

Easy-J

**An Introduction to the World's most
Remarkable Programming Language**

by Linda Alvord and Norman Thomson

October 2002

Contents

Introduction	1
Section 1. Getting Started	2
Section 2. Introducing Simulation	13
Section 3. Defining your own verbs	16
Section 4. Fitting equations to data	24
Section 5. Boxes and Records	26
Section 6. Investigating Possibilities	29
Section 7. Editing, System Facilities and Scripts	33
Section 8. Drawing graphs	36
Section 9. Rank	39
Index	43
Vocabulary	44

Introduction

J is both a language and an exceptional programming package which provides a highly concise notation for specifying much that is done routinely in the day to day business of computing, such as sorting, searching, updating and restructuring data. Its inventor and designer is Dr. K.E. Iverson, who also devised the language APL, out of which J developed. The first J interpreters appeared around 1990, since when the language has grown in popularity and application, particularly in the world of finance, where its conciseness and power for rapid algorithm development is highly valued. Amazingly, this algorithm-rich software is available free by download from

www.jsoftware.com

a download which takes only a few minutes, following which installation is easy.

J also has many enthusiasts in education, where it can be a powerful motivator on account of the clarity with which users can express their intentions on a computer.

The objective of this tutorial is to give a brief introduction which will either encourage the user to perform the above download and discover that the claims which are made on this page are in no way understated, or, if the download has already been performed, encourage him or her to take the first steps up the J learning ladder into a world of discovery and delight. It is impossible in a pamphlet of this size to cover anything other than a tiny part of the facilities which J affords. However, that need be no disadvantage, since the J system is fully self-describing through a comprehensive Help facility, through Labs (that is prepared sessions) which can be invoked by the Studio drop down menu, and through libraries of scripts and packages which encapsulate the work of many previous users. Once the topics in this tutorial have been grasped, the user should find little difficulty in extending his or her knowledge both by exploiting the help features listed above and by direct exploration on the keyboard. Further, the J language is totally consistent across all the many platforms on which it is now available, and a vigorous Internet forum (forum@jsoftware.com) is perhaps the best support mechanism of all.

SECTION 1 : GETTING STARTED

This tutorial assumes that the reader has either

- (i) successfully installed J on a computer and is ready to use it; or
- (ii) is interested in J as a vehicle for reading and writing algorithms and wishes to obtain something of its flavour prior to using it on a computer.

J in its simplest use acts just like a desk calculator. If you are working at a terminal, type the following input, pressing the “enter” key after each line :

```
3+4
7
5*20
100
```

Symbols like + and * for *plus* and *times* in the above phrases are called verbs and represent functions. You may have more than one verb in a J phrase, in which case it is constructed like a sentence in simple English by reading from left to right, that is $4+6\%2$ means 4 added to whatever follows, namely 6 divided by 2 :

```
4+6%2
7
```

Here is another phrase :

```
4%6+2
0.5
```

In a similar way this means 4 divided by everything to the right, namely $6+2$ which is 8. Hence $4\%6+2$ evaluates to $4\%8 = 0.5$. Notice how the verbs themselves are executed in left to right order, that is the rightmost verb + is executed before the leftmost, %. The simplicity of these rules for reading phrases on the one hand and executing them on the other avoids the need for artificial rules of precedence such as apply in ordinary arithmetic to state that for example * has priority over +.

Pursuing the analogy with English, the sentence “Jack visited the house which he built” is read from left to right, but in order to “execute” it you must do the “build” first, so that you can properly identify which house was visited.

