Appendix

Alfred Thompson, Jr.
almacro@acm.org

May 8, 2025

This technical note is a "port" of the "Introduction to APL" appendix of the text Introduction to College Mathematics with A Programming Language *by E. J. LeCuyer, to the APL-derived J language.*

1 Introduction to J

We now discuss some of the fundamental features of the J language. To begin with, there are (at least) two types of data in J: literal data and numerical data.

1.1 Literal data

Literal data are enclosed in quotes. For example:

```
   'Welcome to APL'
```

If this is typed by the programmer, followed by pressing the RETURN key, the computer will respond

```
Welcome to APL
```

1.2 Numerical data

The J console can be used to do computations. For example:

```
   5+3
8
```

To add 5 to 3, enter 5+3 and press RETURN. The computer responds with the answer 8.

Other computations are done similarly. For example:

```
   3-5
_2
```

Observe that the symbol for subtraction is distinguished from the negative symbol in J.

1.3 Assignment

The symbol `=.` is used to assign a name to a value (*local assignment*). Consider the following examples:

`  a=.5`	The name a is assigned to 5.
`  a`	A request for the value of a.
`5`	The computer responds with 5.
`  b=.7`	
`  b`	
`7`	
`  a-b`	The value of $a - b$ is requested.
`_2`	
`  b=.a-b`	b is now assigned the value of $a - b$.
`  b`	The request for b.
`2`	Notice that the computer responds with the latest value of b.

The computer retains only the latest value of any name.

Global assignment uses `=:` in place of `=.` and makes the assignment at the top-level of scope.

1.4 Assignment of literal data

Literal data may also be assigned to a name.

`  c=.'welcome'`	Literal data are assigned names by enclosing the characters in quotes.
`  d=.'to'`	
`  e=.'j'`	
`  c,d,e`	The comma is used in J to chain data end to end. Notice that the computer didn't space the words as one would probably like.
`welcometoj`	

In order to get the correct spaces between the words, there are two possibilities:

`  c=.'welcome '`	Leave a space between the last letter and the end quotes.
`  d=.'to '`	
`  e=.'j'`	
`  c,d,e`	Spaced correctly.
`welcome to j`	

Or:

```
c=.'welcome'
d=.'to'
e=.'j'
c,' ',d,' ',e                Tell the computer to leave spaces by
                             typing a space between the quotes.
'welcome to j'
```

1.5 Mixing literal data and numerical data

Literal and numerical data may be displayed on the same line, provided that they are separated by semicolons. For example:

```
x=.3                         x and y are numerical data.
y=.5

l=.' is less than '          l is literal data.

x;l;y                        Numerical and literal data are dis-
                             played in a table.
```

```
┌─┬──────────────┬─┐
│3│ is less than │5│
└─┴──────────────┴─┘
```

1.6 Order of Operations

In conventional mathematics, we are accustomed to a hierarchy of numerical operations. Therefore, if we are presented with the expression $3 \times 2 + 5$, we would probably multiply 3 by 2 and then add 5, obtaining 11. We are used to a rule that multiplication is done before addition. However, if we were to enter this expression in a J console, we would obtain the result 21. This is because in J there is no hierarchy of operations. J has too many operations to make this desirable. The rule in J is to perform the rightmost operation first and then proceed from right to left. Thus the addition $2+5$ is done first, followed by the product 3×7. This "right to left rule" holds regardless of the operations. The only rule that takes precedence over this right to left rule is that operations within parentheses are done first as they are encountered in going from right to left. Thus $(3 \times 2) + 5$ would yield 11.

The expression $5 \times 3 + 3 \times 4$ yields 75 in J, since in J it is equivalent to $(5\times(3+(3\times 4)))$. In order to get the answer of 27, we would have to insert parentheses as follows: $(5\times 3)+(3\times 4)$. This right to left rule can cause the beginning student problems if she is not careful. She should be sure that the expression she is entering is really the expression she wants evaluated. When in doubt, she should insert parentheses. Consider the following examples:

```
  3+4*2
11
  2^3+2
32
```

`^` is the exponentiation operation in J. `2^5` is 2 raised to the 5th power.

```
  (2^3)+2
10
  10-3^2+1
_17
  10-(3^2)+1
0
  1+i.5
1 2 3 4 5
```

Note: `i.N` yields the vector of non-negative integers from 0 to $N-1$.

```
  (i.4)+2
2 3 4 5
```

1.7 Monadic and dyadic functions

J often uses the same symbol in two ways; one monadic and the other dyadic. A *monadic function* has only one argument. In J, the argument always appears to the right of the function. For example:

```
  ÷4
0.25
```

The reciprocal of 4.

```
  |_5
5
```

The absolute value of `_5`.

