Introduction to J

© **Harvey Davies**

CSIRO Division of Atmospheric Research

Private Bag No. 1,
Mordialloc 3195,
Australia

email: hld@dar.csiro.au

December 15, 1994

Contents

1 Introduction

J is an exciting new programming language designed by Ken Iverson (well-known as the creator of *APL*) and Roger Hui. Iverson describes J as a dialect of APL, but the differences from APL are more than sufficient to justify the new name.

Why the name *J*? I once assumed a connection with the imaginary number $\sqrt{-1}$, which is conventionally denoted in mathematics by j (or i). In fact this is represented by `0j1` within the J language. However Roger Hui provides the simple explanation "*It is easy to type*" in Hui (1992).

The following are some important features of J:

- It uses the standard ASCII character set rather than the special APL character set.
- It is array-oriented. Arrays are treated in a very systematic, general manner. The elements of arrays can themselves be arrays.
- It has a rich set of *verbs* (functions) to manipulate *nouns* (arrays).
- *Verbs* can be modified, combined and defined using powerful facilities such as *adverbs* and *conjunctions*. For example, the conjunction `D.` gives the derivative of a verb.
- Expressions are used to define verbs, adverbs and conjunctions, as well as nouns. All four types of object are assigned names in the same way.
- Verbs can be defined in traditional *explicit* manner using *pronouns* (variables representing nouns) to represent arguments.
- Verbs can also be defined using a succinct *tacit* (functional) programming style which avoids the need to mention the arguments of the verb.

Although J can be used as a very sophisticated programming language, it can also be used in a very simple way as a desk calculator. Thus if you type '`2+3`', J displays the result '`5`'.

Iverson's company *Iverson Software Inc.* (ISI) has implemented J for MS Windows, MS DOS, Macintosh and Unix. The most recent release is called *J Release 2* and this comes with two manuals Iverson (1994) and Burke (1994). The *J Release 2 for Windows* version includes powerful tools (like those in *Visual Basic*) for developing Windows programs and is available in personal and commercial editions.

ISI products are distributed worldwide by:
Strand Software Systems,
2 Adelaide Street West, Suite 203,
Toronto, Ontario, Canada M5H 1L6
Tel.: (416) 369-1630
Fax.: (416) 369-9590
Email: `anne.faust@rose.com`

The Australian agents are:
Adaptable Systems,
49 First Street,
Black Rock, Victoria 3193, Australia
Tel.: (+61 3) 589 5578
Fax.: (+61 3) 589 3220

The Internet newsgroup `comp.lang.apl` includes a great deal of discussion about J. An *FAQ* document providing answers to *Frequently Asked Questions* is available by anonymous ftp to `watserv1.waterloo.edu`. The file is called `faq` and will be found in the directory `languages/apl/j`, which also contains other documentation and source code for the old version 7, which (unlike J Release 2) is available as shareware.

2 Overview

This section is intended to provide a brief overview of the major features, many of which are covered in more detail later.

2.1 Getting Started

The following annotated log demonstrates some of the basic facilities. Note that command lines (which you type in) are indented while result lines are not. Anything following "`NB.`" is treated as comment.

You may like to run J while reading the following. This will enable you to clarify points by experimenting. It is the best way of learning J.

I have followed Iverson in using grammatical terms in place of mathematical terms. Thus *noun* is used in place of *operand* and *verb* is used in place of *function* or *operator*.

```
   3 + 2                NB. Verb '+' adds nouns '3' and '2'
5
   2-5                  NB. Negative sign is '_' not '-'
_3
   2 ^ 4                NB. Power
16
   2e1 * _1.3           NB. '2e1' means '2*10^1' i.e. 20
_26
   9 % 6                NB. Division symbol is '%' not '/'
1.5
   radius=. 7           NB. 'radius' is pronoun.  Assign is '=.' not '='.
   radius               NB. Display value (assign does not display)
7
   2p1 * radius         NB. '2p1' means '2*pi^1' (c.f. '2e1' above)
43.9823
   7 3 _2 * _1 4 _9     NB. Arguments can be vectors
_7 12 18
   radius =. 7 3 9.5    NB. Vector containing three radii
   2p1 * radius         NB. Three circumferences (scalar times vector)
43.9823 18.8496 59.6903
   1p1 * radius^2       NB. Three areas
153.938 28.2743 283.529
```

2.2 Monadic and Dyadic Verbs

All the above examples have nouns (arguments) on both the left and right of the verb. The following example has only a right argument.

```
   -radius
_7 _3 _9.5
```

Such a *unary minus* is familiar in ordinary mathematical notation and computer languages. Iverson has extended this concept to allow most verbs to take either one or two arguments. Such verbs are said to be *ambivalent*. The two meanings are often related, but this is not always the case. A verb with a single argument (e.g. '`-radius`' above) is said to have valence 1 and is called a *monadic verb* or *monad*. A verb with two arguments (e.g. '`2-5`' above) is said to have valence 2 and is called a *dyadic verb* or *dyad*.

The primitives (in-built verbs, etc.) are denoted by one of the following:

- a single graphic (e.g. '`%`')
- a single graphic followed by '`.`' or '`:`' (e.g. '`=.`')
- a name followed by '`.`' (e.g. '`NB.`')

The following demonstrates some common monads.

```
   % 2 3 5 2r3          NB. Monadic '%' means 'reciprocal'.  '2r3' means '2/3'
0.5 0.333333 0.2 1.5
   ^ 1 2 3              NB. Monadic '^' means 'exp'
2.71828 7.38906 20.0855
   i. 9                 NB. Monadic 'i.' generates integers
0 1 2 3 4 5 6 7 8
   y =. _8 + 2 * i. 9   NB. and thus any arithmetic progression
   y
_8 _6 _4 _2 0 2 4 6 8
   * y                  NB. Monadic '*' is sign (signum) function
_1 _1 _1 _1 0 1 1 1 1
   | y                  NB. Monadic '|' is magnitude (abs)
8 6 4 2 0 2 4 6 8
   6 | y                NB. Dyadic '|' is mod (remainder or residue)
4 0 2 4 0 2 4 0 2
```

New verbs (called *pro-verbs*) are defined by assignment, very much as pronouns are defined. This can be used to assign familiar aliases, such as `mod` in the following example.

```
   mod=. |
   3 mod y
1 0 2 1 0 2 1 0 2
   %: 4 16 9            NB. Monadic '%:' means square root
2 4 3
   %: _9                NB. '0j3' means '0 + 3*j' where j is square root of _1
0j3
   3 %: 125 8 27        NB. Dyadic 'x %: y' means x'th root of y
5 2 3
   !4 1.5 999           NB. Factorial. Monad always has argument on its right
24 1.32934 _
```

`!y` is generalised to non-integers by defining it as $\Gamma(y+1)$. The third element '`_`' means '∞', indicating overflow in this case.

2.3 Combining Verbs

We next consider how verbs are combined to form more complicated expressions. The following expression represents $-e^{\frac{1}{2}}$ using three monads:

```
   -^%2
_1.64872
```

Note that the order of execution has to be *right-to-left*. It is treated as

```
-(^(% 2))
```

Most computer languages have complex rules for combining verbs. For example, Fortran 77 has nine levels of operator precedence; most operators are left-associative (executed left-to-right), one (`**`) is right-associative and some (e.g. `.EQ.`) are non-associative.

Iverson's notation has no such precedence rules. All verbs have equal rights! There seems to be no good reason to give certain verbs precedence over others. Such rules would be horrendous in J which has over 70 primitive verbs and the capacity to define any number of others. The only precedence rule in J is that adverbs and conjunctions have higher priority than verbs.