The repertoire of basic arithmetic verbs + - * % in J is completed with the *power* verb ^ :

```
4^3
64
```

If you wish to change the order of execution of verbs, you may do so in the normal way by using parentheses :

```
1+2%3
1.66667
(1+2)%3
1
```

You might wonder why the first of these answers was represented to 6-digit precision rather than any other. The reason is that print precision is controlled by using a so-called foreign conjunction – foreign because it “belongs” to the system rather than to the J language proper. The default value is 6 can be confirmed by typing

```
9!:10 ''          NB. get the current print precision
6
```

and reset to other values, say 4, by :

```
(9!:11)4         NB. set print precision
1+2%3
1.667
```

You should think of `9!:10` and `9!:11` as further verbs which perform system functions within the J workspace environment. The above lines also introduce the symbol `NB.` which indicates that everything to its right on a line should be read as a comment.

The underbar symbol `_` is used to indicate “negative”, and is an inseparable part of a negative number. You may not leave a space between the underbar and the digits of a number.

```
3-8
_5
3-_8
11
```

Within the workspace, data is stored by assigning values to variables with names chosen by the user, subject to the requirement that the first character in such a name must be alphabetic :

```
b=.4
value1=._0.3    NB. decimals <1 must have leading 0
value1=.1.6     NB. a second assignment to value1
```

Single-value items such as the above are known as “scalars”. Entry of a name by itself causes the most recently assigned value of the named variable to be output :

```
b
4
value1
1.6
```

However, J is much more than just a calculator, since arbitrarily large *lists* of numbers (which are also sometimes called “vectors”) can be assigned :

```
c=.3 1 4 0.5 _2
d=.4 0 3 _1.2 7
```

Again the underbar symbol is used to express negative numbers. Adding two lists means that corresponding items in each are added:

```
c+d
7 1 7 _0.7 5
c*d
12 0 12 _0.6 _14
d%c
1.333 0 0.75 _2.4 _3.5
```

Dividing *c* by *d* involves a division by 0 in the second position. The result of this is infinity, denoted by a single underbar :

```
d%c
0.75 _ 1.333 _0.4167 _0.2857
```

There are some list operations which are not meaningful, for example adding a list with five items to one with only three. If you attempt to do this the result is as follows :

```
c+1 2 3
|length error
| c +1 2 3
```

Three observations should be noted about this error message :

- (a) it is concise;
- (b) the word “length” gives insight into the nature of the error; and
- (c) the added blank spaces indicate the exact point in the phrase where, reading from right to left, the error was detected.

J allows complex numbers, so for some verbs such as *square root*, denoted by the digraph %:, may have results with a number separated without spaces by the letter j. The number 0j1.414 below indicates a real component of 0 and an imaginary component of 1.414.

```
%.c NB. square roots of c
1.732 1 2 0.7071 0j1.414
```

The *natural logarithm* verb ^. produces logarithms to the base e, and can also generate results containing complex numbers :

```
^.c NB. natural logarithms
1.099 0 1.386 _0.6931 0.6931j3.142
```

Logarithms to base 10 are obtained by :

```
10^.c
0.4771 0 0.6021 _0.301 0.301j1.364
```

and similarly for logarithms to any other number bases, e.g. 2 :

```
2^.1 2 4 8 20
0 1 2 3 4.322
```

If you wish to count or *tally* # the number of items in a list, type :

```
#c
5
```

The verb # is treated like any other verb, so 2 plus the tally of items in c is :

```
2+#c
7
```

The verb *from* { references items in a list. The fourth item in list c is :

```
3{c
0.5
```

You probably expected the answer `_2`; what you must take into account is that items in lists in J are always indexed from 0. A list can be used to select from a list :

```
0 3 4{c          NB. 0 3 4 is a list of indices
3 0.5 _2
```

Data need not be numeric – in the next example a list of characters is defined, and the characters themselves are tallied:

```
cv='J is useful.'
cv
J is useful.
#cv
12
```

cv can be indexed by a numeric list containing repeated items :

```
0 7 10 10 2 7 3{cv
Jellies
```

In J a scalar is a single data item which can be either a number, or a literal character, which in turn may be a letter of the alphabet, digit or symbol. A numeric scalar item is the value of the number represented by the combination of its characters, sign and decimal point. Literal characters are enclosed in single quotes. Thus the tally of the literal items in the representation of the single number `_3.875` is :

```
#'_3.875'  
6
```