A *dyadic function* has two arguments; one on each side of the function. For example:

```
  2 ÷ 4
0.5
```

The division function.

```
  3|7
1
```

The residue function. It returns the remainder when 7 is divided by 3.

1.8 The "quad" and the "quote quad"

In APL, the "quad" is used to request input from the person using the computer. The J console uses a *foreign function* to collect interactive input. This is illustrated below:

```
  a=.". 1!:1]1
```

The user is allowed to enter any data she desires for *a*. The foreign function invocation redirects keyboard input to the console so that is collected

```
                                into a value and assigned to the name.
  5                             She has decided to enter 5.
  a                             A request for the value of a.
5
  a+". 1!:1]1                   The user is asked to add the nuber of
                                her choice to a.
  2                             She chooses to add 2.
7                               The result is 7.
```

As seen in the text, this invocation of the J foreign function equivalent to APL's "quad" operation is very useful in writing programs in which the user is to interact with the computer. It is also possible to allow a student to enter literal data of her choice using the J foreign function in a form equivalent to APL's "quote quad". This form of invoking the foreign function is simpler in that the conversion from literal to numerical data (using `."`) is not needed. The following example illustrates this usage.

```
  a=.1!:1]1                     A request for literal data to be named
HELP                            a.
  a                             a has been assigned the word HELP.
HELP
  b=.1!:1]1                     A request for literal data to be named
ME                              b. b has been assigned the word ME.
  a,' ',b                       The computer is requested to print
HELP ME                         out a space b.
```

1.9 Reduction

If α is a function and v is a vector, then the notation α/v is called reduction. It reduces v to a single number by applying the operation α to the successive elements of v (from right to left). For example:

```
  v=.1 2 3 4                    This is sum reduction. The elements
  +/v                           of v are added.
10
  */v                           This is times reduction. The elements
24                              of v are multiplied.
```

1.10 Compression

If a is a vector consisting entirely of 0's and 1's and if v is another vector, then the notation `a#v` is called compression. The result of `a#v` is that the elements of v corresponding to the 1's in a are kept, while the elements in v corresponding to the 0's in a are deleted. In general, a and v must have the same number of elements. For example:

```
   0 1 0 1 1 0#3 4 7 8 2 9
4 8 2
   1 0 1 1 0#'APPLE'
APL
```

1.11 Outer product

If a and b are vectors and α a dyadic function, an expression of the form aα/b is called an outer product. The result is an array or matrix obtained by performing the function α on every pair of elements of a and b. Consider the following examples:

```
   a=.1 2 3
   b=.4 5 6
   a +/ b                     Each element of a is added to each
                              element of b.
5 6 7                         1 is added to 4,5, and 6.
6 7 8                         2 is added to 4,5, and 6.
7 8 9                         3 is added to 4,5, and 6.

   a */ b                     Each element of a is multiplied by
                              each element of b.
 4  5  6                      1 is multiplied by 4,5, and 6.
 8 10 12                      2 is multiplied by 4,5, and 6.
12 15 18                      3 is multiplied by 4,5, and 6.

   a =/ a                     Each element of a is compared to each
                              element of a.
1 0 0                         1 is compared to 1,2, and 3.
0 1 0                         1 is compared to 1,2, and 3.
0 0 1                         1 is compared to 1,2, and 3.
```

If two numbers are =, a 1 is printed. 1 can be interpreted to mean "true" and 0 "false."

1.12 Inner product

If a and b are arrays of the same length, and if α and ω are dyadic functions, then the expression of the form aω/.αb is called an inner product. The result is that α is applied to a and b element by element, followed by ω reduction applied to the result. For example:

`a=.1 2 3`	
`b=.4 5 6`	
`a +/ .*b`	The corresponding elements of *a* and *b* are multiplied and then the results added.
`+/a*b`	Another way to accomplish the same result using reduction.
`a */ .+b`	The corresponding elements of *a* and *b* are added and then the results multiplied.
`*/a+b`	Another way to accomplish the same result using reduction.

2 Program definition

Commands can be accumulated into a script, which by convention is a file ending with suffix .ijs.. Custom functions are objects of type 3:0 for monadic (or optionally ambivalent) verbs, or type 4:0 for dyads. Functions close with).

