AN OPERATOR CALCULUS

Kenneth E. Iverson
I.P. Sharp Associates
Suite 1900, 2 First Canadian Place
Toronto, Ontario
Canada M5X 1E3

Roland Pesch
I.P. Sharp Associates
Suite 201, 220 California Ave.
Palo Alto, California
U.S.A. 94306-1638

J. Henri Schueler
I.P. Sharp Associates
Suite 1900, 2 First Canadian Place
Toronto, Ontario
Canada M5X 1E3

This paper extends a line of APL development presented in a sequence of papers [1-7] over the past six years. The main topics addressed are the interactions of operators such as rank, composition, derivative, and inverse (i.e., the beginnings of a calculus of operators), a simplification in the complement of attributes tentatively presented in [6], and a treatment of the shapes of individual results (as defined in [7]) in the case of empty frames.

Brief treatments are also given to a number of smaller matters: a *transliteration* or *token substitution* facility, the treatment of niladic functions, a custom (variant) operator, the obsolescence of certain system variables, and some changes in the function definition operator and in the treatment of supernumerary axes.

THE STRUCTURE OF FUNCTIONS

The term "attribute", used in earlier papers, led to some misunderstanding because it improperly suggested two conclusions:

1) That a function, like most entities, is something more than a collection of its attributes.

2) That "attribute" means the same as "property" in discussing mathematical functions. An attribute merely determines what the result of a particular operator is. To be useful, this attribute will approximate some mathematical property, but in principle it need not.

For example, the inverse operator applied to the sine function (1¨○) will yield the arcsine (¯1¨○), although these functions are proper inverses only over a sub-domain of "principal" values, and even on that sub-domain there any implementation provides only an approximation. Moreover, such approximations have long been incorporated in APL, as in the result 0 given for the identity element of < (resulting from </⍳0) although it is a left identity only, and only over the boolean sub-domain.

We will therefore adopt the more neutral term "part" instead of "attribute".

A function comprises one or more parts: a *core*, and zero or more ancillary parts that extend the domain of, or otherwise modify, the function defined by the core.

For example, if the core of *f* is the function ⌹ as defined in APL\360, and if *f* possesses no other parts, then the right domain of *f* includes arguments of rank 2, but none of higher rank. However, if *f* also contains a *rank* part that specifies how an argument of higher rank is to be split into cells of rank 2 for application of the core function, then the domain of *f* is extended to right arguments of higher rank. Finally, the domain of the *derived* function *f⌿* will include an array of shape 3 4 4 but will exclude one of shape 0 4 4; the addition of an *identity function* part to *f* will extend the domain of *f⌿* to include arguments with a shape such as 0 4 4 in which the split along the leading axis induced by the reduction produces an empty collection of slices.

The core itself has five parts, two *kernels* (monadic and dyadic), and three *parameters*, referred to in the APL expressions that comprise the kernels by the distinguished names ⎕*A*, ⎕*B*, and ⎕*C*. The first two of these were formerly referred to as underscored *F* and *G* [5 6], and are specified by the arguments of most operators.

For example, if *c* is the composition [3] of functions *a* and *b* (that is, *c*←*a*⍤*b*), then the kernels of *c* are the vectors '⎕*A* ⎕*B*⍤¯ ⍵' and '(⎕*B*⍤¯ ⍺) ⎕*A* ⎕*B*⍤¯ ⍵', and the parameters ⎕*A* and ⎕*B* are the functions *a* and *b*. In the case of the definition operator [4], the kernels of the function *d*←'÷⍵'∇'⍺+÷⍵' are '÷⍵' and '⍺+÷⍵', and the (unused) parameters ⎕*A* and ⎕*B* have the same values '÷⍵' and '⍺+÷⍵'.

The third parameter (⎕*C*) is the *custom* or *variant* parameter that can be used to provide variants of a function in the sense introduced in [1]. It is respecified by the right argument of the variant operator (assumed here to be the function-variable case of the dot, as in ⍋. 0 for 0-origin grade, and =.*tol* for equality comparison with a specific tolerance *tol*).

