This document contains a series of 5 articles written for Vector magazine, www.vector.org.uk and published in Vols 13-14.

The documents were originally Word files, and have been merged together into a single PDF file. The conversion has some minor problems with alignment, but otherwise the documents are essentially as originally published.

Chris Burke
2 March 2005

APL and J

1. Function Rank

Though J shares many concepts with APL, in many respects it is radically different, and almost all APL constructions that are in J differ in some way. These differences can be a stumbling block to the experienced APL programmer who would like to learn a little about J.

In a series of articles entitled *APL and J*, we will try to explain the J way of thinking, by contrasting APL and J solutions to specific problems. The term APL will be used generically to refer to facilities found in most commercial implementations of APL.

We will start off by taking a look at function rank - one of J's most important new facilities - and also look at the related topics of J's agreement and assembly rules.

Housekeeping
We are going to need several arrays, and will use the letters `V`, `M`, and `A` followed by numbers to denote arrays and their shape. Thus `V5` for a vector of length 5, `M34` for a 3-by-4 matrix, `A243` for a 2-by-4-by-3 array and so on. Where actual values are required, `V5` will be defined as `ι5`, `M34` as `3 4ρι12`, etc.

APL terminology will be used, except where J terminology is required for clarity. `⎕io←1` throughout.

Agreement
Suppose we want to multiply two arrays together (or apply any scalar function to the arrays). What shapes must they have in order for the operation to be successful? In APL, the answer is straightforward: they must either have the same shape, or else one of them must be a scalar, in which case it is reshaped to match the other (this is known as *scalar extension*).

What happens if these conditions are not met, but the operation required is otherwise quite reasonable? For example, it is often the case that you want to multiply two arrays, where the shape of one array matches some leading elements, or trailing elements, of the shape of the other. Lets look at some simple examples, and typical APL solutions.

Example 1
The arrays are `M34` and `V4` and we want to multiply each row of the matrix by the vector. Straight multiplication fails:

```
      M34 × V4
RANK ERROR
```

The error is a rank error, since APL requires two conforming non-scalar arrays to have the same rank. A standard solution in APL is to reshape the vector into a matrix:

```
    M34 × (⍴M34)⍴V4
```

Another solution uses the following three-step process, which is of general utility. The data is split up into enclosed items, the function is applied to each item in turn, and the results are then reassembled:

```
    ⊃(⊂[2]M34)×¨⊂V4
```

Example 2

Now lets consider the case `M34` times `V3`, where we want to multiply each column of the matrix by the vector. Reshape still works, but the shape of the matrix must be reversed, and the result must be transposed:

```
    M34 × ⍉(⌽⍴M34)⍴V3
```

Similarly, three-step needs the result transposed:

```
    ⍉⊃(⊂[1]M34)×¨⊂V3
```

Example 3

In the next example, we multiply each 3-by-4 block of the array `A234` by the matrix `M34`. The methods of Example 1 can be used essentially unchanged:

```
    A234 × (⍴A234)⍴M34

    ⊃(⊂[2 3]A234)×¨⊂M34
```

Example 4

Finally, we multiply each 3-by-4 block of `A342` by the matrix `M34`, as follows:

```
    A342 × 3 1 2⍉(¯1⌽⍴A342)⍴M34

    3 1 2⍉⊃(⊂[1 2]A342)×¨⊂M34
```

Solutions in J - Examples 2 and 4

In APL, the default axis is the last axis, whereas in J, it is the first axis. Therefore, while in APL, Examples 2 and 4 are more complex than Examples 1 and 3, in J they are simpler:

```
    M34 * V3                       Example 2

    A342 * M34                     Example 4
```

Why do these multiplications work in J? The answer is that two array arguments agree if the frame of one array is a prefix of the frame of the other. This is an extension of the rules of APL.

The term 'frame' will be explained later, but for now, you can substitute the word 'shape' with the same effect. Thus, in Example 2, the shape of `V3` is the first element of the shape of `M34`, while in Example 4, the shape of `M34` is the first two elements of the shape of `A342`. In such cases, J extends the smaller array to match the larger; this is known as *prefix agreement*.

Solutions in J - Examples 1 and 3

Direct multiplication does not work this time in J, but note that the error is a length error, not a rank error:

```
   M34 * V4
length error

   A234 * M34
length error
```

The solutions use the rank operator:

```
   M34 *"1 V4                  Example 1

   A234 *"2 M34                Example 3
```

You can get a rough idea of what is going on by thinking of the numeric argument to rank as being the dimension to which the function applies. Thus in J Example 1, the function is applied to the rank 1 arrays, which are the rows of `M34`, and the vector `V4` as a whole. In J Example 3, the function is applied to the rank 2 arrays, which are the matrices of `A234`, and the matrix `M34` as a whole.

Function Rank

In APL, data may be of arbitrary size and dimension, where the dimension of the data is referred to as its rank. J extends this notion to functions - all functions are assigned a rank which determines the behavior of the function on certain subarrays called the cells.

The shape of an array can be split up into two parts, a leading part called the *frame*, and the remaining part called the *cells*. Either frame or cells can be empty. The frame can be thought of as the holding structure for the cells. The term *k-cell* specifies a cell of rank k.

For example, the shape of the matrix `M46` can be split up in three different ways:

- the size of the frame is empty, and size of the cell is 4 by 6. The cell is a 2-cell and is the entire matrix.
- the size of the frame is 4 and the size of the cells is 6. The cells are 1-cells and are the rows.
- the size of the frame is 4 by 6, and the size of the cells is empty. The cells are 0-cells and are the scalar elements of the matrix.

A function of *rank k* acts on the k-cells, with the following three-step process:

- the arguments are split up into k-cells
- the function is applied between each pair of k-cells
- the individual results are then reassembled

Note that these steps are analogous to the three-step APL examples. However, they are not identical. In APL, the steps must be given explicitly, and the interpreter carries out each step in turn. In J, the steps are only notional; in general the interpreter knows how to evaluate functions with rank and carries out the evaluation directly.

The *rank operator* can be used to enforce different argument pairings, which are independent of the function to which it is applied. The function is then applied to each of the pairs, and in turn takes care of its own pairings.

Applying Function Rank

Now lets see how function rank is applied in the examples above.

In J Examples 2 and 4 no function rank was specified, therefore the multiplications are carried out with the rank assigned to *, which is 0. This means that the cells are the scalars, and the frames are the entire argument shapes. This explains why, in the discussion on J Examples 2 and 4, it was stated that the term 'frame' could be replaced by the word 'shape'.