The order is simply right-to-left for both dyads and monads. This may seem strange at first, but it really is much simpler than the complex traditional rules. There is much less need for parentheses.

```
   9-5+2                NB. Treated as 9-(5+2)
2
   3*4-2                NB. Simply right-to-left again
6
   (3*4)-2              NB. Parentheses can be used
10
   _2+3*4               NB. But can often be avoided
10
   3+%4                 NB. Combination of monad & dyad
3.25
   y                    NB. Using y defined above
_8 _6 _4 _2 0 2 4 6 8
   -3**y-2              NB. -( 3 * ( sign(y-2) ) )
3 3 3 3 3 0 _3 _3 _3
```

2.4 Trigonometric Functions

The dyadic verb '`o.`' provides various trigonometric and hyperbolic functions. Note the mnemonic association between the circular letter '`o`' and functions related to the circle.

```
   1 o. 0.5p1           NB. '1 o. x' means 'sin x' so this is 'sin pi%2'
1
   2 o. 60 % 180p_1     NB. '2 o. x' means 'cos x' so this is 'cos 60 deg'
0.5
```

It is convenient to assign conventional names (as pro-verbs) to these functions. This can be done using the *bond* conjunction '`&`' to define a monad in terms of a dyad with one argument bonded (i.e. fixed or constant).

```
   sin=. 1 & o.            NB. sin is o. with left argument bound to 1
   sin 360 30 90 % 180p_1  NB. sin of 360 30 90 degrees
0 0.5 1
```

2.5 Comparisons and Logical (Boolean) Values

The following comparison dyads return the result 1 for *true* and 0 for *false*. Note that here '`:`' means *equal*.

```
   3 6 4 < 2 9 4        NB. less than
0 1 0
   3 6 4 <: 2 9 4       NB. less than or equal to
0 1 1
   3 6 4 = 2 9 4        NB. equal to
0 0 1
   3 6 4 ~: 2 9 4       NB. not equal to
1 1 0
   3 6 4 > 2 9 4        NB. greater than
1 0 0
   3 6 4 >: 2 9 4       NB. greater than or equal to
1 0 1
```

As shown above, dyadic '`=`' does *element-by-element* comparison. There is also a dyadic verb *match* (`-:`) which tests whether the two entire arguments are *equal in all respects* or *congruent*, which is conventionally represented by the symbol '$\equiv$'. So it is mnemonic to again consider the '`:`' to mean '$=$', giving three horizontal lines in all.

```
   3 6 4 -: 2 9 4
0
   3 6 4 -: 3 6 4
1
```

The dyadic verbs '`<.`' and '`>.`' give the *minimum* and *maximum*

```
   3 6 4 <. 2 9 4       NB. lesser of (minimum)
2 6 4
   3 6 4 >. 2 9 4       NB. greater of (maximum)
3 9 4
   1 >. 8 <. 3 0 9 7    NB. restrict range to 1 to 8 (Again right-to-left!)
3 1 8 7
```

An advantage of treating *true* and *false* as numbers is shown by the following.

```
   3 0 2 # 2 8 5        NB. 'x#y' means x copies of y
2 2 2 5 5
   v =. 9 4 3 5 0
   v >: 4
1 1 0 1 0
   (v >: 4) # v         NB. Select elements greater than or equal to 4
9 4 5
```

The logical operations *and*, *or* and *not* can be done as follows.

```
0 0 1 1 *. 0 1 0 1      NB. and
0 0 0 1
0 0 1 1 +. 0 1 0 1      NB. or
0 1 1 1
-. 0 1                  NB. not
1 0
```

2.6 Character Nouns

So far all nouns have been numeric. J automatically determines whether the type is *Boolean* (i.e. 0 or 1), *integer*, *real* or *complex*. J also allows *character* nouns, which are arrays whose elements are ASCII characters.

```
   cv=. 'Sam can''t!'   NB. Define character vector 'cv'
   cv
Sam can't!
   cv = 'a'             NB. Many verbs apply also to character nouns
0 1 0 0 0 1 0 0 0 0
   cv > 'a'             NB. But some do not
|domain error
|   cv  >'a'
   cv -: 6              NB. Some even allow a mixture of both types
0
```

2.7 Adverb 'Insert' (/)

We have seen how J extends many verbs to arrays in an element by element fashion. We now consider some other facilities such as adverbs. An adverb has a single argument on its left. In the following, a verb to sum the elements of a vector (or the columns of a matrix, etc.) is defined by '`+/`'. The adverb '`/`' is called *insert* because it has the effect of inserting its verb argument between the items of the noun argument of the resulting verb.

```
   data=. 4 _1 5 2
   +/ data              NB. Monadic '+/' means 'sigma' i.e. sum
10
   sum=. +/             NB. So let us define it as the pro-verb 'sum'
   sum data
10
   */ data              NB. Similarly, monadic '*/' means 'product'
_40
   >./ data             NB. and monadic '>./' means 'maximum'
5
   -/ data              NB. and monadic '-/' means 'alternating sum'
8
   4 - _1 - 5 - 2       NB. In other words this (Remember right-to-left!)
8
   # data               NB. Monadic '#' means 'tally' or 'number of elements'
4
   (sum data) % # data  NB. So arithmetic mean is sum divided by tally
2.5
```

2.8 Tacit Definition, Forks and Hooks

You may be surprised to learn that the above calculation of an arithmetic mean can be written more succinctly as

```
   (sum % #) data
2.5
   sum % # data         NB. which is quite different to this
0.25
   sum ( % ( # data ) ) NB. which means this
0.25
   % ( # data )         NB. or this (sum of a scalar is simply itself)
0.25
```

The original expression '`(sum % #)`' is evaluated first (because it is parenthesised) to define a verb meaning *arithmetic mean*. How can a sequence (or *train*) of verbs define another verb? Here we have a train of three verbs called a *fork* and the resultant verb is used monadically. In this case
$(verb_1\ verb_2\ verb_3)\ noun$
is defined to mean
$(verb_1\ noun)\ verb_2\ verb_3\ noun.$

Forks and other trains provide a powerful way of defining verbs. The following definition of `am` is a classic example of *tacit definition*.

```
   am=. sum % #         NB. Define pro-verb 'am' as arithmetic mean
   am data
2.5
   max=. >./            NB. Similarly 'range' can be defined as 'max-min'
   min=. <./
   range=. max - min
   range data
6
```

The following illustrates the dyadic meaning of a fork:

```
   5(+*-)3              NB. This dyadic fork means 'product of sum and difference'
16
   (5+3)*(5-3)          NB. i.e. this
16
```

A train of two verbs is called a *hook*. The monadic and dyadic meanings of hooks are illustrated by the following:

```
   (+%)4                NB. This monadic hook means 'add reciprocal'
4.25
   4+%4                 NB. i.e. this
4.25
   7(+%)4               NB. Dyadic use of hook
7.25
   7+%4                 NB. means simply this
7.25
```

2.9 Explicit Definition

Note that such *tacit* verb definitions do not involve noun arguments. We could have used the conjunction ':' to define `am` using a more traditional *explicit definition* such as:

```
   am=. '(sum y.) % # y.' : ''
   am data
2.5
```

Note that in any explicit definition, `y.` represents the right argument of the resultant verb. The left (character) argument of ':' defines the monad and the right argument defines the dyad. In the above example the right argument is `''` which means *undefined*. We could define dyadic '`am`' to give a weighted mean as follows:

```
   am=. '(sum y.) % # y.' : '(sum x. * y.) % sum x.' NB. 'x.' is left argument
   1 2 0 3 am data      NB. Weighted mean
1.33333
   am 1 2 0 3 # data    NB. Equivalent to this (using monadic 'am')
1.33333
```

3 Array Manipulation

3.1 Extracting Elements from Vectors

Elements can be extracted from arrays using the verb '`{`' called *from*. The left argument is the subscript, with 0 corresponding to the first item.

```
   data                 NB. Display vector 'data' again
4 _1 5 2
   0 { data             NB. Extract first element from 'data'
4
   2 1 { data           NB. Extract third (2) and second (1) elements
5 _1
   _1 _1 _3 { data      NB. -ve subscripts are relative to end (_1 is last)
2 2 _1
```

The verb *from* (`{`) can be used to write conditional expressions like the C ternary operator '`?:`'

```
   (3 < i. 6) { 2 9      NB. If element > 3 then result is 9, otherwise 2
2 2 2 2 9 9
   2 + 7 * 3 < i. 6      NB. Another way of expressing this step function
2 2 2 2 9 9
   data i. _1 2          NB. Dyadic 'i.'  Find index of values _1 & 2 in 'data'
1 3
   {. data               NB. Monadic '{.' is 'head' i.e. 1st element
4
   {: data               NB. Monadic '{:' is 'tail' i.e. final element
2
   }. data               NB. Monadic '}.' is 'behead' i.e. all except head
_1 5 2
   }: data               NB. Monadic '}:' is 'curtail' i.e. all except tail
4 _1 5
   2 {. data             NB. Dyadic '{.' is 'take' i.e. take first 2 elements
4 _1
   _3 {. data            NB. If left argument -ve then take from end
_1 5 2
   2 }. data             NB. Dyadic '}.' is 'drop' i.e. drop first 2 elements
5 2
   |. data               NB. Monadic '|.' means 'reverse' (Mnemonic: '|' is axis)
2 5 _1 4
   1 |. data             NB. Dyadic '|.' means 'rotate'
_1 5 2 4
```

3.2 Changing Elements of Vector

Specific array elements can be changed by using the adverb '}' (*amend*) to define a dyadic verb to change these elements. Say we want to change element 1 to 0 and element 2 to 9.

```
   v=. 1 2 }             NB. Define verb 'v' to change elements 1 & 2
   0 9 v data            NB. Use it
4 0 9 2
   0 9 (1 2 }) data      NB. All in single expression
4 0 9 2
```

Note that this amendment process has three arguments. One is the argument of the adverb and two are arguments of the resultant verb.

