Mastering J

Donald B. McIntyre
Luachmhor, Church Road
Kinfauns, Perth PH2 7LD, Scotland
United Kingdom
Telephone: 011 44 738 86 726

Abstract

J is a new dialect of APL that provides much additional power over earlier dialects, and also provides a challenge to the user to master this new power. This paper discusses some of the problems the author encountered in learning J, and how they were overcome.

The Dictionary and J Literature

The power and interest of J come, not from its symbols, but from its extended grammar; especially from the syntax that permits composition of functions, both by conjunctions and directly in the new phrasal forms called "hooks" and "forks" (Iverson and McDonnell, 1989), that are explained with many examples in this paper.

The new dialect is defined in *The ISI Dictionary of J* (Iverson, 1990), which, though an essential reference, is not intended or suitable as an introduction. Iverson's *Tangible Math* (1990), the tutorials provided with the system, his paper in *Vector* (July 1990), and his *Programming in J* (1991) are tools to use when learning. The Dictionary is terse. The word means "effectively concise", which is surely a virtue (it is derived from the Latin for "polished"), but I was frustrated when I knew how to perform an operation in APL, yet searched the dictionary and tutorials in vain for the analogous procedure in J. In order to master a language one needs to read texts written at the appropriate level; it is not sufficient to study only a dictionary and grammar book. Because at this early stage the literature of J is meager, we need examples with annotations and commentary to help beginners.

The reader of this paper is assumed to have some familiarity with APL. Although many concepts incorporated in the new dialect should be evident to anyone browsing the paper, it is addressed primarily to those who have made the modest outlay of $24 for the J System and the booklet containing both the *ISI Dictionary of J* and *Tangible Math* (Iverson, 1990). APL is an interactive language, and any dialect of it is best explored with a working system on the desk.

The word "Mastering" in my title indicates a process taking place in the present – not one completed – and I assume that the reader wishes to join in the endeavor. I describe how some problems can be overcome, and I encourage others to start mastering J, a dialect whose generalization, consistency, power, and elegance will delight all who appreciate APL.

Reflections and Suggestions for Getting Started

The question may be asked: Is J a dialect of APL? I think the answer is simple. APL began as an executable form of Iverson notation; today J is the executable form of the latest version of Iverson notation. Many people have contributed to the evolution of commercially available APL systems, but the principal architect of the language is the man who published *A Programming Language* in 1962 and who today gives us the version he calls J.

This paper arose in the following way. Immediately after the APL90 meeting in Denmark, Ken Iverson spent three days with me in Scotland and proposed that I should convert to J various applications I had written while using and teaching APL. He suggested I should submit the results of the experience in a paper for APL91 with the title "Mastering J".

People coming to APL for the first time often protest about the strange symbols – an attitude that seems strange to anyone well accustomed to the language. It was therefore unsettling to discover that my first reaction to J was distress at encountering new symbols. I felt comfortable with the traditional APL symbols, and was uneasy about abandoning them. How much credit, then, is due to Ken Iverson, who with a greater stake than most of us in the old symbols, was the first to recognize that it was the idea – not the graphic symbol – that mattered. As Giuseppe Peano wrote in "The Importance of Symbols in Mathematics" (Peano, 1915): "Algebraic equations are much shorter than their expression in ordinary language, are simpler, and clearer, and may be used in calculations. This is because algebraic symbols represent ideas and not words. ... Algebraic symbols are much less numerous than the words they allow us to represent. ... The symbols of logic too are not abbreviations of words, but represent ideas, and their principal utility is that they make reasoning easier. All those who use logical symbolism attest to this."

After an initial hesitation, I adopted the new symbols and found no difficulty at all in using them. I now find that I tend to type `=.` instead of the assignment arrow when using older versions of APL! Like the old APL symbols, the new ones have been selected with a discrimination that is remarkable. The Language Summary, included in *A Dictionary of J*, is the essential key that needs to be at hand for quick reference until usage results in familiarity. When some symbols were changed with the introduction of Version 3, I had no trouble in making the mental conversion, and my word processor's global "search and replace" command brought the text of my old J files up to date in seconds.

I must admit that my initial problems nearly made me abandon the enterprise, but some of the difficulties will not be encountered by others because they came from bugs in an early implementation (e.g. "grade") and some typographical errors in the text – all of which have long since been corrected. Not being a theoretician I found some of the Dictionary definitions too terse, and the biggest problem was probably the small number of examples then available. It is my hope that I can contribute to the fund of examples without adding too many typographical errors.