A tally of the number as a numeric scalar is :

```
#_3.875  
1
```

Some symbols such as # represent two different verbs. In general, a verb has data on both its left and right, called left and right arguments. Such a verb is called a dyadic verb. A verb with only a right argument is described as monadic, thus square root and tally are monadic verbs. As a dyadic verb # is called *copy*. The left argument of *copy* is the number of copies of each item in the right argument :

```
3#c  
3 3 3 1 1 1 4 4 4 0.5 0.5 0.5 _2 _2 _2
```

The dyadic verb *reshape* verb \$ can create what appear to be matrices, but which are in fact lists of lists. The left argument provides the structure and the right argument gives the data. The data is reused by “wrapping round” :

```
matrix=.3 4$c  
3 1 4 0.5  
_2 3 1 4  
0.5 _2 3 1
```

The monadic verb *shape of* \$ gives the structure of the right argument and is always a list :

```
$matrix  
3 4  
#matrix  
3
```

Note that *tally* counts only the first item in the shape, that is it counts at the topmost level only. For simple lists like c and d their *tally* and *shape* look identical.

Arithmetic verbs (the words “verb” and “function” can often be used interchangeably) may also have one scalar argument and the other a list :

```
c  
3 1 4 0.5 _2  
b  
4  
c+b  
7 5 8 4.5 2  
c*2  
6 2 8 1 _4
```

Technically what happens is that the scalar is replicated (that is extended) to become a list of matching length, and item by item function application takes place as before.

Use the verb *append*, represented by a comma, to join lists and scalars :

```
    c,b
3 1 4 0.5 _2 4
```

Arithmetic verbs have monadic as well as dyadic forms, for example :

```
    +b
4
    -b
_4
```

Monadic + (called *conjugate*) does not change the value of its argument, provided that it is numeric and not complex. However, it can be useful in simultaneously calculating and displaying a value :

```
    +z=.4+6%2
7
```

Monadic minus is called *negate* , and monadic * is called *signum*, which returns 1 for a positive value of its argument, _1 for a negative value, and 0 for a zero value :

```
    *b
1
    *-b
_1
    *b-b
0
```

Monadic % is the function *reciprocal* :

```
    %b
0.25
```

The two digraph symbols <. and >. double up as *minimum* and *maximum* in their dyadic form, and as *ceiling* and *floor* (that is, next integer above and below) in their monadic form. Like the arithmetic verbs, they can be applied to lists as well as to scalars. Here are some examples :

```
    +value1=.1.6
1.6
    >.value1      NB. round up value1
2
    <.value1      NB. round down value1
1
    b>.value1    NB. maximum of b and value1
4
```

```

    b<.value1      NB. minimum of b and value1
1.6
    c              NB. c and d as defined above
3 1 4 0.5 _2
    d
4 0 3 _1.2 7
    c>.d          NB. item by item maxima
4 1 4 0.5 7
    0>.d          NB. d with negative items
4 0 3 0 7       NB.          replaced by zero

```

This is a good point at which to interrupt the description of J verbs with some elementary arithmetic examples which show how J can be applied to simple problems.

(a) Suppose you want to express 1 foot 4½ inches as a percentage of first 2 feet, then 2 ft. 6 ins., 3 ft., 6 ft., and 10 ft., in each case rounding the answer to the nearest percentage above. Using a list allows all five calculations to be done in parallel :

```

>.100*(1+4.5%12)%2 2.5 3 6 10
69 55 46 23 14

```

(b) An Indian is said to have sold Manhattan Island to white settlers in 1650 for 12 dollars. What would be the dollar value of this sum in the year 2000 if invested at compound interest of 3%, 4%, 5%, 10% ?

```

(9!:11)16      NB. set print precision to 16

<.12*(1+0.03 0.04 0.05 0.1)^(2000-1650)
373430 10986263 312921872 3686557834057256