3.3 Matrices

A *matrix* (or *table*) is a 2-dimensional rectangular array. The following illustrates the construction and manipulation of matrices.

```
   mat=. 2 3$4 _1 5 2 _7 0  NB. Dyadic '$' forms array with specified shape & elements
   mat                   NB. Display matrix 'mat'
4 _1 5
2 _7 0
   $mat                  NB. Monadic '$' gives shape (Mnemonic: $ like S = Shape)
2 3
   $$mat                 NB. Shape of shape is 'rank' i.e. number of dimensions
2
   +/mat                 NB. Column sums
6 _8 5
```

```
   sum mat              NB. since we defined 'sum=. +/' above
6 _8 5
   sum"1 mat            NB. Row sums.  '"1' means over row vectors (i.e. rank 1)
8 _5
   # mat                NB. Number of rows
2
   (+/ mat) % # mat     NB. Column means
3 _4 2.5
   am mat               NB. Using above definition of 'am'
3 _4 2.5
   am"1 mat             NB. Row means
2.66667 _1.66667
   0 { mat              NB. Row 0
4 _1 5
   2 { 0 { mat          NB. Element 2 of row 0
5
   |: mat               NB. Transpose
 4  2
_1 _7
 5  0
```

3.4 Concatenating Arrays

Vectors and scalars can be concatenated as follows using the dyad *append items* (,).

```
   data                 NB. As defined above
4 _1 5 2
   data , 13            NB. Concatenate vector and scalar
4 _1 5 2 13
   13 _9,data           NB. Concatenate two vectors
13 _9 4 _1 5 2
   9,0                  NB. Concatenate two scalars
9 0
```

The following shows how this verb handles matrices:

```
   mat                  NB. As defined above
4 _1 5
2 _7 0
   mat2x3=. i. 2 3      NB. Vector argument of i. specifies shape
   mat2x3
0 1 2
3 4 5
   mat , mat2x3         NB. Concatenate columns of two matrices
4 _1 5
2 _7 0
0  1 2
3  4 5
   mat , 3              NB. Concatenate matrix & scalar to add row
4 _1 5
2 _7 0
3  3 3
   mat , 7 8 9          NB. Concatenate matrix & vector to add row
4 _1 5
```

```
2 _7 0
7  8 9
   mat2x2=. i. 2 2      NB. Define another matrix
   mat2x2
0 1
2 3
   mat2x2 ,"1 mat       NB. Concatenate rows of two matrices
0 1 4 _1 5
2 3 2 _7 0
```

Note how the conjunction *rank* (`"`) was again used to specify the rank of the arguments. In this case `,"1` means concatenation of vectors i.e. rows. Dyadic *append* has the same effect, as shown in the first of the following examples:

```
   mat2x2 ,. mat        NB. Append matrix to matrix
0 1 4 _1 5
2 3 2 _7 0
   mat ,. 3             NB. Append scalar to matrix
4 _1 5 3
2 _7 0 3
   1 2 ,. 3 5           NB. Append vector to vector
1 3
2 5
```

Dyadic `,:` is called *laminate*. It concatenates the arguments over a new first dimension after reshaping them to have the same shape.

```
   mat ,: 3             NB. Laminate matrix with scalar
4 _1 5
2 _7 0

3  3 3
3  3 3
   $ mat ,: 3           NB. Shape
2 2 3
   mat ,: 6 7 8 9       NB. Laminate matrix with vector
4 _1 5 0
2 _7 0 0

6  7 8 9
0  0 0 0
   $ mat ,: 6 7 8 9     NB. Shape
2 2 4
   mat ,: mat2x2        NB. Laminate matrix with matrix
4 _1 5
2 _7 0

0  1 0
2  3 0
   $ mat ,: mat2x2      NB. Shape
2 2 3
```

3.5 Monads which Reshape Arrays

Monadic `,` is called *ravel* and produces a vector containing the elements of the argument.

```
   , mat                NB. Ravel of matrix
4 _1 5 2 _7 0
   $ , mat              NB. Shape
6
   , 3                  NB. Ravel of scalar
3
   $ , 3                NB. Shape
1
```

Monadic `,.` is called *ravel items* and produces a matrix.

```
   ,. 3                 NB. Ravel-items of scalar
3
   $ ,. 3               NB. Shape
1 1
   ,. 2 3               NB. Ravel-items of vector gives 1-column matrix
2
3
   $ ,. 2 3             NB. Shape
2 1
```

Monadic `,:` is called *itemize* and adds a dimension of 1.

```
   ,: 5 7               NB. Itemize of vector gives 1-row matrix
5 7
   $ ,: 5 7
1 2
   ,: mat               NB. Itemize of matrix gives 3D array
4 _1 5
2 _7 0
   $ ,: mat
1 2 3
```

3.6 Function Table

We have seen above how the adverb *insert* (`/`) defines monadic verbs such as sum above. Such verbs also have a dyadic meaning called *function table* or *outer product*.

```
   2 3 4 */ 0 1 2 3 4   NB. Dyadic product (multiplication table)
0 2 4  6  8
0 3 6  9 12
0 4 8 12 16
```

Let v be the verb argument of '/'.
Let X be the vector left argument of 'v/'.
Let Y be the vector right argument of 'v/'.
Let Z be the matrix result.
Z_{ij} is defined as $X_i \; v \; Y_j$.

In the above example v is '`*`', X is '2 3 4' and Y is '0 1 2 3 4'. So Z is defined by:
$Z_{ij} = X_i * Y_j$

This is a convenient way of defining many matrices. For example:

```
   i=. i. 5
   i
0 1 2 3 4
   i =/ i               NB. Identity matrix
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

Dyadic '$x!y$' gives the number of combinations of y things, x at a time.

```
   |: i !/ i            NB. Pascal's triangle (|: is 'transpose')
1 0 0 0 0
1 1 0 0 0
1 2 1 0 0
1 3 3 1 0
1 4 6 4 1
```

3.7 Linear Algebra

Simultaneous linear equations can be solved using '%.', which is *matrix divide* as explained below. For example let us solve the system:

$$\begin{array}{rcrcr} 3x & - & 6y & = & 10 \\ 9x & + & 15y & = & -14 \end{array}$$

```
   A=. 2 2 $ 3 _6 9 15  NB. Define 2 by 2 matrix
   A
3 _6
9 15
   10 _14 %. A          NB. Matrix-divide right-hand-side by this matrix
0.666667 _1.33333
```

Thus the solution is $x = \frac{2}{3}$ and $y = -\frac{4}{3}$. This can be checked using matrix multiplication. Matrix multiplication is not a primitive, but it can easily be defined using the conjunction '.' meaning *dot product*.

```
   matmul =. sum . *    NB. Matrix product is dyadic sum of products
   A matmul 2r3 _4r3    NB. Check result
10 _14
   %. A                 NB. Matrix inverse
  0.151515 0.0606061
_0.0909091  0.030303
   (%. A) matmul 10 _14 NB. Matrix divide 'x %. y' is defined as (%. y) matmul x
0.666667 _1.33333
   det =. -/ . *        NB. Determinant is monadic alternating sum of products
   det A
99
   (3 * 15) - (_6 * 9)  NB. In other words, this.
99
```

3.8 Boxed Arrays

The *array* is the only data structure in J. A *scalar* is treated as an array with rank 0. All the elements of an array must be of the same data type. This is the reason for the failure of the following attempt to mix characters with numbers:

```
   3 , 'A'
|domain error
|   3  ,'A'
```

Besides data types *numeric* and *character*, there is another called *boxed array* or simply *boxed*. Boxed arrays can be elements of higher level arrays. One can think of putting arrays into boxes and then forming an array of these boxes.

Internally, the C code represents a box as a pointer to (i.e. address of) the array inside. The ability to nest arrays to any depth provides an elegant means of representing data structures such as *trees*.

Any array can be boxed using the monadic verb *box* (`<`). A useful mnemonic is to imagine a '`>`' on the right (e.g. `<3>`) to complete the box. The following example illustrates the use of `<`

```
   b =. < 2 _6                 NB. b is box containing vector
   b                           NB. Display b.  Note the surrounding box
+----+
|2 _6|
+----+
   $ b                         NB. Shape of b

   # $ b                       NB. Rank of b
0
```

Thus `b` is a scalar (rank 0 array) even though it contains a vector. Since boxed arrays are scalars they can be used as the elements of a higher level array.