Each of the 81 "words" in the language has a unique symbol. Because the symbols have been ingeniously chosen for their mnemonic value, after a little practice their use is effortless. To begin with, it may be an advantage to assign and use names such as

```
power=. ^
rotate=. |.
mean=. m.
gradeup=. /:
```

or, for inner product,

```
ip =. +/ .*
```

Because each symbol is either a single character alone or a single character followed by a period or a colon, in the definition of the inner product a space must separate the slash (`/`) from the period that follows. When standing alone, the period and the colon are symbols for the conjunctions "dot product" and "explicit definition". It follows for the same reason that the following will give "nonce" errors:

```
1 2 3 4 .5 6 7
3 + .5
```

J's interpretation of `.5` is easily demonstrated:

```
x=. .5
x
```

While no space is needed to produce the list of the first 5 integers (origin zero, of course):

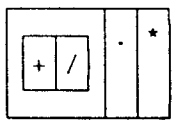
```
i.5
```

I have found that inserting spaces in such expressions improves readability; for example, to assign the integers `0 1 2 3 4` to `i` I would insert two spaces (not required) and write:

```
i=. i. 5
```

If phrases are first named and the names are then used to build more complex sentences, spaces (and parentheses) that would otherwise have been required are not needed:

```
x=. .*
sum=. +/
ip=. sum x
ip
```




```
1 2 3 ip 3 4 5
26
```

J is less permissive than older APL: J (at least in Version 3) does not permit the transpose of a scalar or the addition of a vector of length 1 to a longer vector. The following expressions produce a "length" error:

```
|:5
(,5) + i. 4
```

Because many verbs perform the same function as their equivalents in older APL, much of J is immediately comprehensible to anyone with APL experience; for example to generate and display 10 random integers in the range 0-99:

```
]y=. ?10$100
13 75 45 53 21 4 67 67 93 38
```

and the sorting permutation is given by "grade up":

```
   /:y
5 0 4 9 2 3 6 7 1 8
```
But J includes useful natural extensions to familiar functions. For instance the sorted values can be obtained by the dyadic form:
```
   y/:y
4 13 21 38 45 53 67 67 75 93
```
We can illustrate the use of the adverb "both" (~) in the verb "sort"; when a right argument alone is given, "both" ensures that there is an identical argument on the left:
```
   sort=. /:~
   sort y
4 13 21 38 45 53 67 67 75 93
```

Some of the new verbs perform well-known functions that were not available before as primitives; for example:

increment and decrement	>:	<:
double and halve	+:	-:
square and root	*:	%:
nub and nub sieve	~.	~:

The bracket notation for indexing has long been recognized as anomalous, because it does not correspond to the syntax of other functions. In J it is replaced by the dyadic function "find" ({). We can use it to generate data to illustrate "nub" and "nub sieve". The adverb "cross" (~) interchanges the left and right arguments, thus avoiding the need for parentheses:
```
   ]y=. 'abcde'{~?12$4
cacbcddbacbc
   sort y
aabbbccccdd
   ~.y
cabd
   ~:y
1 1 0 1 0 1 0 0 0 0 0 0
```

APL has had adverbs and conjunctions from the beginning, though it was some time before they were recognized for what they are (see Backus, 1978). J not only adds new adverbs and conjunctions to APL, but permits the user to define them as easily as defining nouns and verbs. One of the most important conjunctions to understand early is the "rank" conjunction ("). The following examples may be helpful supplements to the information in the dictionary:

Define a rank-3 array:
```
   ]a=. i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```
The shape is:
```
   $a
2 3 4
```
The rank is the number of "atoms" in the shape:
```
   #$a
3
```
and the number of "items" in the array is given by the first atom in the shape:
```
   {.$a
2
```
The number of items is given more directly by "tally" (#):
```
   #a
2
```
The rank-2 cells have the shape 3 4, and their "frame" is the rest of the shape vector; that is, 2. The rank-1 cells have the shape 4 and their frame is 2 3.

Reverse the (two) items:
```
   |. a
12 13 14 15
16 17 18 19
20 21 22 23