Consider the following program for computing simple interest at 5 percent per year.

```
.NB compute simple interest
interest =: 3 : 0
'p t' =. y    NB. collect principal and term
r=.0.05       NB. the interest rate is 5 percent
i=.p*r*t
)
```

In a J session, load the program:

`load'interest.ijs'`	Load the program into J
`interest 100 2`	Run the program
`10`	

Instead of entering the values of `p` and `t` as a one argument vector, they can be given separately as `x` and `y` arguments:

```
.NB compute simple interest
interest =: 4 : 0
p=.x          NB. principal
t=.y          NB. term
r=.0.05       NB. interest rate is 5 percent
i=.p*r*t
)
```

Load and run this version:

```
   load'interest.ijs'
   100 interest 2
10
```

The program interest uses the values of p and t to produce the value i. As a function, this program can now be used in other programs as a *subprogram.* To illustrate this point, we shall use the program interest as a subprogram in the following program amount which computes the amount of the loan at 5 percent interest.

```
amount=: 4 : 0
p=.x
t=.y
a=.p+p interest t    NB. the amount owed is principal plus interest
)

   100 amount 2
110
```

As another example, consider the following program which computes the amount s accumulated when a principal p is deposited in a savings bank at a yearly interest rate of 5 percent for n years. Such a bank uses compound interest. So the name chosen for the program is compound.

```
compound=: 4 : 0
r=.0.05
p=.x
n=.y
s=.p*(1+r)^n
)

   100 compound 4
121.551
```

If $100 is deposited in a savings bank at 5 percent interest compounded yearly, then in 4 years, it yields $121.55.

3 Branching

Branching is an instruction to change the regular sequence of steps in a program. In J, unconditional branches are indicated by a label and a goto to jump to the next sentence following the label.

Consider the following program function which prints out the ordered pairs (x, y) for the function $y = x^2$ starting at some specified value of x.

```
function=: 3 : 0
label_1.
x=.y^2
echo y,x
y=.y+1
goto_1.
)
```

In order to run this program, the student chooses an initial value of x, say `_5`, and enters the following:

```
     function _5
   _5 25
   _4 16
   _3 9
   _2 4
   _1 1
   0 0
   1 1
   2 4
   3 9
   4 16
   5 25
```

The trouble is that there is no built-in way to stop this program. It will go on incrementing x and printing out pairs indefinitely unless the programmer does something to stop it. In order to stop an endless program such as this one, press `Ctl-C` (press the `Control` and `C` keys at the same time). The computer will then stop the program and is ready for new work.

```
   |attention interrupt: function
   |       function _5
```

Let us consider a better way to write this program with a built in stopping function. J uses the if control word for conditional branching.

```
function=: 3 : 0
i=.0
label_counter.
i=.i+1
x=.y^2
echo y,x
y=.y+1
```

```
if. i<11 do. goto_counter. end.
)
```

In the previous program, as soo as i is not less than 11, the computer proceeds to line 8, where the program is terminated.

```
     function _5
   _5 25
   _4 16
   _3 9
   _2 4
   _1 1
   0 0
   1 1
   2 4
   3 9
   4 16
   5 25
   6
```

3.1 Line labels

In the above program function, a line label counter was used. A line label is a name given to a particular statement in a program. It is always followed by a period (sometimes called "dot"). In programs involving branching, it is a good idea to use line labels for reference–especially if one expects to alter the program. This is because very few programmers can write every program in final form on their first try. Thus, if originally they wanted to branch to the statement on line 5, in a revised form of the program the original line 5 might now be changed to line 7. This would require the programmer to rewrite the branching statement. However, if the statement on line 5 had a line label, there would be no need to rewrite the branching statement. In addition, line labels can make the finished program easier to interpret by users of the program. Line labels are like local variables. That is, they do not retain their values outside of the program. However, they do not need to be included in the header of the program.

3.2 Another example

We now consider a program for computing the absolute value of a number. The absolute value of a number is the positive value of the number. Thus, if x is positive, then the absolute value of x is x. However, if x is negative, then the absolute value of x is $-x$. Note that the negative of a variable x is denoted by $-x$ rather than _5 which is used to represent a negative constant.

The absolute value of 0 is 0.

```
absolute=: 3 : 0
if. y<0 do. goto_negative. end.
label_positive.
av=.y
return.
label_negative.
av=.-y
av
)
```

If the statement return is used in a program, this is equivalent to telling the computer to exit from the program.

We conclude this brief discussion of branching with one final example. This program, sum, computes the sum of the first k positive integers.

```
sum=: 3 : 0
k=.y
s=.0
n=.1
label_addon.
s=.s+n
n=.n+1
if. n<:k do. goto_addon. end.
s
)
```

Notice that in this program, the incrementing is done after the computation rather than before the computation as was done in the program function. The reader should study the differences between these two programs.

It is worth mentioning that in J there is usually less need for branching than in many other programming languages. This is because of the large number of functions available in J and the array handling capabilities in J. A much easier form of sum using a couple of these functions as follows:

```
sum=: 3 : 0
+/i.>:y.
)
```

`i.>:y` yields the vector of positive integers up to and including y. `+/i.>:y` adds the numbers in `i.>:y`.

```
  sum 100
5050
```

It is also worth mentioning that there really is no need for a program to add the first k positive integers. It can be accomplished directly as follows:

```
   +/i.>:100
5050
```