```
   a =. b , < 3                NB. Append boxed scalar to boxed vector
   a
+----+-+
|2 _6|3|
+----+-+
   $ a                         NB. Shape of a
2
   0 { a                       NB. 1st (0) element of a
+----+
|2 _6|
+----+
   1 } a                       NB. 2nd (1) element of a
+-+
|3|
+-+
```

The effect of *box* (`<`) is reversed by the monadic verb *open* (`>`).

```
   > b                         NB. Open b
2 _6
   > > b                       NB. Open has no effect if already open
2 _6
   > 0 { a                     NB. Open 1st element
2 _6
   $ > 0 { a                   NB. Check shape
2
   > 1 } a                     NB. Open 2nd element
3
   $ > 1 } a                   NB. Shape is empty i.e. scalar

   # $ > 1 } a                 NB. i.e. rank = 0
0
```

The following shows how

- dyad *link* (`;`) combines the action of '`<`' and '`,`'
- several arrays can be combined into a single (array) entity

```
   (< 2 _6) , < 3              NB. Instead of this
+----+-+
|2 _6|3|
+----+-+
```

```
   2 _6 ; 3                    NB. we can use ';' (link)
+----+-+
|2 _6|3|
+----+-+
   'hello' ; 3 9               NB. Can link character & numeric arrays
+-----+---+
|hello|3 9|
+-----+---+
   c =. 8 5 ; 'x' ; a ; 9      NB. And even boxed arrays
   c
+---+-+--------+-+
|8 5|x|+----+-+|9|
|   | ||2 _6|3|| |
|   | |+----+-+| |
+---+-+--------+-+
   1 { > 0 { > 2 { c           NB. Extract element from inner box
_6
```

The following phone list example begins by showing how boxed arrays can be used to store text containing lines of different lengths.

```
   names =. 'Ed' ; 'Mary' ; 'Tom'
   names
+--+----+---+
|Ed|Mary|Tom|
+--+----+---+
   $ names
3
   matrix =. > names           NB. Open whole array, not just single element
   matrix
Ed
Mary
Tom
   $ matrix                    NB. Shape of matrix
3 4
   matrix = ' '                NB. Show position of spaces
0 0 1 1
0 0 0 0
0 0 0 1
   list =. 2 3 $ names , 123 ; 432 ; 987
   list
+---+----+---+
|Ed |Mary|Tom|
+---+----+---+
|123|432 |987|
+---+----+---+
   |: list                     NB. Transpose
+----+---+
|Ed  |123|
+----+---+
|Mary|432|
+----+---+
|Tom |987|
+----+---+
```

The following examples show how the conjunction *under* (`&.`) can be used to apply some verb to each of the enclosed arrays.

```
   d =. 3 0 4 ; a
   d
+-----+----+-+
|3 0 4|2 _6|3|
+-----+----+-+
   +/ &. > d                   NB. Sum of each boxed array element
+-+--+-+
|7|_4|3|
+-+--+-+
   |. &. > d                   NB. Reverse each boxed array element
+-----+----+-+
|4 0 3|_6 2|3|
+-----+----+-+
```

Verbs are displayed in boxes. In fact verbs are represented internally as boxed arrays. The following example shows how one can display a verb. Note how the boxes show the order of execution.

```
   range =. >./ - <./          NB. Define verb using fork
   range                       NB. Display verb
+------+-+------+
|+--+-+|-|+--+-+|
||>.|/|| ||<.|/||
|+--+-+| |+--+-+|
+------+-+------+
```

3.8.1 Boxed Subscripts

One important use of boxed arrays is as subscripts.

```
   table=. 'cat','dog',:'emu' NB. Create 3x3 matrix
   $table
3 3
   table
cat
dog
emu
   ]s=. <2 1                  NB. Define scalar single-level boxed subscript
+---+
|2 1|
+---+
   s{table                    NB. which specifies individual elements
m
   1{2{table                  NB. i.e. same as this
m
   ]s=. 2 1;0 0;1 0           NB. Vector single-level boxed subscript
+---+---+---+
|2 1|0 0|1 0|
+---+---+---+
   s{table                    NB. gives several individual specifications
mcd
   ]s=. <2 1;0 2              NB. 2-level boxed subscript
+---------+
|+---+---+|
||2 1|0 2||
|+---+---+|
+---------+
   s{table                    NB. means rows 2 & 1, columns 0 & 2
eu
dg
   ]s=. <(<2 1),<<1           NB. 3rd level means 'all except'
+---------+
|+---+---+|
||2 1|+-+||
||   ||1|||
||   |+-+||
|+---+---+|
+---------+
   s{table                    NB. i.e. same (rows 2 & 1, all columns except 1)
eu
dg
```

4 J Programming

So far we have been mainly concerned with using J as a powerful calculator using the primitive (inbuilt) verbs. We now consider how to

- define new verbs and thus use J as a programming language.
- store J code (including definitions of verbs) in script files

4.1 Defining Verbs with Explicit Definition (:) Conjunction

We first consider the case where both the left and right arguments are character vectors. The following example shows how the left argument of ':' defines the monadic meaning and the right argument defines the dyadic meaning. The pronouns `x.` and `y.` respectively represent the left and right arguments of the defined verb.

```
   max=. '>. / y.' : 'x. >. y.'        NB. Define ambivalent verb 'max'
   max                                 NB. Display definition
+--------+-+--------+
|>. / y.|:|x. >. y.|
+--------+-+--------+
   max 1 8 _3 0                        NB. Use monad
8
   _2 0 1 max 1 _9 3                   NB. Use dyad
1 0 3
```

The arguments of ':' can be any character expressions. So the above definition of 'max' is equivalent to:

```
   max=. '>.'
   max=. (max , '/ y.') : ('x.' , max , 'y.')
   max 1 8 _3 0
8
   _2 0 1 max 1 _9 3
1 0 3
```

Note that `max` was first defined as a noun, then redefined as a verb. It was necessary to parenthesise the character expressions because conjunctions (like '`:`') and adverbs have higher precedence than verbs.

An expression defining a verb can be used directly without assigning it to a pro-verb (and thus giving it a name). The above example could be done directly as:

```
   m=. '>. / y.'                        NB. Monad
   d=. 'x. >. y.'                       NB. Dyad
   m : d 1 8 _3 0
8
   _2 0 1 m : d 1 _9 3
1 0 3
```

The following shows that the space preceding '`:`' is needed to prevent '`m:`' being treated as a single primitive object. (Two-character primitives have either '`:`' or '`.`' as the second character.)

```
   m:d 1 8 _3 0
|value error
|     m: d 1 8 _3 0
```

The left and right arguments of '`:`' can be verbs, as in the following:

```
   max=. >. / : >.
   max 1 8 _3 0
8
   _2 0 1 max 1 _9 3
1 0 3
   m : >. 1 8 _3 0                      NB. Can also have 'string : verb'
8
   >. / : d 1 8 _3 0                    NB. or 'verb : string'
8
```

There was no need to parenthesise the left argument of '`:`' because adverbs (like '`/`') and conjunctions are executed left to right.

4.2 Examples: Verbs for Statistics

I must acknowledge Smillie (1993) who provided some of the following examples. His document is available by `ftp` to `ftp.cs.ualberta.ca` in file `pub/smillie/intj.ps`.

4.2.1 Arithmetic Mean

The following shows various ways of defining our old friend the arithmetic mean.

```
   data=. 1.5 8.1 _1.2 4
   am=. '(+ / y.) % # y.' : ''
   am data
3.1
   '(+/ y.) % # y.' : '' data
3.1
   (+/ % #) data                        NB. Tacit definition
3.1
```

The monad *same* (`]`) simply gives the value of its argument. It can be used as follows to assign and display in a single sentence. The value of a sentence (expression) is displayed only if the final (left-most) operation is not assignment. For example:

```
   ] mean=. am data
3.1
   mean
3.1
```

Let us now generalise verb `am` to optionally take a left argument providing weights (e.g. frequencies).