In J Example 1, the pairing is specified with rank 1. The three-step J process is thus:

Step 1. Break arguments into 1-cells. The 1-cells of `M34` are the rows:

```
    (1 2 3 4)   (5 6 7 8)  (9 10 11 12)
```

The 1-cell of `V4` is the entire vector: `(1 2 3 4)`

Step 2. The function is then applied between the cell pairs:

```
    (1 2 3 4*1 2 3 4) (5 6 7 8*1 2 3 4) (9 10 11 12*1 2 3 4)
```

Step 3. The individual results are then assembled into:

```
 1  4  9 16
 5 12 21 32
 9 20 33 48
```

In J Example 3 the pairing is specified with rank 2. The three-step J process thus uses the 2-cells, which for `A234` are the matrices:

```
   1  2  3  4        13 14 15 16
   5  6  7  8        17 18 19 20
   9 10 11 12        21 22 23 24
```

The 2-cell of `M34` is the entire matrix:

```
   1  2  3  4
   5  6  7  8
   9 10 11 12
```

This cell is multiplied with the 2-cells of `A234` and the results assembled into:

```
  1   4   9  16
 25  36  49  64
 81 100 121 144

 13  28  45  64
 85 108 133 160
189 220 253 288
```

Rank vs Each

J's rank operator `"` and APL's each operator `¨` are notationally similar, and are often confused. To understand the difference, note that the each operator is used for two essentially different expressions, for which J has two quite different solutions.

The first case is that discussed above, where APL each is used as part of the three-step solution to applying a function to non-conforming arrays; this usage corresponds to J's rank operator.

APL each also can be used to simply apply a function to each element of an array, typically an enclosed array. The corresponding expression in J is also called `each`, which is a standard utility defined as `&.>` . In such cases, the following are the same:

```
    f ¨ A                       APL
    f each A                    J
```

Assembly

The third step of applying rank is to reassemble the results. In the examples given here, assembly presented no problem since each item had the same shape. In practice, J can assemble items of different rank and shape as follows: the items are first brought to a common rank by introducing leading unit axes to any of lower rank, and are then brought to a common shape by padding with an appropriate fill element. This allows J to assemble results which in APL would signal rank or length errors:

```
      ⊃ (1 2 3) (2 2⍴10 11 12 13)             APL
RANK ERROR

   >1 2 3;2 2$10 11 12 13                     J
 1  2 3
 0  0 0

10 11 0
12 13 0
```

The combination of J's more general agreement rules, function rank and permissive assembly allows many more calculations to be expressed in J without looping or reshaping.

APL and J

2. Indexing

This series of articles entitled *APL and J* will try to explain the J way of thinking, by contrasting APL and J coding examples. The term APL will be used generically to refer to facilities found in most commercial implementations of APL.

Here we look at how J handles indexing and indexed replacement.

Housekeeping
We use the letters `V`, `M`, and `A` for vectors, matrices and arrays of any dimension, optionally followed by numbers to denote their shape. Thus `V` for any vector, `V5` for a vector of length 5, `M45` for a 4-by-5 matrix, `A245` for a 2-by-4-by-5 array. Where actual values are required, `V5` will be defined as `⍳5`, `M45` as `4 5⍴⍳20`, for example:

```
      M45
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

APL terminology will be used, except where J terminology is required for clarity. Since J uses origin 0 only, here `⎕io←0` is assumed throughout.

Indexing
In APL, you index variables using an indexing expression in brackets. Such expressions are not supported in J; instead, J uses a function `{` *from*. Compare the following expressions to select row 2 of a matrix:

```
    M[2;]                       APL

    2{M                         J
```

To understand this change, note that bracket indexing in APL does not conform to the rest of the APL language. Indexing is essentially a dyadic function taking arguments of the data and indices, yet the APL expression does not have the form of other APL dyadic functions (i.e. a function with arguments to the left and right).

Moreover you cannot use the APL expression as you would a normal function. For example if N is an enclosed array, each item of which is a matrix, you could not use bracket indexing to select row 2 of each item:

```
    N ¨ [2;]
SYNTAX ERROR
```

If you wanted to do this, you could define a cover function to index a matrix:

```
    ∇ r←ndx rowndx mat
[1]   r←mat[ndx;]
    ∇
```

You could then select row 2 of M as:

```
    2 rowndx M
```

and row 2 of each element of N as:

```
    2 rowndx¨ N
```

The corresponding J expressions are:

```
    2 { M
    2 {each N
```

It may be seen that `{` is simply a functional form of bracket indexing. Because it is an ordinary function, you can use it in functional expressions, as in the example above. The idea of `{` is similar, but not identical, to the index function ⌷ in APL2.

Note that in the above example, should you wish to index row 2 of a 3D array, you would need to modify *rowndx* accordingly, but the J expression would remain unchanged.

Here are a couple more examples:

```
    item2=. 2&{                          function to return item 2

    item2 M45
10 11 12 13 14

    item2 A324
16 17 18 19
20 21 22 23

    mrow=. {&M45                         function to return rows of M45

    mrow 3 2
15 16 17 18 19
10 11 12 13 14
```

From Left Argument

The left argument can take various forms:

1. It can be one or more integers, which select corresponding items from the right argument. Negative integers select from the end, for example:

```
    2 _1 { M45                                  row 2 and last row of M45
10 11 12 13 14
15 16 17 18 19
```

It is often convenient to apply rank with this form, for example:

```
    2 _1 {"1 M45                             column 2 and last column of M45
 2  4
 7  9
12 14
17 19
```

2. The left argument can be one or more boxed lists, each element of which is a list of integers that select from corresponding axes:

```
    M45[1;3]                                 APL
8

    (<1 3){M45                               J
8
```

This form allows scattered indexing:

```
    M45[(1 3)(2 4)]                          APL2 and Dyalog APL
8 14

    (1 3;2 4){M45                            J
8 14
```

3. The boxed lists in the left argument may themselves contain boxed lists, each element again selecting from corresponding axes:

```
    M45[1;3 4]                               APL
8 9

    (<1;3 4){M45                             J
8 9
```

Note that in J, this last form permits scattered indexing where each index may have a different number of elements. The following example has two indices: the first picks row 1, columns 3 and 4; the second picks row 2, column 3. The two results are assembled into a matrix, padding the second row with a fill of 0:

```
    ((1;3 4);2 3){M45
 8 9
13 0
```

Selecting Entire Axis

In APL, when an axis is elided, the entire axis is selected. This is not the same as using an empty index:

```
    M45[;2]                          pick all rows, column 2
2 7 12 17

    M45['';2]                        pick no rows, column 2
```

In J, the elided axis is given by the boxed empty vector. Thus:

```
   (<(<'');2){M45                   pick all rows, column 2
2 7 12 17
```

```
   (<'';2){M45                      pick no rows, column 2
```

Here, it is convenient to use `a:` *ace*, which is the boxed empty vector:

```
   (<a:;2){M45                      pick all rows, column 2
2 7 12 17
```

In general, a boxed argument used as an index selects all but the indices given. So, the boxed empty means select all but the empty, i.e. select all. The following picks all but the first and last row from columns 2 and 3:

```
   (<(<0 _1);2 3){M45
 7  8
12 13
```

Indexed Replacement

As with indexing, indexed replacement in APL does not conform to the rest of the language. It requires three arguments - a set of indices, and two sets of data, at least one of which must be named. The named data item and the indices are given to the left of assignment, and the other data item to the right:

```
   M[2;] ← V                        replace row 2 of M with V
```

Indexed replacement is not a functional expression. The “result” of indexing is the values assigned (V in the above example), so that M is updated as a side effect.

In contrast, indexed replacement in J is an ordinary part of the language. It has two steps:

- the indices to be replaced are given as the argument to the adverb `}` *amend*, returning a dyadic function that merges its two arguments according to the indices.
- the function is then applied to the two data arguments. The elements of the left argument replace the elements of the right argument according to the indices.

For example, to replace row 2 of M with V:

```
   V 2} M
```

The order of execution is shown by:

```
   V (2}) M                    2} is a function that merges item 2
```

To replace row 2 of each item of a boxed list of matrices:

```
   (<V) 2} each N
```

Note that indexed replacement in J does not require assignment and does not update the arguments, so there is no need for a named argument. The following updates M:

```
    M=. V 2} M
```

Arguments To Amend

The index arguments to `}` are symmetrical with index arguments to `{`. For example:

```
    2{M45                                              select row 2
10 11 12 13 14

    (100*i.5) 2} M45                              amend row 2
 0   1   2   3   4
 5   6   7   8   9
 0 100 200 300 400
15  16  17  18  19

    (<1;1 2 3){M45                                select row 1 columns 1 2 3
6 7 8

   100 200 300 (<1;1 2 3)} M45               amend row 1 columns 1 2 3
 0   1   2   3  4
 5 100 200 300  9
10  11  12  13 14
15  16  17  18 19
```

Selective Assignment

Modern APLs can replace a portion of an array selected by an expression given to the left of assignment, this is called selective assignment. For example, the following replaces all occurrences of 2 in V by 20:

```
    V←1 2 2 3 1 2

    ((V=2)/V) ← 20

    V
1 20 20 3 1 20
```

Similar facilities in J are provided by using a function argument to amend, which function is used to create the required indices. Thus:

```
    bx 1 0 1 1 0 1 1                     the utility bx returns indices of a boolean
0 2 3 5 6

    f=. bx @ (2:=])                      f returns indices where right argument is 2

    20 f} 1 2 2 3 1 2                    use f to replace 2 by 20
1 20 20 3 1 20
```

Fetch

A new function in J3.02 is `{::` *fetch* which when used dyadically as in `x fetch y` returns the subarray of y according to the path x; the selection at each level is based on `{`, for example:

```
   [A=. 1 2 3;4 5;i.4 5
+-----+---+--------------+
|1 2 3|4 5| 0  1  2  3  4|
|     |   | 5  6  7  8  9|
|     |   |10 11 12 13 14|
|     |   |15 16 17 18 19|
+-----+---+--------------+
```

```
   (2;_1 _1) fetch A
19
```
fetch from item 2, the element in the last row and column

```
   (2;<2;2 3 4) fetch A
12 13 14
```
fetch from item 2, the elements indexed by 2;2 3 4

This is similar to the *reach* facility of Dyalog APL, except that in J the final selection need not be a scalar:

```
   A[(2(2 2))(2(2 3))(2(2 4))]
12 13 14
```

APL and J

3. Operators - Reduce, Scan, Outer Product

This series of articles entitled *APL and J* will try to explain the J way of thinking, by contrasting APL and J coding examples. The term APL will be used generically to refer to facilities found in most commercial implementations of APL.

For the next few articles, I will discuss operators. To start off with, we will take a look at J's use of operators as a whole, then look specifically at reduce, scan and outer product.

Housekeeping
As usual, we use the letters `V`, `M`, and `A` for vectors, matrices and arrays of any dimension, optionally followed by numbers to denote their shape. Thus `V` for any vector, `V5` for a vector of length 5, `M45` for a 4-by-5 matrix, `A245` for a 2-by-4-by-5 array. Where actual values are required, `V5` will be defined as `⍳5`, `M45` as `4 5⍴⍳20`, for example:

```
      M45
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

APL terminology will be used, except where J terminology is required for clarity. Since J uses origin 0 only, here `⎕io←0` is assumed throughout.

Overview
J makes much greater use of operators than APL, and this is probably the hardest aspect of J to grasp.

For example, if you tried to write programs without using operators at all, in APL this would be a minor inconvenience, whereas in J you would be completely stuck - since all J programs are created with operators!

In general, while operators are common in APL, they typically occur in a relatively few expressions such as `+/` and `∧.=` which become familiar and are used as if they were primitive functions. Thus the APL programmer may try to solve a problem by thinking solely in terms of the "functions" that can be brought to bear on it. This approach will not work in J; or at least, if you try to think solely in terms of functions, you will find it tough going! Operators are essential in solving problems, and it is important in J that you develop a good understanding of how they work.

Moreover you not only use operators, you also need to be able to define them, whereas in APL it would be unusual to define new operators to solve problems. Indeed one major implementation, APL+WIN, does not even allow you to define new operators.

This focus on operators in J can be a bit intimidating to the newcomer, who may think that programming with operators is difficult. True there is some learning involved, but once mastered, it becomes pretty simple. Moreover, it gives you a much clearer understanding of how APL works - a good reason to learn J is that it will teach you so much about APL!

Tacit Definition

An operator expression like `+/` is called a tacit definition, in that it defines a new function without referring to the new function's argument. Compare with *explicit* definitions which do refer to the function's argument (e.g. any APL function defined with the function editor). Tacit definitions occur frequently in both APL and J, though the terminology is new to J and can be a source of difficulty. I have heard experienced APL programmers say they do not understand tacit definition - clearly they are unaware that they use it all the time in APL.

Tacit definition may be clarified by giving the new function a name, which in APL is only possible with Dyalog APL (but which should be possible in all APLs):

```
   sum← +/                    Dyalog APL
   sum=. +/                   J
```

Using Dyalog's composition operator, it becomes easy to generate the kind of tacit definitions so often seen in J:

```
   root← * ∘ 0.5
   square← * ∘ 2
   dist← root ∘ sum ∘ square
   dist 5 12
13
```

When should you use tacit, and when explicit definition? Whenever you feel like it! For example the following APL function uses both tacit and explicit definition:

```
    ∇ r←dist dat
[1]   r←(+/dat*2)*0.5
    ∇
```

Is tacit definition better than explicit definition? It certainly can be - try writing an APL function that simulates `+/` without using tacit definition. Then modify this function to handle the general case of `f/`.

