC Y
Z←<Z

SEG◇1≠ρω,'◇'◇''◇(<K↑ω),SEG(1+K←+/∧\(ω≠'◇')
  ∨≠\ω='''')↓ω

SLB◇0=ρω◇αSLB ¯1↓ω,0ρ±(0≠ρZ)↓'Z',(Z←,>''ρΦ
  ω),'←(¯1↓ρW)↓ιA'◇''

SU◇((ω≡⍥>><,'¯')Φ(<'9E99'),((ρω),1)ρω)[;1]

SUB◇X|¨>(ρω)S1 Y↑α,(Y←ρX←ρω)ρ<L/ι0

S1◇0≠ρ,ω◇''◇(<(>''ρα)S2>''ρω),(1↓α)S1 1↓ω

S2◇(1↑ω)=L/ι0◇ω◇(~X∈α|1↓ω)/X←ια

TK◇CQ SU'((ω='').'')∧(~IQω)∧~(1⌊ρ□VIω)↑□VIω
  )ENCω'ONA TAω←,ω

TA◇'(2×~(IQ ω)∨ω∈CN)ENC ω'ONA(IQω)ENCω


R←A TR R;K;L;Z
→(0=K←|L←>>T0)/0
Z←(¯2↑1,ρZ)ρZ←⍒(T←, 0 1 ↓M←(Sε⍥>PF)Φ(<1,(
  2⌊K),2),[0.5] S←ρ⍥>R)↑⍥>R
((0>L)∨' '≠1↑A)/((K,ρ,A)ρA),((K←K⌊1↑ρZ),¯1
  ↑ρZ)↑Z

Z←A UPON B;Q
Z←>PF
Q←(B←>B)[0;1;]
B[0;1;]←3ρ<<⌊/ι0
Z[1;0; 0 1]←A,<B
Z[0;;]← 2 3 ρ(<'FαGω'),(<'F Gω'),(<<'∘'),Q
Z←<Z
```

OPERATORS

```
      ⍒
||        |AαD01ω|
|AαD10ω| |AαD11ω|

      ⍥
|∘| |∘|
|∘| |AαUPONω|
```