```
   f=. 1 2 0 1                         NB. frequencies
   am=. '(+ / y.) % # y.' : '(+/ x. * y.) % +/ x.'
   am data
3.1
   f am data
5.425
   ] all=. f # data                    NB. Check
1.5 8.1 8.1 4
   am all
5.425
```

4.2.2 Frequency Table

What about the reverse process of defining frequencies from raw data? This can be done using *self-classify*, which is monadic '='. But first let us consider the related monad *nub* (`~.`) which provides a list of unique items.

```
   raw=. 1.4 _3 1.4 0 1.4 _3
   ] nub=. ~. raw                      NB. Define nub
1.4 _3 0
   nub =/ raw                          NB. Relate raw to nub
1 0 1 0 1 0
0 1 0 0 0 1
0 0 0 1 0 0
   = raw                               NB. Which is equivalent to monadic =
1 0 1 0 1 0
0 1 0 0 0 1
0 0 0 1 0 0
   ] f=. +/"1 = raw                    NB. Row sums are required frequencies
3 2 1
   ] i=. /: nub                        NB. Ascending sort permutation vector
1 2 0
   nub=. i { nub                       NB. Use to sort nub
   nub
_3 0 1.4
   ] f=. i { f                         NB. & f
2 1 3
   ] cf=. +/\ f                        NB. Cumulative frequencies
2 3 6
   nub ,. f ,. cf
 _3 2 2
  0 1 3
1.4 3 6
```

Several of the verbs and adverbs used here are new and require some explanation. Verb *grade up* (`/:`) gives a permuted (re-ordered) subscript vector corresponding to ascending sorted values. In other words

```
(/: x) { x
```

gives `x` in ascending order.

Adverb *prefix* (`\`) defines successive items of the result by repeatedly applying its left verb argument to the first i items of its noun right argument, with i varying from 1 to the number of items in the noun argument. In the above case the left argument was '`+/`', so the resultant verb gave partial sums. The following is another example of the use of *prefix*:

```
   <\ 2 4 6 8
+-+---+-----+-------+
|2|2 4|2 4 6|2 4 6 8|
+-+---+-----+-------+
```

Let us now define a monadic verb to produce such a frequency table from raw data. We use the conjunction '`:`' with both arguments of 0. This causes the definition to be read from the following lines, terminating with a line containing just '`)`'.

The result of the verb is the result of the final sentence. This result (like all within verbs) is not displayed.

```
freqTable=. 0 : 0
nub=. ~. y.                           NB. nub
f=. +/"1 = y.
i=. /: nub
nub=. i { nub
f=. i { f
cf=. +/\ f
N=. {: cf                             NB. tail of cf is total number cases
nub ,. f ,. cf ,. (f%N) ,. cf%N
)
   freqTable raw
 _3 2 2 0.333333 0.333333
  0 1 3 0.166667      0.5
1.4 3 6      0.5        1
   N
|value error
```

Columns were added for the *relative frequency* and *relative cumulative frequency*. Note that `N` is undefined at the J command level because it is local to verb `freqTable`. A verb can define *global* objects by assigning with '`=:`' rather than '`=.`'.

4.2.3 Simulation of Dice

The expression '`?6`' gives a random number from 0 to 5. Let us define a verb `dice` to simulate y trials, each involving x (default 1) dice. The dyad gives the sum of x dice in each of the y trials.

```
   dice=. '1 dice y.' : '+/ 1 + ? (x. , y.) $ 6'
   dice
+---------+-+----------------------+
|1 dice y.|:|+/ 1 + ? (x. , y.) $ 6|
+---------+-+----------------------+
   freqTable dice 10000               NB. single die
1 1650  1650  0.165  0.165
2 1631  3281 0.1631 0.3281
3 1637  4918 0.1637 0.4918
4 1751  6669 0.1751 0.6669
```

```
5 1670  8339  0.167 0.8339
6 1661 10000 0.1661      1
   freqTable 2 dice 10000              NB. 2 dice
 2  276   276 0.0276 0.0276
 3  547   823 0.0547 0.0823
 4  841  1664 0.0841 0.1664
 5 1085  2749 0.1085 0.2749
 6 1380  4129  0.138 0.4129
 7 1657  5786 0.1657 0.5786
 8 1413  7199 0.1413 0.7199
 9 1116  8315 0.1116 0.8315
10  858  9173 0.0858 0.9173
11  549  9722 0.0549 0.9722
12  278 10000 0.0278      1
```

Let us now make the number of sides on each die a variable `nsides`. This time we use the `0 : 0` facility:

```
   dice=. 0 : 0
1 dice y.
:
+/ 1 + ? (x. , y.) $ nsides
)
   dice
+---------+-+--------------------------+
|1 dice y.|:|+/ 1 + ? (x. , y.) $ nsides|
+---------+-+--------------------------+
   nsides=. 2
   freqTable dice 10000
1 4956  4956 0.4956 0.4956
2 5044 10000 0.5044      1
```

Note how the line containing just '`:`' separates the definition of the monad from that of the dyad. This was not needed in the definition of `freqTable` because there was no dyad. Also note that the pronoun `nsides` was visible from within the verb even though it was defined using *local assign* (`=.`).

4.3 J-Scripts

A file containing J sentences is called a *J-script* and conventionally has a filename with extension `.js`. The effect of executing a script is the same as if you typed in its contents. Under *J Release 2 for Windows* you can execute a script by selecting the menu item *Run/File.*

It is also possible to execute J-scripts using verbs defined using *foreign conjunction* (`!:`), which is discussed further in Section 5.2.5. The following examples use such a verb called `script`, which requires the following different definitions under Version 7 and Release 2.

```
   script=: 0!:2        NB. Version 7
   script=: 0!:1        NB. Release 2
```

Say the above definition of verb `dice` is stored in a file `dice.js`. The following shows how to use `script` to execute this J-script file and thus define `dice`: Note that the filename argument has to be boxed using `<`.

```
   script < 'dice.js'     NB. Execute file 'dice.js'
   dice
+---------+-+--------------------------+
|1 dice y.|:|+/ 1 + ? (x. , y.) $ nsides|
+---------+-+--------------------------+
```

It is usually convenient to define standard verbs and other objects in a J-script (often named `profile.js`) which is automatically executed when J is started. See Burke (1994) (p52) for further details.

4.4 Explicit and Tacit Definitions of Standard Deviation, etc.

The first step is to define a verb `dev` which gives *deviations from the mean.* We will use upper case for *explicitly* defined verbs and lower case for those defined using *tacit definition.*

```
   obs=. 2.3 5 3.5 6                        NB. observations
   AM=. '(+/ y. % # y.)' : ''               NB. explicit definition of AM
   AM obs
4.2
   am=. +/ % #                              NB. tacit definition of am
   am obs
4.2
   DEV=. 'y. - am y.' : ''                  NB. explicit definition of DEV
   DEV obs
_1.9 0.8 _0.7 1.8
   dev=. - am                               NB. tacit definition of dev using hook
   dev obs
_1.9 0.8 _0.7 1.8
```

This tacit definition of `dev` consists of a train of two verbs '`-`' and '`am`'. A train of two verbs '$v_1 v_2$' is called a *hook.* The monadic meaning of any hook '$(v_1 v_2)n$' (where n is the noun argument) is defined as '$n v_1 v_2 n$'. Thus '`- am`' means *subtract mean.*

We now want to define verbs giving the *sum of squares of deviations from the mean.*

One might be tempted to try '`ss=. +/ *: dev`', but this definition is a train of three verbs (i.e. a *fork*), which has a quite different meaning. We can specify that we simply want one verb executed after another by specifying them as arguments of the conjunction *atop* (`@`).

```
   SS=. '+/ *: dev y.' : ''                 NB. *: is square
   SS obs
7.98
   ss=. +/ @ *: @ dev
   ss obs
7.98
```

Next we define verbs giving the *variance* and *standard deviation*:

```
   VAR=. '(ss y.) % # y.' : ''
   VAR obs
1.995
   var=. ss % #                             NB. fork
   var obs
1.995
   SD=. '%: var y.' : ''                    NB. %: is square root
   SD obs
1.41244
   sd=. %: @ var
   sd obs
1.41244
```

Finally, let us modify these (by subtracting 1 from the sample size) to give an *unbiased estimate* of the population variance and its square root.