 0  1  2  3
 4  5  6  7
 8  9 10 11
```
This is equivalent to "reverse rank-3", and could be stated explicitly:
```
   |."3 a
```
Reverse the rank-2 cells:
```
   |."2 a
 8  9 10 11
 4  5  6  7
 0  1  2  3

20 21 22 23
16 17 18 19
12 13 14 15
```
Reverse the rank-1 cells:
```
   |."1 a
 3  2  1  0
 7  6  5  4
11 10  9  8

15 14 13 12
19 18 17 16
23 22 21 20
```
Because "reverse" is monadic, its rank refers to the cells of the single (right) argument. Rank permits the application of a verb to its argument in different ways without the introduction of other verbs. In older versions of APL this was accomplished by placing an axis "operator" in brackets immediately after the function to which it referred.

When the rank conjunction applies to a dyadic function, such as "Append" (,), the ranks of both left and right arguments can be specified, but a single value will be taken to apply to both.

```
   x=. i. 5
   y=. i. 5 3
```

Append the 0-cells of x to the 1-cells of y

```
   x,"0 1 y
0  0  1  2
1  3  4  5
2  6  7  8
3  9 10 11
4 12 13 14
```

The frame of the 0-cells of x is of rank 1, and so is the frame of the 1-cells of y. When the rank conjunction has a negative number, this indicates that its magnitude (absolute value) is the rank of the frame; thus we can get the same result by writing:

```
   x,"_1 y
0  0  1  2
1  3  4  5
2  6  7  8
3  9 10 11
4 12 13 14
```

Suppose now that x is rank-2:

```
   x=. i. 5 2
```

Then appending the rank-1 cells of x to the rank-1 cells of y:

```
   x,"1 1 y
0 1  0  1  2
2 3  3  4  5
4 5  6  7  8
6 7  9 10 11
8 9 12 13 14
```

which could be written `x,"1 y` or `x,"_1 y`

A common application of the conjunction "with" (&) is to attach an argument to a function. For example the verb

```
   f0=. ,&0"1
```

appends 0 to the rank-1 cells of its argument.

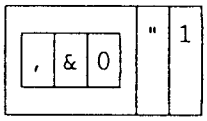
Thus applying the verb f0 to an array a:

```
   a=. i. 3 2 4
   f0 a
 0  1  2  3 0
 4  5  6  7 0

 8  9 10 11 0
12 13 14 15 0

16 17 18 19 0
20 21 22 23 0
```

The verb f0 is a specialized monadic function whose (right) argument is the left argument of append (,). A conjunction grabs whatever is to its right as its right argument and takes the entire verb phrase to its left as its left argument. The conjunction "with" (&) is necessary to attach 0 to the right of append (,), but "rank" (") is itself a conjunction and therefore takes the 1 to its right as its right argument without any further assistance.

Much distress can be saved by making defining verbs in short phrases, which can be grouped together later as a complex sentence. As each verb is defined, its boxed display should be examined to verify that the interpretation is as intended. The boxed display is an extremely useful tool for mastering J at every level. In this case:

```
   f0
```

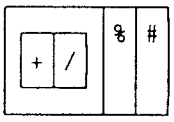


If we define the verb mean:

```
   mean=. +/%#
```

at first glance its display looks rather like the verb f0

```
   mean
```



but the two could hardly be more different.

The function f0 (we use the terms function and verb synonymously) is the verb "append" (,) in which the right argument (0) is built-in by the conjunction "with" (&), and it is to apply to the rank-1 cells (this is an upper limit) of the argument of f.

The function "mean" consists of three verbs, one of which is modified by the adverb "insert" (/). A sequence of three verbs would have no meaning in older forms of APL, but in J the composition of three functions is called a "fork". In the monadic case, which we have here:

```
   (f g h) y  is  (f y) g (h y)
```

thus:

```
   (+/%#) y   is   (+/ y) % (# y)
```

It is worth observing here an important distinction between J and older versions of APL. Whereas in the old versions the default axis was the last one, in J it is the first axis. Consequently a simple function for the arithmetic mean was likely to be written to apply to a vector and not to a matrix argument; the introduction and use of "tally" (#) reverses this situation in the sense that it makes the function mean work on the "items" of the array; so that it will give a valid result however high a rank the array may have.

The deviations from the mean are given by:

```
   y - (mean y)
```

This is likewise a common construction which is the composition of two functions "minus" and "mean". J supports this composition of two functions as a "hook". In

the monadic case we have:

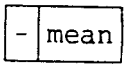
```
   y f (g y)  is  (f g) y
```

thus:

```
   deviations=. - 'mean'~
```

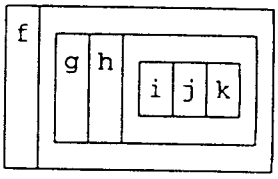
By placing the name of the verb mean in quotes and using the adverb "evoke" (~) the function deviations will use the current definition of mean and display the pro-verb (or name of the verb) in boxed display:

```
   deviations
```



The construction of hooks and forks having been implemented, J provides a meaning for the composition of any sequence of verbs. Thus if the verbs f, g, h, i, j, k have been defined then (f g h i j k) is f (g h (i j k))

```
   f g h i j k
```



Taking test data y:

```
   ]y=.?14#100
13 75 45 53 21 4 67 67 93 38 51 83 3 5
```

Given the verbs squares and sum

```
   squares=. *:
   sum=. +/
```

We can get the sum of squares of the deviations (from the mean) in desk calculator mode:

```
   sum squares deviations y
12399.7
```

In order to define the verb ss we cannot simply write:

```
   ss=. sum squares deviations
```

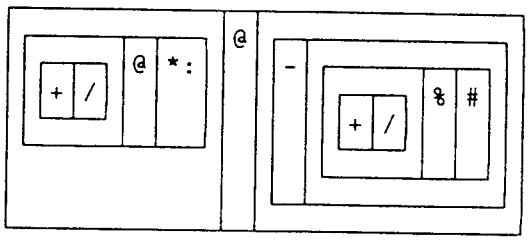
because this would be a fork. We need the conjunction "Atop" (I think of it as "After"), whose symbol is @

```
   ss=. sum @ squares @ deviations
   ss y
12399.7
```

Parentheses are not required around the component pro-verbs even if they are needed around the definitions of their defining expressions. The result is identical whether or not pro-verbs have been used in the definition – there is therefore no penalty in execution, yet with pro-verbs the definition can be more readable.

In building a definition we may choose to define its components by using simple pro-verbs, such as f, g, h. Once the verb has been defined it no longer needs these pro-verbs and will not be changed should the proverbs f, g, h be used for some other purpose.

```
   ss
```

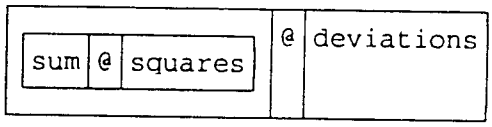


The boxed display contains the definitions of each of the three verbs that compose it. When the embedded verbs are complex, we may "evoke" their current definitions in the following way:

```
   ss=.'sum'~@('squares'~)@('deviations'~)
   ss y
12399.7
```

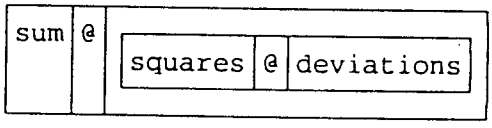
An advantage of this form of definition is that the boxed display does not swamp the reader with detail – it assumes that you know the meaning of sum, squares, and deviations. When executed the function will evoke the current definitions of the pro-verbs.

```
   ss
```



One might well argue that the component verbs are to be executed one "atop" ("after") the other; that is, the deviations should be squared before sum is activated. To achieve this we must write the definition in the more cumbersome form:

```
   ss2=.'sum'~@('squares'~ @ ('deviations'~))
   ss2 y
12399.7
   ss2
```



Although I find that the sequence f @ g @ h – which is (f @ g) @ h – usually produces the same result as f @ (g @ h) I am not able to provide a formal explanation of the conditions that require the insertion of parentheses. If the result of f @ g @ h is not what you intend, you may need parentheses.

The function ss could, of course, have been written using primitives alone:

```
   ss=. +/ @ *: @ (- +/%#)
   ss y
12399.7
```

Only necessary parentheses are included, but extra spaces are inserted to improve readability. The parentheses are needed because each "atop" grabs whatever is immediately to its right. If the rightmost "atop" takes the minus it would

prevent the construction of the fork that defines the deviations.

Notice that this is a purely functional definition; that is, no direct reference to the argument appears. This form of definition is called "tacit" definition.

A definition in "explicit" form is as follows:

```
   ss=. '+/ *: (- (+/ % #)) y.' : ''
   ss y
12399.7
```

The verb ss is here defined specifically as a monadic function; the dyadic definition is empty. Parentheses are used to produce the necessary compositions, but the only conjunction needed is "define" (:). The argument is given explicitly (y.)

If parentheses are not used, the sentence is treated as a sequence of functions to be executed one after the other as in older forms of APL

```
   ss=. '+/ *: - (+/ % #) y.' : ''
   ss y
12399.7
   ss=. '+/ *: - +/ % # y.' : ''
   ss y
0.00510204
```

Inherited Rank

If f and g are functions then when f @ g is executed, f will "inherit" the rank of g; that is, the result of the function g has rank equal to the rank of g.

In early versions of J this meant that in execution of sum@squares, sum inherited rank 0, with the result that it appeared that the verb sum did nothing. The user needed to specify an arbitrary high rank for squares. To avoid this, Version 3 gave an infinite rank to functions such as squares.

Grade (/:), however, is not a scalar function, and consequently it will fail to produce the result we want if it inherits rank-0. If x is a 12 by 5 numeric array of random numbers (range 0 through 99), then the permutation of rows required in order to arrange a given column into ascending order is given by the verb colgrade. In the example, the verb { ("From") would select row 3 had we not specified that the 3 is to refer to rank-1; that is, take column 3 from x. But to ensure that "grade" inherits a higher rank than "from" would otherwise pass to it, we have added the additional specification of rank-9. The final line will take the items (rows) from x in the order necessary for column 3 to become sorted into ascending order.

```
   x=. ?12 5$100
   colgrade=. /: @ ({"1"9)
   3 colgrade x
   (3 colgrade x){x
```

colgrade will not work if it is defined in the obvious way as:

```
   colgrade=. /: @ ({"1)
```

Using Version 3 we can demonstrate the (surprising) effect of inherited rank by giving squares rank 0 (as it had in earlier Versions)

```
   x=. ?12 5$100
   squares=. ^&2"0
   ss=. +/ @ squares
```

When ss is applied to x the result is that, although each atom in x is squared, there is no visible evidence of any summation – the summation took place over the rank-0 cells!

Give squares a sufficiently high rank and the problem disappears:

```
   squares=. ^&2"9
```

In Version 3 there is no need to specify this high rank because the verb "exponential" (^) is implemented with infinite rank.

The problem of "inherited rank" (for example, in computing sums of squares) gave me days of anguish and frustration before Ken Iverson explained the underlying principle.

We have been describing J as one-line programs, which should be rather familiar to those who have used Iverson's Direct Definition in earlier versions of APL (Iverson, 1976). Functions in explicit form can, however, be defined as a sequence of sentences with labels to facilitate branching. Space does not permit giving examples of multi-line functions, but the following hint may be helpful. Be sure that each statements in an explicit definition is boxed. I usually define a series of lines some of which are single sentences while others consist of two or more sentences collected together with link (;). Those that are linked are already boxed, but it is easy to overlook a line that is a single sentence perhaps consisting of phrases joined together by the verb "left" ([), which those familiar with Direct Definition in older APL probably simulated with a dyadic function, with a name such as "where" or "after", that returned its left argument. Lines like this must be explicitly boxed (<).

Other conjunctions worthy of early examination, but which for lack of space cannot be discussed here, include:

power	^:
cut	;.
under	&.
fit	&:

Some functions familiar to an APL programmer may be overlooked because in J they appear with a slight disguise. An example is compression (which is a special case of the more general function "replicate"):

```
   v=. i. 15
   u=. 15$1 0
   u # v
0 2 4 6 8 10 12 14
   m=. i. 5 5
   u=. 5$ 1 0
   u#m
 0  1  2  3  4
10 11 12 13 14
20 21 22 23 24
   u#"1 m
 0  2  4
 5  7  9
10 12 14
15 17 19
20 22 24
```

After I had searched in vain for APL's familiar "expand" function, Ken Iverson wrote me as follows (November 15, 1990):

For expansion the simple solution is:

```
   x=.6 7 8
   u.=1 0 1 0 0 1 0
   u*+/\u
1 0 2 0 0 3 0
   (u*+/\u){0,x
6 0 7 0 0 8 0
```

However, the following is more general:

```
   expand=./:@\:@[{#@[{.]
```

Try `v expand b` with `v=.1 0 1 0 1` and `b=.6 7 8`, but also with cases such as `b=.i.3 4` and `b=.i.3 4 5` and `b=.3 4$'abcdef'` and `b=.3 4 5$'abcdef'`.

Notice how this tacit definition reads in English: The inverse of the downgrade of the left argument permutes the (over)take of the right argument by the number of items of the left argument.

You might also look at the general case of re-merging the results of compression:

```
   a=.'abcdefg'
   ]b=.u#a
acf
   ]c=.(-.u)#a
bdeg
   (/:\:u){b,c
abcdefg
```

Anyone who still doubts that J is indeed a dialect of APL should note that the last example is the "mesh" defined by Iverson (1962, p. 20): "If, for example, a = (s,e,k), b = (t,a), and u = (0,1,0,1,0), then \a,u,b\ = (s,t,e,a,k)."

In APL we are accustomed to extract the value at row *I* and column *J* of a matrix *M* by using the syntax *M[I;J]*. Because J omits the anomalous bracket notation, we use the verb "from" ({) instead:

```
   m=. i.6 6
   (<3 4){m
22
```

To change the value in this cell supply its index as the left argument of the adverb "amend" ({}), and use this derived function with x, the replacing value, as left argument, and the array m containing the atom to be replaced, as right argument:

```
   x=. 100
   x (<3 4)} m
 0  1  2  3   4  5
 6  7  8  9  10 11
12 13 14 15  16 17
18 19 20 21 100 23
24 25 26 27  28 29
30 31 32 33  34 35
```

To place the elements of matrix x in cells at rows 2 4, and columns 2 4 of the square matrix m, we use the syntactical form:

```
   x i } m
```

where i is the array of indices of the cells in m to be amended:

```
   i=. <"(1) 2 4,"0/ 2 4
   x=. 100 101,:102 103
   x i } m
 0  1   2  3   4  5
 6  7   8  9  10 11
12 13 100 15 101 17
18 19  20 21  22 23
24 25 102 27 103 29
30 31  32 33  34 35
```

This form is completely general and allows complete freedom in choosing the cells to be amended, unlike standard APL, which permits replacing only rectangular sections of the argument. Try the same expression with `i=. ((<5 3),<2 1),:(<0 3),<1 1.`

At any early stage in my attempts to master J, I wrote to Ken Iverson for help with the problem of sorting a boxed array. Although able to produce the required result, I could not generalize the procedure to my satisfaction. In reply I received the following letter, dated September 18, 1990, from Eugene E. McDonnell. It is a remarkably lucid and helpful account of certain aspects of programming in J, and, with his permission, I am happy to share it with others.

I was visiting Ken last week when your letter to him concerning a problem you were having with J arrived. I hope you don't mind my chiming in about it.

Your problem was to arrive at a sorted version of a boxed matrix, based on the values in the open of one of the columns

of the matrix. You were able to obtain the required permutation vector, but couldn't see how to apply this properly to the matrix.

How does one tackle such a problem? One way is to begin at the root: what is the last thing to be done? In your problem it is to reorder the rows. In APL\360 one would think of writing *M*[*p*;], where *p* is the desired permutation. In SAX, one would probably think of using the "from" function, and write *p*{*M*. If you walk into the tool crib in a J factory and look around at what tools are available, you will come to the "sort" verb.

In J, the verb "`a sort b`" is defined by `(grade b) from a` ; that is, `a` is sorted into an order specified by `b`, and so "sort" combines the aspects of both determining and applying the permutation (the verbs "sort" and "grade" are the dyadic and monadic forms of the `/:` verb). In your case the thing to be sorted is the open of the column of the argument given by `a`, so in using J one might well think of writing

```
b/: > a {"1 b
```

If we write this in the function definition form that uses the explicit mention of the arguments, we could write