For example, if the function *sin* has a monadic kernel '1○⍵×○÷2×⎕*C*' and ⎕*C* set to 0.5, then *sin* ⍵ yields the sine of an argument expressed in radians, *sin*. 90 ⍵ yields the sine of an argument in degrees, and *sin*.(0.5) is equivalent to *sin*.

In principle, a function may incorporate any number of ancillary parts, but the present treatment is limited to seven: rank, coherence, shape surrogate, inverse, dyad, derivative, and identity.

If a part referred to by some operator is not present, the operator does not produce a domain

© 1984 ACM 0-89791-137-7/84/006/0213 $00.75

error, but does produce a derived function with a restricted (perhaps empty) domain. For example, the "dual" operator ¨ in the expression *h*←⌽¨| does not produce a domain error, but any subsequent application of *h* does, since the operator requires the inverse of its right argument [3], and the magnitude function does not possess an inverse part.

Absence of the rank part is equivalent to the presence of an infinite rank, and the same is true of coherence.

RANK

The rank part is a three-element vector whose elements limit the ranks of the argument cells presented to the function defined by the core; the successive elements apply to the monadic, left dyadic, and right dyadic arguments.

The *rank operator* is the function-variable case of ⍤, and *f*⍤*k* is equivalent to *f* with its rank part specified by ⌽3⍴⌽*k*.

COHERENCE

The following definition is adapted from [7] with the arguments of the corresponding operator reversed and with the term "coherence" used instead of "conformance".

In the application of a dyadic function *f*, the outer shapes *ol* and *or* are each split into two sets of axes (called *bound* and *free*) such that *ol*≡*bl*,*fl* and *or*≡*br*,*fr*; the shape of the overall result is *b*,*fl*,*fr*,*sir*, where *b* is one of *bl* and *br*, and *sir* is the shape of the individual results of applying the function to its cells.

A shape *s* is said to be *single* if 1=×/*s*; if *one* of *bl* and *br* is single, then *b* equals the other; if both are, then *b* equals the one of greater length; if neither is single, then *bl* and *br* must agree, and *b* is chosen as either one.

The lengths of *bl* and *br* (that determine the number of bound axes) are each limited by the *coherence part* of a function; all primitive functions have infinite coherence. The *coherence operator* (denoted by *k*.*f*, where *k* is a non-negative integer), produces a derived function equivalent to *f*, but having coherence *k*. For example:

```
      ⍴(a←2 3 4⍴⍳24)0 .×(b←2 3 5⍴⍳30)
2 3 4 2 3 5
      ⍴a 1 .× b
2 3 4 3 5
      ⍴a 2 .× b
2 3 4 5
      ⍴a 3 .× b
length error
      ⍴a 1 .×(1 4⍴9)
2 3 4 4
      ⍴(1 1 1⍴8)+(1 1⍴9)
1 1 1
      ⍴2 3+1 1 1 1⍴4
2
```

The case of zero coherence (0 .*f*) is equivalent to outer product (∘.*f*). The reason for reversing the arguments of the earlier conformance operator [7] is to leave the case *f*.*k* free for the variant operator, with no inhibition on the use of *k*←<'' or *k*←∘ that would have been required in using the form *k*.*f*.

SURROGATE ARGUMENTS

In applying a monadic function *f* of (non-negative) rank *k* to an argument *a*, the shape of the overall result is *os*←(-*k*)↓⍴*a* suffixed by *sir*, the (necessarily common) shape of the individual results obtained in applying *f* to each of the ×/*os* cells of shape *cs*←(-*k*⌊⍴⍴*a*)↑⍴*a*.