Now lets look at some APL operators.

APL Reduce

At first sight, reduce in J (called *insert*) seems to be the same as in APL:

```
   +/ 2 3 5 7                 APL
17

   +/ 2 3 5 7                 J
17
```

However, with a matrix argument, APL applies +/ along the last axis, and J along the first axis:

```
   M34
 0 1  2  3
 4 5  6  7
 8 9 10 11

   +/ M34                          APL
6 22 38
```

```
   +/ M34                        J
12 15 18 21
```

So `+/` in J is closer to APL's `+⌿` or `+/[0]`, i.e. plus reduce along the first axis. To apply `+/` in J along the last axis as in APL, apply it *rank 1*, using the rank operator:

```
   +/"1 M34
6 22 38
```

Since reduce applies a function between each element of a vector, you might think that `,/` is an identity on a vector. This is true in J, but not in APL:

```
   V5 -: ,/ V5                   J
1

   V5 ≡ ,/ V5                    APL
0
```

Instead, `,/` in APL encloses its result:

```
   (⊂V5) ≡ ,/ V5                 APL
1
```

What is happening is that *insert* in J inserts the function between the *items* of its argument, whereas *reduce* in APL inserts the function between the *elements of the vectors along the last axis, and encloses the result for each vector*, so that:

```
   f/ M34
```

means:

```
   0 1 2 3  f  4 5 6 7  f  8 9 10 11                             J

   (0 f 1 f 2 f 3) (4 f 5 f 6 f 7) (8 f 9 f 10 f 11)          APL
```

Strictly speaking, the latter is the APL2 definition of reduce, seen in APL2, APL+WIN and Dyalog APL. The treatment in SHARP APL is essentially that of J, except items are based on the last axis.

It is interesting that APL does an implicit enclose. It is very often the case that an APL definition uses enclose, where J does not. For example:

```
   ,/ M34                                               J
0 1 2 3 4 5 6 7 8 9 10 11

   display ,/ M34                                       APL
┌→──────────────────────────────────────┐
│ ┌→──────┐ ┌→──────┐ ┌→────────┐ │
│ │0 1 2 3│ │4 5 6 7│ │8 9 10 11│ │
│ └~──────┘ └~──────┘ └~────────┘ │
└∊──────────────────────────────────────┘
```

Since there is no difference in APL between a scalar and an enclosed scalar, the final result of an APL reduce may be simple. For example, `+/M34` is:

```
    (0 + 1 + 2 + 3) (4 + 5 + 6 + 7) (8 + 9 + 10 + 11)
```

which is:

```
    (6) (22) (38)
```

which is:

```
    6 22 38
```

The effect of `,/ M34` in APL can be produced in J using link insert:

```
   ;/ M34
┌───────┬───────┬──────────┐
│0 1 2 3│4 5 6 7│8 9 10 11 │
└───────┴───────┴──────────┘
```

The distinction between the way reduce applies to *vectors*, and the way insert applies to *items* becomes clearer with a 3D argument:

```
    display ,/ A234                                          APL
┌→─────────────────────────────────────────────────────┐
↓ ┌→────────┐      ┌→────────┐      ┌→──────────┐      │
│ │0 1 2 3  │      │4 5 6 7  │      │8 9 10 11  │      │
│ └~────────┘      └~────────┘      └~──────────┘      │
│                                                      │
│ ┌→─────────────┐ ┌→─────────────┐ ┌→─────────────┐   │
│ │12 13 14 15   │ │16 17 18 19   │ │20 21 22 23   │   │
│ └~─────────────┘ └~─────────────┘ └~─────────────┘   │
└∊─────────────────────────────────────────────────────┘
```

```
    ;/ A234                                                  J
┌──────────┬───────────┐
│0 1  2  3 │12 13 14 15│
│4 5  6  7 │16 17 18 19│
│8 9 10 11 │20 21 22 23│
└──────────┴───────────┘
```

```
    ,/ A234                                                  J
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
```

Replicate
In APL, the symbol `/` is used both for reduce and for replicate, whereas in J, replicate is given by a different symbol, `#` (copy).

APL Scan

Scan is called *prefix* in J, and is clearly not the same as in APL:

```
   +\ 1 2 3 4                          J
1 0 0 0
1 2 0 0
1 2 3 0
1 2 3 4
```

The difference is that in APL, scan applies the verb with reduce, while J just applies the verb. Thus

```
   f \ 1 2 3 4
```

means:

```
   (f / 1) (f / 1 2) (f / 1 2 3) (f / 1 2 3 4)    APL

   (f 1) (f 1 2) (f 1 2 3) (f 1 2 3 4)                     J
```

Note that in the APL case *f* is dyadic, whereas in J, `f` is monadic.

In J, each application of `f` returns an item of the result, so that

```
   +\ 1 2 3 4
```

is:

```
   (+1) (+1 2) (+1 2 3) (+1 2 3 4)
```

J assembles these four items into a matrix, padding with zeroes as required.

To simulate APL scan, you need to use reduce explicitly:

```
   +/\ 1 2 3 4
1 3 6 10
```

The benefit of not having an automatic reduce in prefix, is that many operations do not require it. Such operations are straightforward in J, but may be awkward in APL. The following is a reverse scan in J - how would you do this in APL?

```
   <@|.\ i.6
┌─┬───┬─────┬───────┬─────────┬───────────┐
│0│1 0│2 1 0│3 2 1 0│4 3 2 1 0│5 4 3 2 1 0│
└─┴───┴─────┴───────┴─────────┴───────────┘
```

For another example, to enclose successive prefixes of a variable, you can use box prefix in J, but not enclose scan in APL. Curiously, since `,/` is an APL reduction that encloses its argument, it turns out that at least for a vector, `,\` performs an enclose scan!

```
   <\ 1 2 3 4                                    J
┌─┬───┬─────┬───────┐
│1│1 2│1 2 3│1 2 3 4│
└─┴───┴─────┴───────┘
```

```
    display ,\ 1 2 3 4
┌→──────────────────────────────┐
│    ┌→───┐ ┌→─────┐ ┌→───────┐ │
│ 1  │1 2│ │1 2 3│ │1 2 3 4│ │
│    └~───┘ └~─────┘ └~───────┘ │
└∊──────────────────────────────┘
```

Inverse

Some uses of prefix have inverses. For example, the inverse of `+/\` is the first differences, and of `*/\` is the successive ratios:

```
    inverse=. ^:_1

   +/\ inverse 1 3 6 10
1 2 3 4

    */\ inverse 2 3 5 7 11
2 1.5 1.66667 1.4 1.57143
```

Suffix

Suffix is analogous to prefix, except it applies to lists of elements taken from the end. Thus a reverse sum scan is:

```
    +/\. 1 2 3 4                J
10 9 7 4

    ⌽+\⌽ 1 2 3 4                APL
10 9 7 4
```

APL Outer Product

In J, the verb defined using insert may also be used dyadically (when it is called *table*), and is similar to APL's outer product.

```
    1 2 3 +/ 10 20 30 40                J
11 21 31 41
12 22 32 42
13 23 33 43

    1 2 3 ∘.+ 10 20 30 40               APL
11 21 31 41
12 22 32 42
13 23 33 43
```

Again, the definitions are not quite the same. Table in J applies the function between each cell of the left argument, and the entire right argument, whereas outer product in APL applies the function between each pair of elements from the left and right argument.