```

(c) A body falling from rest for y seconds drops a height of $\frac{1}{2}gt^2$ cms. where $g=981$ cm./sec². Find the height fallen after 1,2,4,8,16 secs., then the velocities v at these times (the formula for velocity is $v = \sqrt{2gh}$).

```

(9!:11)6      NB. set print precision to 6

+ht=.0.5*981*1 2 4 8 16^2
490.5 1962 7848 31392 125568
+vel=.%:2*981*ht
981 1962 3924 7848 15696

```

Returning to the description of J verbs, the next one to be introduced is | which in its monadic form means *absolute value*, and in its dyadic form the remainder or *residue* when the right argument is divided by the left argument :

```

|c
3 1 4 0.5 2
  1.6|b
0.8

```

Frequently one wants to add, multiply etc. all the items in a list. This process is called *insertion*, and the notation for doing it is

```

    +/c
6.5
    */c
_12
    -/c
3.5

```

The last example deserves a little more attention. A more complete description of insertion is that the symbol / represents an adverb which when applied to a verb modifies its behaviour in some way. The modification for / consists of placing the verb between each item in the list so that +/v is equivalent to

$$3 + 1 + 4 + 0.5 + _2$$

and right to left execution (or alternatively left to right reading) takes place as usual.

Try this with – in the gaps, and it will immediately become clear why –/c is 3.5. You will notice also that 3.5 is the alternating sum of c, that is the sum of the items in odd-numbered positions (first, third, etc.) minus the sum of items in even-numbered items. Following a similar argument, %/c is the alternating product of c. Perhaps even more interesting is the effect of putting >. and <. in the gaps, which produces the largest and smallest items respectively in the list c.

J has a full complement of relational verbs, that is

$$< \quad <: \quad > \quad >: \quad = \quad \sim:$$

standing for *less than*, *less than or equals*, *greater than*, *greater than or equals*, *equals* and *not equals* respectively. These are verbs which give Boolean results representing truth by 1 and falsity by 0. As with arithmetic verbs, corresponding items in lists of the same length are processed in parallel :

```

    c<d
1 0 0 0 1

```

Frequently an expression like the above is used as an intermediate step in a further operation such as a *copy* where 1 means “copy the item” and 0 means “don’t copy” :

```

    (c<d) #c
3 _2
    (c>:d) #c
1 4 0.5

```

Another verb which returns Boolean (that is 0 or 1) values is the *membership* verb e.:

```

    6 e. c
0

```

Read this as a question “is 6 in c?” If the left argument is a list, the question is asked for each item individually :

```

c e. b      NB. c is 3 1 4 0.5 _2, b is 4
0 0 1 0 0
c e. d      NB. recall d is 4 0 3 _1.2 7
1 0 1 0 0

```

The set of “logical” verbs also give Boolean results. They are

```

*.      +.      -.      *:      +:

```

which represent *and*, *or*, *not*, *not-and* and *not-or* respectively. For example :

```

0 1 1 *.1 1 0
0 1 0
-.c<d
0 1 1 1 0
+./0 1 0 1 0
1

```

The symbol *i.* called *index generator*, produces a list of integers starting from 0:

```

i.6
0 1 2 3 4 5

```

If the integer argument is negative the result is a list in descending order :

```

i._6
5 4 3 2 1 0

```

An important special case arises if the argument is 0, when an empty list is generated :

```

i.0      NB. list with zero items
#i.0     NB. tally of empty list is scalar zero
0
$i.0     NB. shape is a one-item list, viz. zero
0
$b      NB. shape of a scalar is an empty list
#b      NB. .. but its tally is 1
1

```

Mathematically inclined readers should also note the following :

```

+ / i.0
0
* / i.0
1

```

These are the “identity values” of the verbs to the left of /, that is, those values $i.d$ for which $x \text{ verb } i.d$ is equal to x for all values of x . Recall that comma means append, so $+/c, x$ and $(+/c)++/x$ must be equal, since the sum of the list c, x is equal to the sum of the sums of its parts. If x is $i.0$ then $+/c, x$ is just $+/c$ and so $+/i.0$ must be the identity value for $+$. A similar argument applies to $*$.