If there are no cells (i.e., 0=×/*os*), the value of *sir* cannot be determined by applying *f*, and must be determined from the cell shape *cs* alone. It will be defined as the shape of the value of lowest rank and smallest shape that could be produced by applying *f* to any argument of shape *cs*. For example:

```
  f:   ⍴⍤k     ,⍤k     <⍤k    ⍉⍤k    ⍕⍤k

sir:   ⍴cs     ×/cs    ⍳0     ⌽cs    cs
```

The dyadic case is treated similarly (in terms of the left cell shape *ls* and the right cell shape *rs*), but is complicated by the cases of "scalar extension", that is, a left cell *value lv* will be present if the left outer shape is *single* (i.e., the product over it is unity), and a right value *rv* will be present if the right outer shape is single.

For example, if *f* is the dyadic function ⍴⍤(1,*k*) and a left value *lv* is present, then *sir* is *lv*; if only *ls* is available, the value of *sir* is *ls*⍴0, since the value that serves as *lv* must be *ls*⍴0, that is, something of shape *ls* that produces a result of minimum shape.

The entire situation can be handled by providing *surrogate argument functions* that, in each case of an empty frame, apply to the cell shape and function argument to produce a surrogate argument (whose shape equals the cell shape). This surrogate is submitted to the original function to produce a result whose shape properly determines the "individual result shape" required. Table 1 specifies surrogate argument functions for existing primitives.

As examples of the use of Table 1, consider the following cases:

```
      ⍴ ⍉⍤3(0 1 2 3 4⍴0)
0 1 4 3 2
      ⍴ 1 0 2⍉⍤1 3(0 1 2 3 4⍴0)
0 1 3 2 4
      ⍴ (0 1 3⍴0)⍉⍤1 3(2 3 4⍴0)
0 1
      ⍴ 2↑⍤1(0 3⍴0)
0 2
      ⍴ (0 2⍴0)⍴0
0 0 0
      ⍴ 1 2⍴⍤1(0 2⍴0)
0 1 2
      ⍴ 1 2⍴⍤1(0 0⍴0)
length error
```

INVERSE

The result of the inverse operator ⊂ applied to *f* is the inverse part of *f*, except that *its* inverse part is respecified as *f*.

DERIVATIVE

Consider a dyadic operator *DOP* such that *f DOP k* yields the *k*th derivative of *f* if the scalar *k*≥0, and the (|*k*)th *integral* if *k*<0. In order to provide all derivatives, the derivative part of *f* must have two components, a dyadic function *df* and an index *k* such that *k df ω* yields the *k*th derivative of *f* evaluated at *ω*.

More generally, we will assume that *df* is defined to apply to a *vector* index *v* that specifies successive derivatives. For example, 2 ¯3 4 *df ω* yields the second derivative of the third integral of the fourth derivative of *f*.

The derivative part of *f*, and the action of the derivative operator, may now be defined as follows. The derivative part of *f* has the monadic kernel '⎕*B* ⎕*A ω*', a null dyadic kernel, and the parameters *df* and *v*. Both the core and the derivative part of *f DOP k* become the derivative part of *f* with the parameter ⎕*B* replaced by *k*,⎕*B*. Moreover, the dyads of *f DOP k* become null (because the derivative has no dyadic definition), but all remaining attributes are inherited from *f*.

As pointed out in [2], the derivative of a function *f* having a "result rank" of *r* and an argument rank *a* must have a result rank of *r*+*a*. This behaviour must be incorporated in the definition of the function *df* referred to in the preceding paragraphs. Moreover, integration must produce an indefinite integral, and *f DOP* (-*k*) therefore incorporates a supernumerary axis of length 1+*k* such that (*c*,1)+.×*f DOP* (-*k*)*ω* yields the value for any specified constants of integration *c*.

IDENTITY FUNCTION

In extending the notion of an identity element, first introduced to give meaning to expressions such as +/⍳0 and ∧/⍳0, it is clear that the result for a non-scalar function must depend upon the shape of the cell to which it is applied. For example, in +.×⌿0 4 4ρ?9, the identity element must be the 4 by 4 identity matrix (⍳4)∘.=⍳4.