Consider the following APL example of catenating each pair of elements of two vectors:

```
    display 1 2 ∘., 10 20 30
┌→──────────────────────────┐
↓ ┌→─────┐ ┌→─────┐ ┌→─────┐ │
│ │1 10│ │1 20│ │1 30│ │
│ └~─────┘ └~─────┘ └~─────┘ │
│                             │
│ ┌→─────┐ ┌→─────┐ ┌→─────┐ │
│ │2 10│ │2 20│ │2 30│ │
│ └~─────┘ └~─────┘ └~─────┘ │
└∊──────────────────────────┘
```

You might expect that `,/` does the same thing in J, but it does not:

```
   1 2 ,/ 10 20 30
1 2 10 20 30
```

This is because the rank of catenate is unbounded, hence the left argument has only one cell (the entire argument), and this is applied to the entire right argument. The above expression is therefore the same as:

```
   1 2 , 10 20 30
1 2 10 20 30
```

You can change this behavior by using rank. For example, if both ranks are 0, then J will catenate each pair of elements as in APL. The individual results are assembled in a simple array, rather than enclosed:

```
   1 2 ,"0/ 10 20 30
1 10
1 20
1 30

2 10
2 20
2 30
```

To duplicate APL's behavior exactly, use box atop catenate:

```
   1 2 <@,"0/ 10 20 30
┌────┬────┬────┐
│1 10│1 20│1 30│
├────┼────┼────┤
│2 10│2 20│2 30│
└────┴────┴────┘
```

Alternatively, by setting the left rank to 0 and leaving the right rank unbounded, you can catenate each element of the left argument to the entire right argument:

```
   1 2 ,"0 _/ 10 20 30
1 10 20 30
2 10 20 30
```

Leaving the left rank unbounded and setting the right rank to 0 catenates the entire left argument to each element of the right argument.

```
   1 2 ,"_ 0/ 10 20 30
1 2 10
1 2 20
1 2 30
```

N-Wise Reduction

In APL2, a reduce may take a left argument that specifies the number of items to be used in each application of the function, called *n-wise reduction*, e.g.

```
    2 +/ 1 2 3 4 5 6
3 5 7 9 11
```

This is:

```
    (1+2) (2+3) (3+4) (4+5) (5+6)
```

In J, this is handled by prefix (where it is called *infix*). As in APL2, the left argument specifies the number of items to be used. If positive, the items overlap; if negative, the items are non-overlapping. For example:

```
    2 +/\ 1 2 3 4 5 6
3 5 7 9 11

    _2 +/\ 1 2 3 4 5 6
3 7 11
```

As with prefix, the verb argument to infix need not be applied with reduce. For example:

```
    _2 <\ 1 2 3 4 5 6
┌───┬───┬───┐
│1 2│3 4│5 6│
└───┴───┴───┘
```

A neat way of creating a matrix with a given number of columns from a vector, is to use identity infix:

```
    _2 ]\ 1 2 3 4 5 6
1 2
3 4
5 6
```

APL and J

4. Function Application and the Axis Operator

This series of articles entitled *APL and J* will try to explain the J way of thinking, by contrasting APL and J coding examples. The term APL will be used generically to refer to facilities found in most commercial implementations of APL.

Continuing the discussion of APL's operators, we now look at the axis operator, and the whole question of how functions apply to arrays.

Housekeeping
As usual, we use the letters `V`, `M`, and `A` for vectors, matrices and arrays of any dimension, optionally followed by numbers to denote their shape. Thus `V` for any vector, `V5` for a vector of length 5, `M45` for a 4-by-5 matrix, `A245` for a 2-by-4-by-5 array. Where actual values are required, `V5` will be defined as `⍳5`, `M45` as `4 5⍴⍳20`, for example:

```
      M45
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
 16  17  18  19  20
```

APL terminology will be used, except where J terminology is required for clarity. Here, `⎕io←1` is assumed throughout.

Note that the behaviour of the axis operator is not identical in all APLs. This does not affect the gist of this article, but you may find that, with your copy of APL, the examples are not exactly as shown.

Overview
In this article, we cover what is perhaps the most important single difference between APL and J – the way functions apply to arrays. The experienced APLer may be surprised that there is anything to say at all, since it is not obvious where J could possibly differ from APL. Yet there are major differences that can be quite difficult to grasp. To explain what is going on, we will construct a model for APL, that can also be applied to J.

Function Application in APL
We start off with a description of function application in APL. We consider the monadic application of `f` to `A` and ask whether it is possible to split `A` up into subarrays in such a way that applying `f` in turn to each subarray, then assembling the results, would be the same as applying `f` to `A`. If so, we could consider `f` as being, in effect, defined on those subarrays. The following examples should make this clear.

Signum
Consider the signum function applied as below:

```
      ⎕← M← M23 - 3
¯2 ¯1 0
 1  2 3

       ×M
¯1 ¯1 0
```

```
  1   1  1
```

Clearly we could apply signum to each scalar element of *M* in turn, with the same effect. Thus `×`*M* is equivalent to the following array of function applications:

```
    (×¯2) (×¯1) (×0)
   (×1)  (×2)  (×3)
```

Since signum can apply to each scalar in turn, we say that it *applies to scalars*, or equivalently, since scalars have rank 0, we say that it is a *rank-0* function.

Note that this is not the usual meaning of the APL term *scalar function*, which means a function that not only applies to scalars, but also returns scalars. In fact, signum is such a function. However in this discussion, we are classifying functions by how they apply, and not by the results they return.

For example, we could have a function that applies to scalars, but which does not return scalar results. For such a function, we would need some way of determining how the overall result is assembled. There are essentially two such *assembly schemes* in use:

1. Suppose *sf* is the shape of the array of function applications (for example, *sf* would be `2 3` in the signum example above), and that *sr* is the shape of the result of each function application, then we could assemble the overall result as an array of shape *sf*`,`*sr*. This scheme works fine as long as the result of each function application does indeed have a common shape, *sr*. If not, then the individual results would first have to be coerced into the same shape, typically by prefixing unit axes, and padding with fill elements.
2. In the second scheme, the result of each function application is enclosed, and therefore is a scalar. Consequently, *sr* is empty and the overall shape of the result is *sf*.