```
   VAR1=. '(ss y.) % _1 + # y.' : ''
   VAR1 obs
2.66
   var1=. ss % <: @ #                 NB. <: is decrement (subtract 1)
   var1 obs
2.66
   SD1=. %: @ var1
   SD1 obs
1.63095
   sd1=. %: @ var1
   sd1 obs
1.63095
```

This definition of `var` is treated as '`ss % (<: @ #)`' because the conjunction `@` has higher priority than the verbs. The verb `<: @ #` means *one less than the number of items* and, together with verbs `ss` and `%`, forms a fork.

5 Adverbs and Conjunctions

Much of the power of J comes from its ability to define new verbs, adverbs and conjunctions using expressions whose operators are *adverbs* and *conjunctions* and whose arguments are verbs and nouns. Computer scientists would describe verbs, adverbs and conjunctions as *first class objects* because J provides facilities to define them at run time.

An *adverb* has a single argument on its left. A *conjunction* is positioned between its two arguments. Adverbs and conjunctions have higher precedence than verbs and are executed *left-to-right*.

Note how adverbs and conjunctions are identified in the *Vocabulary Table* on the back cover of both Iverson (1993b) and Iverson (1994). Adverbs are in *italics* and conjunctions are in **bold**.

5.1 Adverbs

Adverbs *insert/table* (`/`) and *amend* (`}`) were discussed previously and will not be considered here.

We also discussed how the adverb *prefix/infix* (`\`) was used to define monadic verbs such as *partial sum*. The term *infix* refers to its use to define a dyad which applies some verb to the elements in each position of a moving window. Thus we can define a *moving average* as follows:

```
   data =. 8 2 9 3 0 2
   # data                     NB. number of items
6
   am =. +/ % #               NB. arithmetic mean
   ] ma =. 2 am\ data         NB. moving average with window of length 2
5 5.5 6 1.5 1
   # ma                       NB. number of items
5
   ] ma =. 3 am\ data         NB. moving average with window of length 3
6.33333 4.66667 4 1.66667
   # ma                       NB. number of items
4
```

The adverb *reflex/commute* (`~`) acts on a dyad u to produce

- the *reflex* monad defined by $y\ u\ y$
- the *commuted or crossed* dyad defined by $y\ u\ x$

where x is the left noun argument of the new dyad and y is the right noun argument of the new monad or dyad. For example:

```
   +~ 4                           NB. Reflex puts argument on left & right
8
   4 + 4                          NB. In other words, this
8
   */~i.4                         NB. Use reflex to produce multiplication table
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
   4 -~ 3                         NB. Commute swaps arguments
_1
   'abcd' {~ 2 , 0                NB. Commute puts subscript on right
ca
```

5.2 Conjunctions

5.2.1 Fit or Customize (!.)

Conjunction *fit (customize)* (`!.`) modifies various verbs in various ways. Often it essentially provides a third argument to a dyad. The following example shows how it changes the relative tolerance allowed when deciding whether two values should be considered equal.

```
   9 = 9 + 1e_13                  NB. Default (relative) tolerance is 2^_44 (5.7e_14)
1
   9 * 2^_44                      NB. Thus absolute tolerance is this
5.11591e_13
   9 = 9 + 1e_12
0
   9 (= !. 1e_11) 9 + 1e_12       NB. Increase tolerance
1
   9 (= !. 0) 9 + 1e_13           NB. Decrease tolerance to 0 (exact comparison)
0
```

5.2.2 Rank (")

The *rank* conjunction (") modifies the verb left argument by changing its rank to that of the numeric noun right argument. The rank of a verb is a vector containing three elements corresponding to the ranks of the monad argument, dyad left argument and dyad right argument. If the right argument has one element, it specifies all ranks; if two, these specify the dyadic ranks. A rank of infinity means that the argument is processed as a whole. The following illustrates the effect of monadic rank

```
   ] m=. i. 2 3         NB. Define 2x3 matrix
0 1 2
3 4 5
   <"0 m                NB. Box scalars
+-+-+-+
|0|1|2|
+-+-+-+
|3|4|5|
+-+-+-+
   +/"0 m               NB. Sum scalars
0 1 2
3 4 5
   <"1 m                NB. Box vectors
+-----+-----+
|0 1 2|3 4 5|
+-----+-----+
   +/"1 m               NB. Sum vectors
3 12
```

```
   <"_ m                 NB. Box whole array
+-----+
|0 1 2|
|3 4 5|
+-----+
   +/"_ m                NB. Sum whole array
3 5 7
```

The following example defines an adverb **ranks** which provides the rank vector of a verb. This definition uses the *explicit definition* conjunction (`:`) with a right argument of 1, which specifies that the object being defined is an adverb. It also uses the *base characteristics* (`b.`) conjunction which provides various information about its verb left argument. This new adverb **ranks** is used to display the standard ranks of `<` (used above) and some other verbs.

```
   ranks=. 'x. b. 0' : 1        NB. Define adverb 'ranks'
   < ranks                      NB. Ranks of 'box/lessThan'
_ 0 0
   * ranks                      NB. Ranks of 'sign/times'
_ 0 0
   %. ranks                     NB. Ranks of 'matrixInv/MatDivide'
2 _ 2
   +/ ranks                     NB. Ranks of 'sum'
_ _ _
   +"1 ranks                    NB. Note how scalar value is extended
1 1 1
   +"1 2 ranks                  NB. Note how dual values are extended
2 1 2
```

Note that the traditional arithmetic dyads (`+ - * % ^`) act on scalar arguments, but their corresponding monads act on entire arrays.

The following shows the effect of rank on *equal* (`=`) and *match* (`-:`):

```
   = ranks                      NB. Like traditional arithmetic dyads
_ 0 0
   -: ranks                     NB. Rank _ means whole arrays
_ _ _
   m                            NB. As above
0 1 2
3 4 5
   m=4
0 0 0
0 1 0
   m = 2 4
0 0 1
0 1 0
   m (="1 1) 3 4 5
0 0 0
1 1 1
   m -: 4
0
   m (-:"0 0) 4                 NB. (-:"0 0) is equivalent to =
0 0 0
0 1 0
   m (-:"0 0) 2 4
0 0 1
0 1 0
   m (-:"1 1) 3 4 5
```

```
0 1
```

One often has to specify `"0` or `"_` in the definition of a verb which is to be applied to arrays of any rank.

Verb rank is part of the essence of J. Iverson (1994) (p89) explains how ranks are described at the top of each dictionary definition. The concept of verb rank is somewhat subtle and beginners often find it difficult to grasp. However it is essential to the full utilisation of the language.

5.2.3 Bond Conjunction (&)

The bond conjunction can be used to define a *monad* in terms of a *dyad* with one argument a constant. This is done using `&` with either argument a verb and the other a noun. Some examples are

```
   log10=. 10 & ^.              NB. Define logarithm to base 10
   log10 100 0.1                NB. Try it
2 _1
   pos=. > & 0                  NB. Define 'pos' which tests whether positive
   pos _2 _1 0 1 2              NB. Try it
0 0 0 1 1
   base8=. 8 & #.               NB. Convert to base 8
   base8 3 5
29
   5 + 8 * 3                    NB. Check
29
   dhm2m=. 1 24 60 & #.         NB. Convert days, hours & minutes to minutes
   dhm2m 2 23 45                NB. Try it with 2 days, 23 hours, 45 minutes
4305
   45 + 60 * 23 + 24 * 2        NB. Check
4305
   quad=. #. & 4 7 _3           NB. Quadratic 4 x^2 + 7x - 3
   quad 2                       NB. Try it with scalar
27
   _3 + 2 * 7 + 2 * 4           NB. Check
27
   ] x=. _3 + i. 7              NB. Define vector
_3 _2 _1 0 1 2 3
   quad x                       NB. Try 'quad' with vector
|length error
|      quad x
   quad ranks                   NB. Check ranks of quad
_ _ _
   quad=. #. & 4 7 _3 "0        NB. Set monadic rank to 0
   x ,: quad x                  NB. Print quadratic table
_3 _2 _1  0 1  2  3
12 _1 _6 _3 8 27 54
```

Note how the verb *base* (`#.`) can be used to define polynomials, but the monadic rank must be set to 0 in order to process non-scalars.

5.2.4 Atop Conjunction (@)

Both arguments of the *atop* conjunction (`@`) are verbs. The resultant verb combines both these by first applying the right argument, then the left. Here are some examples