The notion of an identity element must therefore be replaced by adding a new part that is an identity *function* (that applies to the cell shape), and by adding an operator that assigns a value to the part.

DYADS

Monadic functions such as 10¨⍟ (the base-10 logarithm) and *¨2 (the square function) might appear to be fully defined by the arguments of the operator ¨, and therefore require no special part in the function to which the operator is applied. However, each such function may have parts such as derivative and inverse, and even a dyadic case.

For example, the inverse and derivative of *a*¨+ are (-*a*)¨+ and the constant function 1, respectively, and if *f* is a selection function, then *i*¨*f* can have a dyadic case that provides a *merge* of its arguments, as described in [5 6] for the case where *f* is the indexing function *from* ({).

We define the monadic cases of the derived functions *a*¨*f* and *f*¨*b* as follows:

```
f¨b a ↔ a 0 .f b

a¨f b ↔ b 0 .f⊂ a
```

where ⊂ denotes the commute operator [7]. Moreover, the dyads each inherit the appropriate dyadic rank of *f*, as well as the appropriate dyadic surrogate.

Any inverse, derivative, or dyadic case of the derived functions *a*¨*f* and *f*¨*b* are determined from the information provided in the dyad parts of *f*.

CALCULUS OF OPERATORS

In the case of an operator such as inverse, the entire derived function is determined by the inverse part of the original function, and there is no question of parts of the derived function being determined from any other parts of the original. However, in the case of an operator such as composition, it is clear that certain parts of the derived function should be inherited from (or at least derived from) various parts of the original functions.

For example, if *h*←*f*⍤*g*, then the inverse part of *h* should have the monadic kernel '⎕*B*⊂⍤(⎕*A*⊂) *ω*', and parameters *f* and *g*.

The cases of the rank and coherence operators are the most interesting, since a number of parts of the function argument might be usefully passed on to the derived function with little or no change.

The effect that the coherence operator should have on the parts of the derived function is rather straightforward, but that of the rank operator is more problematical. For example, the proper inverse of ⍉⍤*r* is clearly ⍉⍤*r*, but since the inverse of the enclose (<), of infinite rank, is the disclose (>) of rank 0, what rank should be assigned to the inverse <⍤3⊂? Moreover, in the dual *f*¨*g*, the inverse of *g* is applied to an argument of whatever rank is produced by applying *f* to the result produced by *g* on one of its cells. What then should be the rank of the inverse *h*←*g*⍤*r*⊂ to apply properly in *f*¨*h*?

Composition. The rank part of the function *f*⍤*g* is the monadic rank of *g*, giving "close composition" as defined in [3]; the kernels are '⎕*A* ⎕*B*⍤¯ *ω*' and '(⎕*B*⍤¯ α)⎕*A* ⎕*B*⍤¯ *ω*'.

The reason for using ⎕*B* of infinite rank (that is, ⎕*B*⍤¯) rather than ⎕*B* in the kernel is illustrated by the following example. If *f*←⌽ and *g*←⍉⍤¯1, and *a*←2 3 4 5ρ⍳120 then (because the rank of *f*⍤*g* is ¯1), the cells of *f*⍤*g a* have shape 3 4 5, and the application of *g* with infinite rank would transpose each of them to shape 5 4 3 before applying *f*. However, if *g* itself were applied, it, being of rank ¯1, would transpose each of the 4 by 5 cells of each shape 3 4 5 cell presented to it, providing arguments of shape 3 5 4 to *f*. In effect, the proposed definition prevents a double application of the rank of *g*.