Both these schemes are implemented in APL, but only the first in J. Also, in APL, the result shape in the first scheme is not always *sf*`,`*sr*, but can be some permutation of this.

The two schemes are easily interchangeable. The second scheme can be obtained from the first by using a function that does an explicit enclose of its result; while the first scheme can be obtained from the second by doing a disclose of the final result (plus a transpose where the APL result is a permutation of *sf*`,`*sr*).

Sum
Now lets look at `+/` and note that unlike signum, it does not apply to scalars. If we tried to apply it to scalars, as in `+/` applied to *M*`23` below, this would result in *M*`23` unchanged:

```
    (+/1) (+/2) (+/3)
   (+/4) (+/5) (+/6)
```

We could say that `+/` applied with rank 0 is the identity function.

On the other hand, it should be clear that `+/` applies to vectors, and is therefore a *rank-1* function. For example, we could apply it as:

```
    (+/1 2 3) (+/4 5 6)
6 15
```

Not only can we do this, but in fact in modern APL's, this is the *definition* of `+/`. Strictly speaking, the definition has an implied enclose after each function application (and so uses the second assembly scheme described above), but since the enclose of a scalar is the same scalar, the enclose has no effect:

```
      (⊂+/1 2 3) (⊂+/4 5 6)
6 15
```

In describing a function as being *rank-n*, we mean that n is the lowest rank at which it works correctly. On higher rank arrays, the function application can be considered as partitioning the array into subarrays of rank n, then applying the function to each subarray, and assembling the results. Thus both signum (a rank-0 function), and `+/` (a rank-1 function) work fine on matrices, or arrays of higher rank.

Ravel

It should be clear that the usual result of ravel is changed if it is applied with a rank less than the rank of its argument. For example, suppose we tried to apply it with rank 0 to `M23`, and used the first assembly scheme. The result of each function application to a scalar would be a vector of length 1, and we would have:

```
      ρ (, with-rank-0) M23
2 3 1
```

If we used the second assembly scheme, then each function application would be enclosed, and the result would have shape `2 3`. This is, in fact, precisely what `,¨` does:

```
    display ,¨M23
┌→────────────────┐
↓ ┌→┐  ┌→┐  ┌→┐   │
│ │1│  │2│  │3│   │
│ └~┘  └~┘  └~┘   │
│                 │
│ ┌→┐  ┌→┐  ┌→┐   │
│ │4│  │5│  │6│   │
│ └~┘  └~┘  └~┘   │
└∊────────────────┘
```

(In general, `f¨` means apply `f` as a rank-0 function, opening each scalar before applying the function, and using the second assembly scheme.)

We say that ravel is a whole-array function, or equivalently, that it is *rank-infinity*.

Axis

Now lets look at what the axis operator gives us. If `f` is applied to an array `A` of rank n, and `f` is a rank-m function where 0 < m < n, then there will be more than one way of picking the rank-m subarrays of `A`. The axis operator lets you pick those subarrays. For example, if we apply the rank-1 function `+/` to a matrix, then we might choose either the row-vectors or the column-vectors.

As another example, we could use axis to choose subarrays for ravel, as in:

```
      ,[2 3] A234
 1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24
```

This example means "apply ravel to axes 2 and 3 of `A234`", and is an example of the first assembly scheme described above (the individual results of ravel are not enclosed).

Also, as this example demonstrates, the axis operator not only allows you to pick the axes at which a function applies, but also, as a side effect, to *specify the rank at which the function applies*. Does this sound familiar – it should! The expression `,[2 3]A234` applies ravel *with rank-2*, to the subarrays along the last two axes.

We could also apply ravel with rank 1, as in `,[2]A234`, but since ravel applied to a vector is the identity, then `,[2]A234` is the identity on `A234` (this is an example where, when using the first assembly scheme, APL does not follow the result shape `sf,sr`). Hence ravel with rank 1 is also an identity function.

The notation for the axis operator also permits application with rank 0, as `,[⍬]`, though in practice this is not supported. If it were, we would expect that:

```
    (⍴,[⍬]A)   ←→   (⍴A),1
```

APL Summary
At this stage, we have produced what seems to be a fairly simple and general model for function application in APL. We can classify functions according to the rank at which they are applied; we have two assembly schemes to put results together; and we can use the axis operator to specify axes where appropriate.

Unfortunately, this nice model does not always work! It has emerged from a consideration of how some functions actually behave, but does not describe how they were defined in the first place. There are simply too many exceptions for it to be consistent. We have seen one such already – the axis operator permits you to specify the rank at which a function is applied – except for rank 0 which has not been implemented! Here are some more exceptions:

Since `,[2 3]` applies ravel to the last two axes, we might expect that `⊂[2 3]` encloses the last two axes, which indeed it does:

```
    display ⊂[2 3] A234
┌→──────────────────────────────────────┐
│ ┌→──────────────┐  ┌→──────────────┐  │
│ │ 1   2   3   4 │  │ 13 14 15 16 │  │
│ │ 5   6   7   8 │  │ 17 18 19 20 │  │
│ │ 9  10  11  12 │  │ 21 22 23 24 │  │
│ └~──────────────┘  └~──────────────┘  │
└∊──────────────────────────────────────┘
```

However, try this with `⍉` or with `⌽`:

```
       ⍉[2 3] A234
SYNTAX ERROR

       ⌽[2 3] A234
LENGTH ERROR
```

Not only do these not work, the error messages are wrong! The first shows a syntax error, but there is nothing wrong with the syntax, which is exactly that used earlier with ravel and enclose. The second shows a length error, but it is not possible for any monadic use of rotate to have a length error! When the APL interpreter itself gets error messages wrong, you know something strange is happening, and indeed the implementation of the axis operator is a patchwork of special cases.

Now lets try using axis with a defined function. Define sum as:

```
     ∇ r←sum m
[1]    r←+/m
     ∇
```

and note that it does not work with axis:

```
      sum[1] M23
NONCE ERROR
```

Next, consider ⎕fi, which is clearly a rank-1 function, and try applying it to a matrix. As with `+/`, we might expect it to apply to each row in turn and then assemble the results together, but it fails:

```
      ⎕fi ⍕M23
RANK ERROR
```

We cannot complain that APL functions do not behave according to the model we have created, since no such model was used at outset. Yet there would obviously be a considerable benefit to programmers were a model of this type to be uniformly implemented.