Grids and Tables

It is easy to generalise $i.$ in order to make a list of any equally spaced values, as you might want to do in marking points on the axis of a graph. For example, the integers from -4 to $+4$ are given by

```

_4+i.9
_4 _3 _2 _1 0 1 2 3 4

```

and those from 3 to 9 at intervals of one half by

```

3+0.5*i.13
3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9

```

The 256 ASCII characters are collected together in a constant noun $a.$ called *alphabet*, in which the upper case letters of the usual alphabet commence at the item indexed as 65 and the lower case ones at 97. Hence the lower case alphabetic characters are

```

(97+i.26){a.
abcdefghijklmnopqrstuvwxyz

```

The dyadic form of / gives verb tables, for example the ordinary addition and multiplication tables for the first five natural numbers are, written side by side

```

k=.1 2 3 4 5
k+/k
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
6 7 8 9 10

k*/k
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25

```

Tables can be used as indices and so all of the ideas in this subsection can be put together in the phrase :

```

(k*/k){(96+i.26){a.
abcde
bdfhj
cfilo
dhlpt
ejoty

```

i . also has a dyadic form called *index of* which is somewhat like e . Remembering that the first item has index 0, compare :

```

      c
3 1 4 0.5 _2
      c e.b      NB. is c a member of b(= 4)?
0 0 1 0 0
      c i.b      NB. what is the index of b in c?
2

```

If both arguments are lists

```

      d      NB. reminder!
4 0 3 _1.2 7
      d i.c   NB. what are the d-indices of c?
2 5 0 5 5

```

the three fives (one more than the largest index in c) indicate that the second, fourth and fifth items in c do not occur in d.

The verbs *take* { . and *drop* } . do just this to a specified number of items from a list – from the left of the list if the left argument is positive, and from the right if it is negative :

```

      3{.c
3 1 4
      _3{.c
4 0.5 _2
      3}.c
0.5 _2
      _3}.c
3 1

```

Used monadically, the default left argument is 1. In addition, the verb *tail* { : selects the last item in a list :

```

{:c
_2

```

Take has the further property that “over-taking” is permitted, that is fill items (0 for numeric data, and blank for character) are used to pad out when necessary :

```

      7{.c
3 1 4 0.5 _2 0 0
      _25{.'J is useful'
          J is useful

```

SECTION 2 : INTRODUCING SIMULATION

You have now seen that using J is in many ways more like handling a very sophisticated calculator than a computer. Used in this way, it is a remarkable tool for solving routine problems in many disciplines. The techniques in this section show some ways in which you can take advantage of its capability for taking samples.

Classifying, tabulating and condensing actual or simulated data are all easy in J. Problems involving rolling dice and tossing coins are simple enough to observe what actually happens as well as to predict what you would theoretically expect to happen.

The symbol `?6` is called *roll* as in “roll a die”, so that `?6` models rolling a die with six faces, or more generally randomly selecting one of the first six non-negative integers :

```
?6          NB. first throw
0
?6          NB. second throw
4
```

The faces are usually numbered from 1 to 6. Conveniently, the verb *increment* `>:` adds 1 to each item (Likewise *decrement* `<:` subtracts 1 from each item) :

```
b=.:a=?6    NB. b is one greater than a
a
4
b
5
```

Suppose we had ten dice. J extends *roll* to a list of integers so that the computer simulates the throws of ten dice, or equivalently ten throws of a single die :

```
>: ?6 6 6 6 6 6 6 6 6 6
4 2 1 5 5 6 3 4 5 1
```