Operators related to composition. The kernels and ranks of the derived functions of three other operators show marked similarity to those produced by composition (⍤).

```
            Ranks and Kernels

f⍤g    mg                  lg   rg
       ⎕A ⎕B⍤¯ ω           ⎕A α ⎕B⍤¯ ω

f¨g    mg                  mg      mg
       ⎕B⊂ ⎕A ⎕B⍤¯ ω       ⎕B⊂(⎕B⍤¯ α)⎕A ⎕B⍤¯ ω

f}g    mg                  mg      rf
       (⎕B⍤¯ ω)⎕Aω         (⎕B⍤¯ ω)⎕A⍤¯ α
```

Intrinsic rank. Although ⊢ is the identity function and is of unbounded rank, the composed function *g*←*f*⍤⊢ may differ from *f*, the difference becoming apparent only when the rank operator is applied to the functions.

For example, if *f*←⍉⍤2 (where ⍉ itself is of infinite rank), then *f*⍤3 is equivalent to ⍉⍤3, but *g*⍤3 is equivalent to ⍉⍤2. We will therefore say that *h*←*g*⍤3 has *extrinsic* rank 3, but *intrinsic* rank 2.

More generally, if *p*←*q*⍤*j*⍤⊢⍤*k*, then *p* is said to have intrinsic rank *j* and extrinsic rank *k*; the extrinsic rank is immediately respecified by application of the rank operator, but the intrinsic rank is unaffected.

Every primitive function will be defined to have an intrinsic rank that is equal to its extrinsic rank, and is non-negative.

The results of rank. Except for the derivative, dyads, and inverse parts, all parts of *f* (including, in particular, its coherence) are inherited by *f*⍤*r*.

Since a derivative of a function *f* must apply to the same cells as *f*, the rank must be inherited by the derivative.

The dyad parts are not inherited by *f*⍤*r*, but the monadic cases of *a*¨(*f*⍤*r*) and *f*⍤*r*¨*b* are, of course, defined as stated earlier.

As remarked earlier, there is no relation between the rank of a function and the rank of its inverse that applies for all functions. However, in the case of a rank-preserving function *f*, the inverse function *f*⊂ *would* be expected to have the same rank, and we propose to choose the treatment of *f*⍤*r*⊂ to provide behaviour appropriate to such a function. Moreover, appropriate behaviour of the dual *g*¨(*f*⍤*r*) can be expected only in the case where *g* is also rank-preserving.

A necessary condition that *f*⊂ be a proper inverse is that *f*⊂⍤*f* be the identity function; it is also desirable that *f*⍤(*f*⊂) be an identity. The problems of defining the inverse appropriate to a function *f*⍤*r* will first be illustrated by the function *t*⍤*r*, where *t* is a self-inverse transpose of intrinsic rank 3 (i.e., *t* ↔ *t*⊂ ↔ ⍉⍤3⍤⊢), and the argument *a* has shape 2 3 4 5 6 7 8.

We will examine two main cases, the direct inheritance of *t*⊂ by *t*⍤*r*, and the modified inheritance of *t*⊂⍤*r*. Within each of these we will examine the cases of rank restriction and expansion.

Case 1: *t*⍤*r*⊂ ↔ *t*⊂

r	⍴*t*⍤*r*⊂⍤(*t*⍤*r*) *a*	⍴*t*⍤*r*⍤(*t*⍤*r*⊂) *a*
2	2 3 4 5 6 7 8	2 3 4 5 8 7 6
¯3	2 3 4 5 6 7 8	2 3 4 6 7 8 5
4	2 3 4 5 6 7 8	2 3 4 6 7 8 5

Case 2: *t*⍤*r*⊂ ↔ *t*⊂⍤*r*

2	2 3 4 5 6 7 8	2 3 4 5 6 7 8
¯3	2 3 4 5 8 7 6	2 3 4 5 8 7 6
4	2 3 4 5 6 7 8	2 3 4 5 6 7 8

From the foregoing it is clear that only case 1 (direct inheritance) gives correct behaviour for all sub-cases for the more important left-inverse (*t*⍤*r*⊂⍤(*t*⍤*r*)) and that neither case can give correct behaviour for all sub-cases of the right-inverse. We therefore propose adoption of the rule of direct adoption of the inverse *f*⊂ for the inverse of the derived function *f*⍤*r*.