Function Application in J

We start off by noting that the foregoing description of APL functions in terms of the rank at which they behave is identical in J. However, as mentioned earlier, J only follows the first assembly scheme (and therefore never does an implicit enclose of any result).

The big (and surprising) difference is that *J has no axis operator*! Whenever a function is applied to a set of axes in J, they are always taken from the end. Thus APL has first-axis functions, last-axis functions, and - using the axis operator - any-axis functions, whereas J has *only* last-axis functions. What J does have is a rank operator – which is just the side-effect noted earlier of APL's axis operator.

For example, compare:

```
    ,[2 3] A234               APL

    ,"2 A234                  J
```

The APL expression means apply ravel along axes 2 and 3 of `A234`. It happens that these are the last two axes, and also that ravel with axis uses the first assembly scheme.

The J expression means apply ravel with rank 2 to `A234`. Since axes in J are always the last axes, then it must be that ravel is applied to the last two axes; also J only uses the first assembly scheme. Therefore, it may be seen that the two expressions are identical.

Now consider the APL expression `,[1 2]A234`. This ravels the first two axes of `A234`. There is no direct equivalent of this in J, since J has no way of specifying the first two axes. The only way to do it is to transpose the first two axes to the end, then ravel and transpose back:

```
    |: (,"2) 0 1 |: A234
```

At this stage, the picture we have is that functions apply to their arguments the same way in APL and J, but the operators that modify the behaviour are different: APL has axis, which includes rank, while J has only rank. With this picture, J seems so limited, you wonder how it works at all! But J has a few cards in hand.

Consistency
First note that while J’s model is simpler than APL’s, unlike APL, it is applied consistently to all functions. Lets go through the earlier examples where APL fails:

In J, you can use rank to apply a function to scalars. For example, ravel with rank 0 appends a unit trailing axis:

```
   $ ,"0 A234
2 3 4 1
```

In J, all functions work with rank:

```
   |: "2 A234                 transpose rank-2
 1  5  9
 2  6 10
 3  7 11
 4  8 12

13 17 21
14 18 22
15 19 23
16 20 24

   |."2 A234                             rotate rank-2
 9 10 11 12
 5  6  7  8
 1  2  3  4

21 22 23 24
17 18 19 20
13 14 15 16
```

You can use rank with explicit definitions:

```
   sum=: 3 : '+/y.'
   sum"1 M23
6 15
```

You can apply any rank-1 function to an array of higher rank:

```
   fi=: 0&".                             equivalent of ⎕fi
   10 * fi ": M23                        applied to a character matrix
10 20 30
40 50 60
```

Dyadic Use
Unlike axis, J’s rank operator can be used to specify the left and right ranks of dyadic functions. In theory, axis could be used to do the same; for example, if a semicolon were used to delimit left and right axes, but this has not been implemented. For example:

```
    A=: 10 * B=: 1 2 3

   A +"0 1 B                          apply + with left rank 0, right rank 1
11 21 31
12 22 32
13 23 33
   A +"1 0 B                          apply + with left rank 1 right rank 0
11 12 13
21 22 23
31 32 33
```

Item Functions

On the face of it, J's lack of an axis operator seems incredibly restrictive. While the example mentioned earlier of applying ravel to the first two axes of a 3D array may not seem very important, it is easy to think of important examples where axis seems essential – for example in determining whether `+/` of a matrix applies along the rows or along the columns. It turns out, however, that the extra facilities provided by axis are typically not required in J. This is because of *item functions*, and *prefix agreement*.

J defines many functions, including `+/`, to apply to the *items* of an array, which are the subarrays consisting of all axes but the first, for example the rows of a matrix.

Thus in J, `+/M23` is *defined* to be:

```
    1 2 3 + 4 5 6
```

Similarly, `+/M234` is *defined* to be:

```
 1  2  3  4       13 14 15 16
 5  6  7  8   +   17 18 19 20
 9 10 11 12       21 22 23 24
```

Note that this definition makes no use of axis, and indeed, `+/` has rank infinity. There is no concept as in APL, of splitting up the argument into vectors, applying `+/` to each vector, then reassembling the results.

It turns out that `+/` in J behaves like `+⌿` in APL. Moreover, when you change the rank of `+/` in J, you get the same effect as using the axis operator in APL. For example, the following are equivalent:

J	APL
`+/ A234`	`+/[1] A234`
`+/"2 A234`	`+/[2] A234`
`+/"1 A234`	`+/ A234`

For example, `+/"1` means apply `+/` with rank 1, i.e. as a vector function. Since this is necessarily the last axis, it must be that `+/"1` in J is the same as `+/` in APL.

Prefix Agreement

J also minimizes the need for axis specification with an extension to APL's scalar agreement known as *prefix agreement*. Arguments to a dyadic function agree if the frame of one is a prefix of the frame of another. In most cases, for 'frame' you can use 'shape' to the same effect (for further discussion, see the first article in this series). Thus the following works in J:

```
   M34 + 10*V3
11 12 13 14
25 26 27 28
39 40 41 42
```

Summary
In both APL and J, we can classify functions by the rank at which they apply. Function application is essentially the same in APL and J, except only that APL sometimes assembles results after first enclosing them. A major difference is that in APL you can modify function application using the axis operator, whereas in J you use the rank operator, which is a subset of the axis operator.

Although it appears that the APL model has more functionality, it is not fully implemented. Moreover, for a proper comparison, you must supplement the J model by taking into account the use of item functions and prefix agreement. In practice, the J model works well.

APL and J

5. Operators – Inner Product, Each, Commute, Compose

This series of articles entitled *APL and J* will try to explain the J way of thinking, by contrasting APL and J coding examples. The term APL will be used generically to refer to facilities found in most commercial implementations of APL.

To conclude the discussion of APL's operators, we now look at *inner product*, *each*, and two operators found only in Dyalog APL: *commute* and *compose*.

Housekeeping
As usual, we use the letters `V`, `M`, and `A` for vectors, matrices and arrays of any dimension, optionally followed by numbers to denote their shape. Thus `V` for any vector, `V5` for a vector of length 5, `M45` for a 4-by-5 matrix, `A245` for a 2-by-4-by-5 array. Where actual values are required, `V5` will be defined as `⍳5`, `M45` as `4 5⍴⍳20`.

APL terminology will be used, except where J terminology is required for clarity. Here, `⎕io←1` is assumed throughout.

Inner Product
Inner product in APL is called *dot product* in J, and both are denoted by the dot symbol, `f.g` for functions `f` and `g`. As may be expected, `f` is applied with reduce in APL, but not in J, where it must be given explicitly if required. Thus the following are equivalent:

```
   V3 +.× M34                          APL
38 44 50 56

   V3 +/ . * M34                       J
38 44 50 56
```

(Note that in J, a space is required before the dot – since otherwise the dot would be read as part of the previous symbol.)