```
'' : 'y./: > x. {"1 y.'
```

However, you were trying to exploit the form of definition in which functional values only appear. I began calling these "pure functions" a while back, but Ken was not satisfied with this and we discussed possible alternatives, at last resolving on "tacit form" for this, as a counterpart to the term "explicit form" for the other. At any rate, how does one go about translating from the explicit form to the tacit form?

After using them for a while, the fork and the hook idioms can be found to occur almost everywhere, so I now expect to be able to use one or both of these in handling any problem. I find myself looking at an expression like `y./:> x.{"1 y.` and trying to put it into fork form. There are two problems with what we have. The first is that, instead of having something in the form `(x f y) g (x h y)`, the part on the left is `y.` by itself; the second is that the part on the right is `> x. {"1 y.`, and while this is a function of two arguments, it is not immediately apparent that it can be transformed into the required dyadic functional form.

Let's solve the first problem first. By now I am familiar with a simple device which permits a ready solution. Whenever we have just one argument to a dyadic function, we can replace it with lev or dex, as appropriate. In our case we have `y.` (the right argument) by itself, so we can replace it by

```
x. dex y.
```

(that is, `x. ] y.`).

The second problem is a bit trickier. We have

```
> x. {"1 y.
```

which, looked at abstractly, is in the form `f x g y`. This abstraction helps us to recognize it as the definition of the `@` conjunction, that is, `x f@g y` is `f x g y`. In our case, `f` is simply `>`, and `g` is `{"1`. We might be tempted to create `>@{"1`, but this would be wrong, since `@` is a conjunction and the parsing rules would cause `@` to grab `{` as its right argument, giving us `(>@{) "1`, whereas we want the entire derived function `{"1` to be the right argument to `@`. So we must put `{"1` in parentheses to achieve this: `>@({"1)`, which is a function derived from two forms of composition, `@` and `"`.

As an aside, I should point out that the box drawing characters used in the display of a tacit function give a wonderful visual clue to the parsing of an expression. Study the difference in the boxing structure of `>@{"1` and `>@({"1)` to see this.

We now know how to write the tacit form of our function as a fork:

```
] /: >@({"1) .
```

We can now define two functions to do the "sort matrix on column" problem, one in the explicit form, and the other in the tacit form:

```
SOCe=. '' : y. /: > x. {"1 y.'
SOCt=. ] /: >@({"1)
```

If you experiment with these two forms on your sample matrix, I wouldn't be surprised if your experience was like mine, and that `SOCt` was almost twice as fast as `SOCe`. I've discussed this with Roger Hui, and his conjecture agrees with mine: the tacit form is completely preparsed, whereas the explicit form requires parsing when it is used. When I showed the display of a tacit form to a colleague recently, he said "It's magic." They seem to me, too, magical, and I find myself thinking of programming in a completely different way because of it.

I have found that when I have a difficulty in creating a function in tacit form, if I revert to the more conventional explicit form as in this letter, it is then usually straightforward to convert the explicit form to the tacit form. In fact, I've wondered whether it might not be possible to do this automatically. In any event, I recommend that until you are more at home with the tacit form, you might consider the methods of this note.

External Conjunctions

An Appendix in the Dictionary lists a number of strange looking conjunctions. Despite one's first impression, these are perfectly ordinary conjunctions (based on `!:`) that provide communication with the system. Here are some examples:

```
read=. (1!:1)&<
```

the file `util.in` on a given path can be read by J with the sentence

```
x=. read 'c:\s\j3\util.in'
```

The content of the file is now available in `x` for processing by J functions. In this way data can be read from ASCII files. The results of processing can, of course, be written as ASCII files in an analogous manner.

The random link can be set to a given value by:

```
setrl=. 9!:1
setrl 16807
```

The random link can be queried by `qrl`, but it must be given an argument even although the argument will not be used:

```
   qrl=. 9!:0
   qrl 1
16807
```

Use of)script and a Word Processor

I recommend that whenever you enter J you capture the session by using the command:

```
   )script - 'session'
```

where - specifies input from the keyboard and the name (which must be in quotes) specifies the path (if desired) and file identification for output.

After leaving J your Word Processor (I use XyWrite) can be used to edit the output file and prepare a script (as in a play or a broadcast) for further input. Errors can be corrected, new data or procedures can be entered outside of J and the file use for input by a command in the form:

```
   )script 'input' 'output'
```

It is important to remember that once the input file has been read the output file is automatically closed. If you wish to record the results of further work you must then specify that further input is to come from the keyboard; for example:

```
   )script - 'continue'
```

I have found it convenient to annotate my input files with comments by using the verb `cc` (`c` by itself occurs too often as a pro-x, but you might use `C`):

```
   cc=. 0 0&$
```

For example you might want always to include the following comment at the end of your input file:

```
  cc 'Remember to specify'
  cc ')script - ''outfile'''
  cc 'before continuing'
```

If you do not need to display the input file, you can use "silent script" as in this example:

```
   )sscript 'eigen.fns'
```

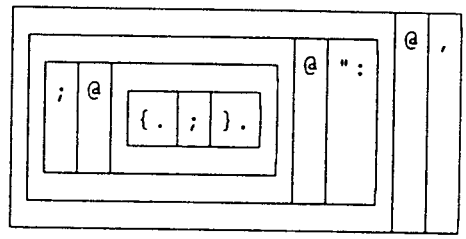
Good Examples

Iverson's utility functions, for example in his "Tangible Math", provide good material for analysis. Here are examples:

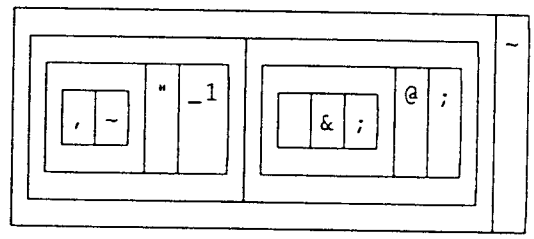
```
   over=. ; @ ({. ; }.) @ ": @ ,
```

Remember that tacit functions are ambivalent. You do not know how many arguments are to be used until you see the function used. This one is written for dyadic use, so the rightmost verb is "append" and not "ravel".

```
   over
```




```
   by=. (,~"_1 ' '&; @ ;)~
   by
```



You might find it easier to understand the rank specifications in this version:

```
   by2=. (,~"1 0 ' '&; @ ;)~

   table=. 1 : '[ by [ over [ x./ ['
   + table 3 4 5 6
```

	3	4	5	6
3	6	7	8	9
4	7	8	9	10
5	8	9	10	11
6	9	10	11	12

Experiment by leaving out the format (`":`) in "over" to see what happens:

```
   over2=. ; @ ({. ; }.) @ ,
   table2=.1 : '[ by [ over2 [ x./ ['
   + table2 3 4 5 6
```

	3	4	5	6
3	6	7	8	9
4	7	8	9	10
5	8	9	10	11
6	9	10	11	12

Note: the syntax of defined adverbs was evolving as this paper was being written. The form is now `s : 1` instead of `1 : s`.

Acknowledgements

Ken Iverson created the notation and developed it during a period of over 30 years. He has helped me with APL since 1969. He went out of his way (literally) to introduce me to J and never failed to give counsel and advice as I worked on the applications on which this paper is based. Although he suggested I should record my experiences for APL91, he is, of course, not responsible for the errors and stylistic failures displayed. I have a long way still to go to Master J.

Gene McDonnell not only wrote me the valuable letter included in this paper, but has shepherded the paper for the Proceedings. Without his help I could not have contributed to APL91.

References

Backus, John, "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. 1977 Turing Award Lecture." *Communications of the ACM, Vol. 21*, No. 8 (1978) pp. 613-641.

Iverson, Kenneth E., *A Programming Language*, John Wiley and Sons, Inc. New York (1962) 286pp.

Iverson, Kenneth E., *Elementary Analysis*, APL Press, Swarthmore, Pennsylvania (1976) 219pp.

Iverson, Kenneth E., and Eugene E. McDonnell, "Phrasal Forms", *Conference Proceedings APL89*, ACM, New York (1989), pp 197-199 (this publication is identical to *APL Quote Quad 19*, 4).

Iverson, Kenneth E., "J", *Vector*, Volume 7, No. 1 (July 1990) pp 68-78.

Iverson, Kenneth E., *The ISI Dictionary of J*, Iverson Software, Inc., Toronto (1990).

Iverson, Kenneth E., *Tangible Math and the ISI Dictionary of J*, Iverson Software, Inc., Toronto (1990).

Iverson, Kenneth E., *Programming in J*, Iverson Software Inc., Toronto (1991) (includes *The ISI Dictionary of J*) 100pp.

Peano, Giuseppe, "The Importance of Symbols in Mathematics", originally published in Italian, *Scientia Vol. 18* (1915) pp. 165-173. English translation in: *Selected Works of Giuseppe Peano*, with a biographical sketch and bibliography by Hubert C. Kennedy. George Allen and Unwin, London (1973), pp. 227-234.