The results of coherence. Since coherence determines only which pairs of cells of the two arguments are submitted to the function, *all* parts of *f* save the coherence are passed on to the derived function *k*.*f*.

SUPERNUMERARY AXES

As remarked in [7], certain operators introduce one or more *supernumerary* axes in addition to the axes produced by the particular function to which the operator is applied. Although such supernumerary axes should *precede* the normal axes, the cited paper proposed an exception for the case of *f*\ (scan along the last axis) as a means of maintaining compatibility with the present behaviour of scan for primitive scalar functions.

A more palatable way of retaining compatibility is to assign rank 1 to the derived function *f*\ (and also to *f*/). Formally, *f*\↔*f*⍀⍤1, and *f*/↔*f*⌿⍤1.

TRANSLITERATION

The ability to represent letters or words in the corresponding characters in another alphabet, known in natural languages as *transliteration*, can also be very useful in formal languages. For example, in APL one might substitute for the word *RHO* (entered by someone using a deficient terminal, or frightened of symbols other than the Roman alphabet) the symbol ⍴, or conversely substitute for ⍴ (entered by someone who wishes to exploit the brevity and connotations of that symbol to refer to a related, but different, defined function) the word *RHO*.

We will consider only substitutions for individual *tokens* (that is, those elements of APL such as *abc*2, 2.34*e*6, and +) that serve as words in APL, and will exclude substitution at a *character* level (such as TCH for CH in the word CHEBYCHEV) as well as substitution for phrases (such as 1 2 3 for ⍳3).

Substitution for phrases will be avoided because it would necessarily concern syntax analysis of the sentence (to avoid, for example, substituting 1 2 3 for ⍳3 in the expression 2 3 5 7 11 ⍳3) rather than simple word substitution. However, there is no difficulty in allowing the entity that *replaces* a token to be a string of tokens as well as a single token. For example, substituting (○1) for *PI* would allow *PI* to be used as a constant, and substituting (1{⎕*ai*) for *CPU* would allow the use of *CPU* as a niladic function to give the computer time used.

We propose the introduction of a *transliteration* system variable (to be referred to here as ⎕*tr*) such that ⎕*tr* is a two-row matrix whose rows consist of enclosed strings of tokens. Just before evaluating any token in an APL sentence, a substitution is made if the token occurs in the first row of ⎕*tr*. Moreover, substituted elements are treated exactly as if they occurred in the original sentence and, as a consequence, substitutions may be chained.

FUNCTION DEFINITION

Because an ambivalent function is evaluated only in the context of arguments, the three expressions *mpf*←+.× and *mp*←*m*+.×*n* and *per*←+.×*n* can be used to assign names to three distinct entities, a matrix product function (*mpf*), a matrix product of two arguments (*mp*), and the *permanent* of a matrix (*per*),

Because a niladic function *nf* requires no argument, a similar distinction between an evaluation of the function, and the function itself, cannot be made. Consequently, *f*←*nf* must be used for *one* of the possible meanings.

If we choose to mean that *f* becomes the niladic function *nf*, then there is no mechanism for indicating evaluation of a niladic function. However, if we choose to mean that *f* becomes the result of executing *nf*, then niladic functions will continue to behave as they always have; moreover, canonical definition provides a means for associating any desired name with a niladic function.

We therefore propose that niladic functions continue to be used and defined in the established manner.

The most recent statement of the evolving "direct" definition operator occurs in [7]. We now introduce a slightly modified statement that 1) makes explicit the use of a system variable ⎕*s* for the sequence control vector (making branching and the re-starting of a halted function possible through expressions of the form ⎕*s*← rather than through the introduction of the branch arrow), and 2) makes indexing of the segments 0-origin:

1. *m*∇*d* produces a function, with *m* and *d* being the representations of the monadic and dyadic cases.