Does `f.g` in APL always give the same result as `f/ . g` in J? Yes – if we look at typical APL uses of inner product, such as `+.×` or `∧.=`. However, the definitions are quite different, and nicely illustrate the two approaches to array operations. J also allows more possibilities, though I have yet to use a dot product in J that could not be written in APL.

In the following discussion, we apply dot product as: `X f.g Y` where the shapes of `X` and `Y` are `SX` and `SY`.

In APL, `¯1↑SX` must match `1↑SY`, and the computation is made by splitting `X` into vectors along the last axis, and splitting `Y` into vectors along the first axis, and applying `g` followed by `f/` to each combination of vectors from `X` and `Y`, and enclosing the individual results.

Since the individual results are enclosed, they do not contribute to the result shape, which must therefore be `(¯1↓SX),1↓SY`. The computation for each pair of vectors `x` from `X` and `y` from `Y` is:

```
    ⊂ f/ x g¨ y
```

For example, `M22 +.× M23` is evaluated as:

```
    (⊂ +/ 1 2 ×¨ 1 4) (⊂ +/ 1 2 ×¨ 2 5) (⊂ +/ 1 2 ×¨ 3 6)
    (⊂ +/ 3 4 ×¨ 1 4) (⊂ +/ 3 4 ×¨ 2 5) (⊂ +/ 3 4 ×¨ 3 6)
```

and assembled into the matrix:

```
  9 12 15
 19 26 33
```

In J, the right argument `Y` is not split up at all, while the left argument `X` is split up into cells of rank 1 plus the left rank of `g` (you can think of *lists* of left argument cells). The number of items in the left argument cells must match the number of items in `Y`. Therefore, in the common case where the left rank of `g` is 0 (e.g. where `g` is `*` or `=`), `X` is split up into vectors along the last axis just as in APL.

For example, `M22 +/ . * M23` is evaluated as:

```
   (+/ 1 2 * M23) (+/ 3 4 * M23)
```

which is:

```
   (9 12 15) (19 26 33)
```

which is then assembled into the matrix:

```
  9 12 15
 19 26 33
```

Note that this makes use of J's ability to multiply a vector by a matrix or arbitrary array, using prefix agreement.

The shape of the result of dot product in J is the shape of the frame of `X` (i.e. `SX` less the shape of the cells used in the dot product), catenated with the shape of the result cells. In the following example, let `f=. (i.10)"_` be the function that returns the list `i.10`, and `g=. 0"2` has left rank 2, then:

```
   $ (2 3 4 5 6$'a') f . g 4 5 $'b'
2 3 10
```

The key changes J makes are: the right argument `Y` is not split up at all; the left argument `X` is split up according to the left rank of `g`; and the results of each computation are not enclosed, but follow normal assembly rules.

Determinant and Permanent

In J, the dot product can be used monadically, when it is the *determinant* function, thus:

```
   -/ . * 2 2 $ 10 7 1 3
23
```

If + is used instead of -, the function is the *permanent*. Any other verb could be used in place of `-` or `+`, but I don't know of a useful example!

Each
The APL expression *f*¨ (*f* each) means apply *f* to each scalar in an array, opening each scalar before applying the function, and enclosing the result. Since each result is enclosed, it must be that (⍴*f*¨*X*)≡⍴*X*.

J has no built-in equivalent to the APL each operator. However, one can be constructed as a special case of the *under* conjunction `&.` , and this is defined in the J standard library. Given verbs `u` and `v` where the inverse v^{-1} is defined, then `u&.v` is equivalent to v^{-1} `u v`. Now, if `v` is the verb `>` (open), then its inverse is < (box), so that `f&.>` means open, apply `f`, then box, which is exactly the same as APL's *f*¨, for example:

```
    n=. 'winston';(i.3 4);10 20

   $ &.> n
+-+---+-+
|7|3 4|2|
+-+---+-+
```

So if we define `each` to be `&.>` , then this is an adverb whose definition and behaviour is exactly like APL's each operator:

```
    each=. &.>

    $ each n
+-+---+-+
|7|3 4|2|
+-+---+-+
```

J's rank operator `"` and APL's each operator ¨ are notationally similar, and should not be confused. For further discussion, see the first article in this series (Vector, July 1996).

Commute
In Dyalog APL, the *commute* operator ⍨ creates a new function which reverses its arguments, so that *X f*⍨ *Y* is equivalent to *Y f X*, for example:

```
    5 -⍨ 7
2
```

The corresponding operator in J is ~ , which reverses the arguments when the resulting function is used dyadically:

```
   5 -~ 7
2
```

Where the resulting function is used monadically, the argument is used as both left and right argument, so that `u~ y` is `y u y`, for example:

```
   #~5
5 5 5 5 5
```

Why bother with such a trivial operator? For one thing because it simplifies expressions by reducing the need for parentheses, and for another because it is useful in constructing tacit definitions, for example:

```
   firstword=. {.~ i.&' '
   firstword 'In the beginning'
In
```

Compose
In Dyalog APL, the *compose* operator ∘ takes two arguments which are either a function and one of its arguments; or both functions.

In the first case, an argument is bound to a dyadic function, creating a new monadic function:

```
      findblank← ⍳ ∘ ' '

      findblank 'one two three'
4
```

The J equivalent is `&` , for example:

```
   findblank=. i. & ' '
```

Joining a function with one of its arguments to create a new function is called *currying* in honour of Haskell Curry.

In the second case, two functions are composed, creating a new function `f ∘ g` which applies first `g`, then `f`. The function `g` is used monadically, while `f` may be monadic or dyadic.

Where `f` is monadic, `f ∘ g X` is equivalent to `f g X`, for example:

```
      f← +/ ∘ ⍳

      f 100
5050
```

The J equivalent for the monadic case is given by `@` or `&` (either will do), e.g.

```
   int=. 1: + i.            origin-1 integers

   f=. +/ @ int

   f 100
5050
```

Where f is dyadic, `X f ∘ g Y` is equivalent to `X f g Y`, for example:

```
      f← × ∘ ⍳

      10 f 5
10 20 30 40 50
```

This composition is the same as J's hook:

```
   f=. * int

   10 f 5
10 20 30 40 50
```

This example makes it clear that a hook could have been constructed as a normal operator, rather than a train of 2 verbs. The reason for making it a train is that it complements the fork, which is a train of 3 verbs, allowing resolution of trains of any length.

Note that where `f` is dyadic, then *f* ∘ *g* is not the same as J's `f @ g` or `f & g`. In these cases:

`X f @ g Y`	is	`f X g Y`
`X f & g Y`	is	`(g X) f (g Y)`