```
   ] y=. _2 _1 0 1 2               NB. Define a data vector
_2 _1 0 1 2
   v=. %: @ %                      NB. Define verb 'v' as 'sqrt atop %'
```

```
   v y                             NB. Use 'v' monadically
0j0.707107 0j1 _ 1 0.707107
   %: % y                          NB. Same as this
0j0.707107 0j1 _ 1 0.707107
   2 v y                           NB. Use 'v' dyadically
0j1 0j1.41421 _ 1.41421 1
   %: 2 % y                        NB. Same as this
0j1 0j1.41421 _ 1.41421 1
```

Thus `%:@%` represents the monad $\sqrt{1/y}$ and the dyad $\sqrt{x/y}$.

5.2.5 Foreign (`!:`)

The conjunction *foreign* (`!:`) provides facilities for housekeeping and access to the world outside J, including input/output.

These facilities are often used via automatically-defined utility-verbs, so that the beginner (especially with J Release 2 for Windows) may not need to use *foreign* directly. As described in Burke (1994) (chapters 2 and 3), J Release 2 for Windows provides a *Session Manager* which includes menus to set options, access files, etc. Burke (1994) (p6) explains how to automatically execute the J-script `profile.js` to define the standard utilities described in chapters 5 and 6.

The foreign conjunction takes two integer arguments and in most cases produces a verb. The remainder of this section is intended merely to introduce the foreign conjunction and give an idea of the types of facilities it provides.

Detailed documentation for Release 2 is provided in the Appendix of Iverson (1994). For J7 it is provided in Appendix E of Iverson (1993b) and file `xenos.txt`, which is one of the *Distribution Files* included in the ISI diskette (or ftp source tar file).

One can execute system commands from J using the verb *host system command.* For example, we might want to display a directory listing of all files with extension `.js` i.e. J-scripts. Under `MS-DOS` (but not Windows where one would use a menu item) one could do:

```
   2!:0 'dir *.js'
```

Under unix one could do:

```
   0!:0 'ls -l *.js'
```

The fundamental input verb is *file read* (`1!:1`). This defines a character vector containing all the characters in the file, including control characters separating lines. The corresponding output verb is *file write* (`1!:2`). The following example writes a simple character vector (without any control characters) and then reads it back in:

```
   $ msg =. 'Hello world'
11
   filename =. < 'hello.txt'    NB. Boxed filename
   msg 1!:2 filename            NB. File Write
   in =. 1!:1 filename          NB. File Read
   in                           NB. Display 'in'
Hello world
   $ in                         NB. Shape of 'in'
11
```

The verb *erase* (`4!:55`) undefines an object previously defined using *assign* (`=.` or `=:`). This frees the associated memory. Note that the name of the object must be boxed. For example:

```
   x =. 7
   x
7
   4!:55 < 'x'                NB. Erase object from memory (1 indicates success)
1
   x
|value error
```

Another useful housekeeping verb is *time required* (`6!:2`) which gives the CPU seconds used while executing the sentence argument. For example the following shows that it took 0.12 CPU seconds to execute the sentence '`i. 10000`':

```
   6!:2 'i. 10000'
0.12
```

The verb *name class* (`4!:0`) returns an integer code indicating the class (2 = noun, 3 = verb, 4 = adverb, 5 = conjunction) corresponding to a name. For example:

```
   mod =. |
   4!:0 < 'mod'
3
```

The following example uses the verb *set prompt* (`9!:5`) to change the prompt string to '`: `' from the default '   ':

```
   9!:5 ': '

: 3+4
7
:
```

5.2.6 Derivative (`D.`)

This conjunction provides the nth derivative of a verb (function). J can analytically differentiate many verbs and others are differentiated numerically. Iverson (1993a) explains how to define partial derivatives, grad, div, curl, etc. in terms of `D.` . The following shows how to obtain the derivatives of x^3

```
   cube=. ^&3"0         NB. Define scalar (rank 0) cube function
   ] x=. i.5
0 1 2 3 4
   cube x
0 1 8 27 64
   cube D. 1 x          NB. 1st derivative
0 3 12 27 48
   cube D. 2 x          NB. 2nd derivative
0 6 12 18 24
   cube D. 3 x          NB. 3rd derivative
6 6 6 6 6
```

The following shows how to define an adverb `D` which provides the scalar first derivative of a function. It then uses `D` to differentiate the function x^3.

```
   D=. ("0) (D. 1)      NB. Define derivative adverb
   Dcube=. ^&3 D        NB. Use to define verb Dcube as derivative of cube
   Dcube x              NB. Use Dcube
0 3 12 27 48
```

```
   Dcube               NB. Display Dcube using boxed representation
+-------------------+-+-+
|+-------+-+-------+|"|0|
||+-+-+-+|@|+-+-+-+|| | |
|||3|&|*|| ||^|&|2||| | |
||+-+-+-+| |+-+-+-+|| | |
|+-------+-+-------+| | |
+-------------------+-+-+
   5!:5 <'Dcube'       NB. Display Dcube using linear representation
3&*@(^&2)"0
```

These two displays of the verb `Dcube` each have three parts. Firstly `3&*` represents the verb *triple*. Next `^&2` represents the verb *square*. Finally `"0` sets the rank of the resultant verb to 0 i.e. makes it a scalar function. Thus the whole verb means *triple the square* i.e. $3x^2$.

6 Gerunds, Recursion and Iteration

One of the new features in Release 2 is that explicit definitions can contain *control words* such as `if`, `case` and `while`, which define *control structures*. Details are provided by Iverson (1994) (pp19, 130–131) and Burke (1994) (pp46–49).

These control structures will be familiar to those who have programmed in other languages and are appropriate in some situations. However J has alternative facilities which are usually more elegant and efficient.

Gerunds are essentially arrays of verbs. The ability to select a verb (as an array element) provides an alternative to control structures such as *if* and *case*.

There are several alternatives to looping control structures such as *while*. The powerful array manipulation facilities in J obviate much of the need for explicit looping. In some cases recursion (verb calling itself) is a natural way of avoiding looping in J. In most other cases iteration (looping) can be done conveniently using the *power* (`^:`) conjunction.

6.1 Gerunds and Agenda

The Penguin Macquarie Dictionary defines *gerund* as "a noun formed from a verb as *running, writing, loving,* etc.". In J a gerund is a noun formed from two or more verbs using the conjunction *tie* (`` ` ``). An important use of gerunds is to provide the functionality of *if* and *case* statements by allowing the definition of verbs which select from other verbs.

Let us define a verb corresponding to the following function

$$y = \begin{cases} x^2 & \text{if } x > 0 \\ -x & \text{otherwise} \end{cases}$$

```
   g =. - ` *:          NB. Form gerund from 'negate' and 'square'
   type 'g'
noun
   g

+-+--+
|-|*:|
+-+--+


   1{g

+--+
|*:|
+--+
```

Now we use the *agenda* (`@.`) conjunction to define a verb which selects an element from a gerund. We use the verb `pos` defined above to provide the index of the element in the gerund from the argument of the required verb.

```
   y=. g @. pos " 0     NB. Use agenda to define required scalar verb 'y'
   ] x=. _5 + i. 11     NB. Test with these values
_5 _4 _3 _2 _1 0 1 2 3 4 5
   y x
5 4 3 2 1 0 1 4 9 16 25
   x ,: y x             NB. Use ',:' to combine into matrix
_5 _4 _3 _2 _1 0 1 2 3  4  5
 5  4  3  2  1 0 1 4 9 16 25
```

Sometimes we want to use a variable other than an argument to select an element from a gerund (as an alternative to a *case* control structure). This can be done using *evoke gerund (append)* (`` `:0 ``) as shown in the following:

```
   g=. !`|`%            NB. Define gerund from three verbs
   case=. 2             NB. Case 2 is reciprocal
   ((case{g)`:0) 4
0.25
   case=. 0             NB. Case 0 is factorial
   ((case{g)`:0) 4
24
```

6.2 Recursion

J allows verbs to call themselves. Such verbs are said to be *recursive*. Recursive algorithms are heavily used in applications such as translators and combinatorics.

We have already come across examples of explicit definitions of ambivalent verbs where the *monad* is defined in terms of the *dyad* or vice versa. This is a kind of *quasi-recursion*.

It is hard to find simple realistic examples of proper recursion. An artificial example used in countless textbooks is our old friend *factorial*, which can be defined recursively as follows

$$f = \begin{cases} 1 & \text{if } x = 0 \\ xf(x-1) & \text{otherwise} \end{cases}$$

The following shows how this is done in J:

```
   1: _5                          NB. '1:' is verb giving result 1 for any argument
1
   1: 'abc'
1
   f=. 1: ` (* f@<:) @. pos " 0   NB. Define factorial verb 'f'
   x ,: f x                       NB. Test
_5 _4 _3 _2 _1 0 1 2 3  4   5
 1  1  1  1  1 1 1 2 6 24 120
```

Note that the parenthesised expression `(* f@<:)` is a *hook* which multiplies the argument by the result of applying `f` to the decrement (`<:`) of the argument. Remember that `f@<:` is evaluated first because `@` is a conjunction and therefore has higher precedence than verbs. Again we need to specify `"0` to indicate that the function is scalar.

Another slightly less artificial example is that of *Fibonacci Numbers* 1 1 2 3 5 8 13 21 ... where each value is the sum of the two preceding it.