2. The general form of each representation is a vector *r* of enclosed *segments*, the segments being executed in an order determined by a (shared) *sequence control* vector ⎕*s* that is initially set to ⍳⍴*r*. Termination occurs upon exhaustion of the sequence control vector.

3. A label in element *k*{*r* is assigned the value *k*↓⍳⍴*r*.

4. The symbols ⍺ and ⍵ denote the left and right arguments, and ∆ is used for *self-reference* to the function itself, being used in recursive definitions as well as for defining one of the two cases in terms of the other.

5. A name is localized if it occurs *immediately* to the left of an assignment arrow in any segment; for example, 3×*a*←4+*b* ←⍵ localizes *a* but not *b*. Name localizations for the monadic and dyadic cases are independent.

6. The explicit result of a function is the result of the last statement executed which produced an explicit result, where expressions such as *x*←3+4 or 3+4 are assumed to produce explicit results, but ⍎'' and ⊣*a* are not. Automatic output is *not* produced by an expression such as 3+4; such output is produced only by expressions using ⎕←.

7. Every vector *v* is treated as ,⊃*v*, that is, a simple vector is treated as a single segment. Single segments may therefore be written in the form '0⌈⍵' ∇ '⍺||⍵' .

A function produced by the definition operator ∇ has unbounded ranks and coherence, and the custom parameter ⎕*C* set to ⍳0.

SYSTEM VARIABLES

As remarked earlier, the variant operator could be employed to make less cumbersome the use of functions now dependent upon system variables. Nevertheless, efforts to remove dependence on system variables should be continued, especially in cases where the dependence was inessential, and therefore ill-considered, and in cases where the need has been obviated by other developments in the language.

Index origin is an example of the former, introduced in [8] (not only for indexing, but for other functions such as residue) because of awareness of the convenience of 0-origin in treating computer hardware, and of the familiarity of 1-origin to people not acquainted with computers.

The situation has changed radically since then: the convenience of 0-origin in *all* areas has become more apparent; familiarity with 0-origin and its convenience has grown; and the bane of forever specifying index origin has become apparent to most APL programmers. We therefore re-iterate the proposal made in [7] that index origin be considered obsolescent, that is, maintained unchanged in existing primitive functions, but used in no new functions or operators.

Printing width (first controlled by a system command) is an example of a parameter which, though essential when introduced (before the existence of the format function, when there was no way within the language of controlling the width of output), is no longer essential, and may, in fact, impede the full exploitation of scrolling facilities now available on video terminals.

For example, in the implementation of Sharp APL on the IBM PC, a long row of a matrix may be shown "extended" rather than "folded" to fit the screen width, and the "window" may be scrolled over the row to view all parts of it. However, a narrow setting of print width (⎕*pw*) will cause each row to be emitted as a sequence of independent segments. An infinite setting of ⎕*pw* could overcome this difficulty, but may be impossible due to limitations in an APL system, or to implicit assumptions about ⎕*pw* made in applications designed for the system.

CATENATION AND RESHAPE OPERATORS

Consider a catenation operator *COP* and a reshape operator *ROP* such that the functions *f*←+*COP*-*COP*×*COP*÷ and *g*←2 2 *ROP f* would each have rank 0, and would produce (for each cell) results of shape 4 and 2 2 respectively.

More generally, we define *f COP g* as a function having the (necessarily common) rank of *f* and *g*, and producing a leading axis catenation of the results of *f* and *g*. For example, if *f*, *g*, and *h* all have result shapes 4 5, then the result shape of *f COP g COP h* is 12 5, and (since the result shape of ,¨<⍤*h* is 1 4 5) the result shape of *f COP g COP* (,¨<⍤*h*) is 3 4 5.

Finally, we define *s ROP f* as equivalent to *s*¨ρ⍤*f*.