Suppose that we now simulate a multiple choice test in which there are ten questions each with five options :

```
+t=.:?10#5
3 4 4 4 1 2 2 0 2 2
```

To convert these into, say, letters of the alphabet use *from* :

```
t { 'abcde'
deeebccacc
```

In a similar way simulate the tossing of six coins where 0 represents a tail and 1 a head. First make six copies of the integer 2 :

```
6#2
2 2 2 2 2 2
```

and then roll (that is) toss each of them :

```
(?6#2){'th'  
hhthth
```

Often what is interesting is aggregates rather than the individual rolls. For example you might want to simulate several times the gender distribution of 1000 births in a hospital where 0 represents a girl and 1 a boy :

```
+/?1000#2  
501  
+/?1000#2  
480
```

All the above examples have used roll monadically. The dyadic case is called *deal* , for which the difference is that the left argument gives the number of draws and also the values of *i* . right argument are progressively deleted.;

```
6?6  
4 1 0 3 5 2
```

This means that the right argument must be at least as great as the left argument, otherwise the left argument is outside the domain, that is, the permitted values :

```
7?6  
|domain error  
| 7 ?6
```

There are no limits to the shape of the structure of the units from which the draws are to be made. For example, suppose the multiple choice test above was taken by each of five children, so that the simulation is now one of six ten-item lists, each item within which is a random draw from *i* . 5 :

```
u=.?6 10$5  
4 2 3 4 3 4 3 3 0 4  
2 2 0 3 2 3 3 0 4 4  
4 2 0 2 4 1 2 1 2 4  
2 2 4 1 1 4 0 3 3 0  
0 3 3 1 2 3 3 1 4 3  
4 0 0 3 3 1 1 0 1 4
```

Now make a draw from each of these, and translate into letters as before :

```
u{'abcde'  
ecdededdae  
ccadcddaee  
ecacebcbce  
ccebbeadda  
addbcddbed  
eaadbbabe
```

A nice application of this technique is to simulate hands of 13 cards from a pack. First assume that the cards are ordered in the pack from smallest to highest, and in suit order Diamonds, Clubs, Hearts, Spades. (Whether any actual pack is physically ordered in this way makes no difference to the quality of the simulation.). A draw is then a choice of 13 integers from 0 to 51 without replacement :

```
+hand=.13?52
43 50 25 45 40 4 9 11 24 42 17 19 37
```

Next calculate the suits by dividing these values by 13 and rounding down (the monadic is called *ravel* and causes the value of its argument, in this case `suits`, to be displayed as a simple list) :

```
,suits=(.<.hand%13){'CDHS'
SSDSSCCCDSDDH
```

Now calculate the values by taking remainders on division by 13 :

```
,values=(13|hand){'23456789TJQKA'
6KA836JKK568K
```

Finally use a verb *laminare* (`, :`) to align the cards nicely in columns :

```
values,:suits NB. 6 of spades, King of spades etc.
6KA836JKK568K
SSDSSCCCDSDDH
```

Simple counts of combinatoric items are obtained by the verbs *factorial* ! and its dyadic form *out of*, which gives the number of combinations of r objects out of n :

```
!i.6 NB. factorial 0 thru factorial 5
1 1 2 6 24 120
2!2+i.6 NB. e.g.21=no of selections of 2 out of 7
1 3 6 10 15 21
```

Pascal's Triangle

The arrangement of integers known as Pascal's triangle in which the binomial coefficients appear in columns is constructed using the table :

```
(i.8)!/i.8
1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7
0 0 1 3 6 10 15 21
0 0 0 1 4 10 20 35
0 0 0 0 1 5 15 35
0 0 0 0 0 1 6 21
0 0 0 0 0 0 1 7
0 0 0 0 0 0 0 1
```