```
   fib234=. '+/ fib y. - 2 1' : ''      NB. Verb giving elements 2, 3, 4, ...
   fib=. (1: ` fib234 @. (1&<)) " 0     NB. All cases
   fib 0
1
   fib 1
1
   fib 2
2
   fib 3
3
   fib 4
5
   fib i. 9
1 1 2 3 5 8 13 21 34
```

Here the verbs `fib234` and `fib` are mutually recursive i.e. each calls the other.

6.3 Power and Iteration

6.3.1 Power Conjunction

We sometimes want to apply a verb several times. For example:

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

12 13 14 15
16 17 18 19
20 21 22 23
   sum=. +/             NB. Define sum
   sum a                NB. Apply sum once
12 14 16 18
20 22 24 26
28 30 32 34
   sum sum a            NB. Apply sum twice
60 66 72 78
   sum sum sum a        NB. Apply sum 3 times
276
   sum sum sum sum a    NB. Apply sum 4 times (same result)
276
```

In such a situation we can use the *power* conjunction (`^:`) which defines a verb which repeatedly applies the verb left argument. If the right argument is a noun, it defines the number of repetitions (iterations). Instead of the above we could do:

```
   sum ^: 0 a           NB. Apply sum 0 times - gives 'a' itself
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
   sum ^: 1 a           NB. Apply sum once
```

```
12 14 16 18
20 22 24 26
28 30 32 34
   sum ^: 2 a           NB. Apply sum twice
60 66 72 78
   sum ^: 3 a           NB. Apply sum 3 times
276
   sum ^: 4 a           NB. Apply sum 4 times (same result)
276
   sum ^: _ a           NB. Same result with any larger number (even infinity!)
276
```

Since the sum of a scalar is itself, any further applications keep producing the same value. And once a value produces itself, there is no need to continue. Thus an infinite number of iterations (as at the end of the above example) can be interpreted to mean
Continue until two successive values are equal.

6.3.2 Obverse Verbs (Inverse Functions)

The power conjunction can have a negative right argument. In this case its magnitude specifies the number of applications of the *obverse* verb, which is normally the *inverse* function. Note the connection with the notation $\cos^{-1} x$ for the inverse of $\cos x$.

Users can use the *obverse* (`:.`) conjunction to specify an obverse as part of the definition of a verb. J is able to determine an analytic inverse for many verbs (but there seem to be problems with forks, etc.). Other cases are handled by a numerical approximation. The following example is based on the function x^2 whose inverse is $\sqrt{x}$.

```
   *: ^: _1             NB. Inverse of square (*:) is sqrt (%:)


+--+
|%:|
+--+


   (*: ^: _1) 2         NB. Inverse square (i.e. sqrt) of 2
1.41421
   ] n=. _2 + i. 5      NB. n is number of iterations
_2 _1 0 1 2
   n ,: *: ^: n 2       NB. Produce table
     _2      _1 0 1  2
1.18921 1.41421 2 4 16
   2 ^ 0.25 0.5 1 2 4   NB. Check
1.18921 1.41421 2 4 16
```

6.3.3 Solution of Non-linear Equations

An important application of iteration is the solution of non-linear equations. One of the simplest techniques is to rewrite the equation in the form $x = F(x)$ We choose some initial approximation x_0 and then calculate a better estimate using $x_1 = F(x_0)$. The process is repeated until a sufficiently accurate result is obtained. The following example can be found in both Iverson (1993b) (section 24) and Iverson (1994) (p12). The equation to be solved is $x - \cos x = 0$ which can be rewritten as $x = \cos x$

```
   x=. x0=. 1           NB. Initial estimate is 1
   ] x=. cos x          NB. Improve estimate
0.540302
   ] x=. cos x          NB. Improve estimate
0.857553
   ] x=. cos ^: _ x     NB. Continue until convergence
0.739085
   cos x                NB. Check result
0.739085
```

This technique has severe limitations. It is not always possible to rewrite equations in the required form. Its rate of convergence is usually poor and it often fails to converge at all.

The Newton-Raphson Method is much faster and more reliable. It uses the iterative formula

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

Let us use this method to solve the same equation solved above. We have

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

```
   f=. 'y. - cos y.' : ''                NB. Define f
   Df=. '1 + sin y.' : ''                NB. and its derivative
   nrf=. 'y. - (f y.) % Df y.' : ''      NB. Newton-Raphson formula
   nrf ^: _ x0                           NB. Apply this until it converges
0.739085
```

This can be simplified using tacit definitions as follows

```
   f=. - cos                   NB. Define 'f' using hook
   Df=. >: @ sin               NB. Define 'Df'
   nrf=. - (f % Df)            NB. Newton-Raphson formula using hook & fork
   limit=. ^: _                NB. Define adverb 'limit'
   nrf limit x0                NB. Apply 'nrf' until it converges
0.739085
```

But we have already seen that the conjunction `D.` can be used to differentiate functions automatically. Let us incorporate this facility into a generic Newton-Raphson adverb `nr`.

```
   D=. ("0) (D. 1)             NB. Define derivative adverb 'D' (as above)
   nr=. '-(x.%x.D)' : 1 limit  NB. Define adverb 'nr'
   fnr=. f nr                  NB. Apply this to f to define verb 'fnr'
   fnr x0                      NB. Apply 'fnr' to initial estimate
0.739085
   (-cos) nr x0                NB. Repeat using hook to define function
0.739085
```

6.3.4 Iteration with Non-scalars

So far we have only considered iterative processes involving scalar nouns and verbs. Some problems require the transmission of several values from one iteration to the next. This can always be done using arrays, boxed if necessary. The following example again uses *Fibonacci Numbers*, but this time instead of using recursion to generate scalars, we use iteration to generate vectors.

```
   tail2=. _2&{.              NB. Define verb 'tail2' giving final 2 elements
   next=. +/ @ tail2          NB. Define verb 'next' giving sum of final 2 elements
   Fib=. , next               NB. Define verb 'Fib' using hook to append next
   fib01=. 1 1                NB. Initial sequence
   Fib ^: 9 fib01             NB. 9 iterations of 'Fib'
1 1 2 3 5 8 13 21 34 55 89
```

6.3.5 Alternative to While Loop

But what if we want to stop when say an element exceeds 1000? The power conjunction with a *verb* right argument provides an alternative to a *while* loop. The number of iterations comes from applying this verb to the right hand argument of the resultant verb. The following shows how this can be used to stop the generation of Fibonacci numbers once 1000 is exceeded.

```
   while=. 1000&> @ {:         NB. Define verb 'while' so 1=continue, 0=stop
   Fib ^: while limit fib01
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The adverb 'limit' is defined as `^:_1` so we have two levels of iteration. The inner level is controlled by the verb `while` which normally gives the value 1, which means *Do a single iteration.* At the end `while` gives the value 0 which means that the argument is simply copied to the result, giving two successive identical values. This causes the outer iteration to terminate.

References

Burke, C., 1994: *J User Manual.* Iverson Software Inc., Toronto.

Hui, R. K., 1992: *An Implementation of J.* Iverson Software Inc., Toronto.

Iverson, K. E., 1993a: *Calculus.* Iverson Software Inc., Toronto.

Iverson, K. E., 1993b: *J Introduction & Dictionary.* Iverson Software Inc., Toronto.

Iverson, K. E., 1994: *J Introduction and Dictionary.* Iverson Software Inc., Toronto.

Smillie, K. *Some Notes on Introducing J with Statistical Examples.* Dept. Computing Science, University of Alberta, Edmonton, Alberta T6G 2H1, Canada, June 1993.