REFERENCES

1. Kenneth E. Iverson, *Operators and Functions* (IBM Corporation, RC7091, 1978).

2. Kenneth E. Iverson, *The Derivative Operator* (Proceedings of APL79: ACM 0-89791-005 2/79/0500 0347), 347.

3. Robert Bernecky and Kenneth E. Iverson, *Operators and Enclosed Arrays* (I.P. Sharp, Proceedings of the User Meeting, 1980).

4. Kenneth E. Iverson and Peter K. Wooster, *A Function Definition Operator* (APL Quote Quad, Volume 12, Number 1, Proceedings of APL81, ACM September 1981).

5. Arthur Whitney and Kenneth E. Iverson, *Practical Uses of a Model of APL* (Apl Quote Quad, Volume 13, Number 1, Proceedings of APL82, ACM, New York 1982).

6. Kenneth E. Iverson, *APL Syntax and Semantics* (ACM, APL83).

7. Kenneth E. Iverson, *Rationalized APL* (I.P. Sharp Associates 1983).

8. Kenneth E. Iverson, *A Programming Language*, John Wiley and Sons, New York, N.Y. 1962.

	Rank			*Surrogate*		
	m	*l*	*r*	*m*	*l*	*r*
+	0	0	0	αρ0	αρ0	αρ0
-	0	0	0	αρ0	αρ0	αρ0
×	0	0	0	αρ0	αρ0	αρ0
⌈	0	0	0	αρ0	αρ0	αρ0
⌊	0	0	0	αρ0	αρ0	αρ0
|	0	0	0	αρ0	αρ0	αρ0
>	0	0	0	αρ0	αρ0	αρ0
!	0	0	0	αρ0	αρ0	αρ0
○	0	0	0	αρ0	αρ0	αρ○1
⋆	0	0	0	αρ0	αρ1	αρ0
÷	0	0	0	αρ1	αρ0	αρ1
⍟	0	0	0	αρ1	αρ0	αρ1
~	0			αρ0		
∧		0	0		αρ0	αρ0
∨		0	0		αρ0	αρ0
≤		0	0		αρ0	αρ0
=		0	0		αρ0	αρ0
≥		0	0		αρ0	αρ0
≠		0	0		αρ0	αρ0
⍲		0	0		αρ0	αρ0
⍱		0	0		αρ0	αρ0
?	0	1	1	αρ1	αρ0	αρ¯
⌹	2	¯	2	*Note*	αρ0	*Note*
<	¯	0	0	αρ0	αρ0	αρ0
ρ	¯	1	¯	αρ0	αρ0	αρ1↑,ω
⍉	¯	1	¯	αρ1↑,ω	αρ0	αρ1↑,ω
ι	¯	¯	¯	αρ0	αρ0	αρ0
⊃	¯	¯	¯	αρ0	αρ0	αρ0
⍋	¯	¯	¯	αρ0	αρ' '	αρ' '
⍒	¯	¯	¯	αρ0	αρ' '	αρ' '
⌽	¯	¯	¯	αρ1↑,ω	αρ0	αρ1↑,ω
⊖	¯	¯	¯	αρ1↑,ω	αρ0	αρ1↑,ω
,	¯	¯	¯	αρ1↑,ω	αρ1↑,ω	αρ1↑,ω
⊣	¯	¯	¯	αρ1↑,ω	αρ1↑,ω	αρ1↑,ω
⊢	¯	¯	¯	αρ1↑,ω	αρ1↑,ω	αρ1↑,ω
⍕	¯	¯	¯	αρ' '	⌽αρ0 1	αρ0
∊		0	¯		αρ0	αρ0
⊥		1	¯		αρ0	αρ0
↑		1	¯		αρ0	αρ1↑,ω
↓		1	¯		αρ¯	αρ1↑,ω
⊤		¯	¯		αρ0	αρ0
≡		¯	¯		αρ0	αρ0

Note: αρ*id* ¯2↑1 1,α where *id*:(ι1↑ω)∘.=ι1↓ω

Table 1: FUNCTION RANKS AND SURROGATES