SECTION 3 : DEFINING YOUR OWN VERBS

The starting point is again the list

```
      c
3 1 4 0.5 _2
```

In section 1 we saw how the expressions `>./` and `<./` supplied the maximum and minimum values respectively, and the range of `v` is simply the difference between these two values. `J` allows us to express this as

```
>./ - <./
```

using a verb structure which is known as a fork :

```
(>./-<./) c
6
```

Instead of constantly having to use parentheses to write this compound verb, it can be consolidated by assignment to a user-defined name :

```
range= >./-<./
range c
6
```

One of the most important features of the verb which has just been defined is that, unlike almost every other programming language you are likely to have encountered, there is no explicit reference to data (that is, arguments) in the definition. This particular style of programming is known as tacit programming.

Another example of tacit programming, which also exhibits the fork structure is

```
mean= .+/%#
mean c
1.3
```

The central operation, divide, is performed on two functions, namely the sum `+/` and the tally `#` of the data. This is just what in mathematics might be written $\frac{\sum x}{n}$, only as noted above, the `J` definition makes no reference to data such as x and n .

A composite verb such as `+/ %#` consists of three verbs written one after the other. When two rather than three verbs are written together the combination is called a hook. The verbs concerned may be primitive (that is part of the basic `J` repertoire) or user-defined, and so a simple example of this structure is `-mean`.

`(-mean) c` means `c-mean c`, that is the single argument is used first as a monadic argument to the right verb, and then as the left argument of the left verb. In general, for verbs `u` and `v`, the hook `(u v)y` is equivalent to `y u v(y)`. So define

```

mdev=. -mean      NB. deviations from the mean
mdev c
1.7 _0.3 2.7 _0.8 _3.3

```

It is natural to ask what happens if there are four verbs in a row, say u v w x. The answer is that the three rightmost are resolved as a fork leaving two verbs to form a hook u (v w x) :

```

(-+/%#) c
1.7 _0.3 2.7 _0.8 _3.3

```

Similarly five verbs resolve into two forks, and so on.

Those who have experience of writing programs may protest that “real” programs of any magnitude consist of actions in sequence - do this, *then* this, *then* this ..., just like main verbs in stories. However, there is another way in which verbs can be combined in English, which becomes apparent when you think of verbs such as ‘stirfry’ and ‘sleepwalk’. If you fried the food completely and then stirred it, there is no way in which you could be said to have ‘stirfried’ it! Rather the two verbs ‘stir’ and ‘fry’ are blended together or fused in a manner which says that the primary verb ‘fry’ is elaborated with an action ‘stir’ which must take place concurrently with it. Similarly with ‘sleepwalk’, the primary action is walking, only it is a new kind of walking, which happens when sleeping and walking take place simultaneously. Yet another example which emphasises concurrency even more directly is the verb ‘timestamp’. Were the event to be timed *after* the stamping, the two activities of timing and stamping would not take place in exact synchronisation, and similarly if the stamping were to take place after the timing.

A generic feature which all these compound verbs have in common is that of ‘joining’ simple verbs, which leads, by analogy with usage in English grammar, to the use of the term conjunction. Further there is a generic property involved in the type of concurrent fusion which is implicit in all three compound verbs described above; this idea is consolidated in the specific conjunction @ which is called *atop*. Thus, in a pidgin mixture of English and J :

stirfry =. stir @ fry, sleepwalk =. sleep @ walk, timestamp =. time @ stamp

Contrast this with the kind of sequencing which takes place in a childlike narrative :

“We got dressed, then we ate breakfast, then we cleaned our teeth, then we walked to the woods, then we picnicked”

where one thing happens strictly after another in sequence. Another J conjunction called *at* @: deals with this method of joining verbs, so that, using the pidgin mixture above, and taking account of the fact that “after” in English requires a verb reversal compared with “then”, the above story could be related :

“We picnicked @: walked @: cleaned teeth @: ate @: got dressed”

Reflection on ordinary linguistic experience thus shows that in combining verbs there is an implicit distinction between fusion with concurrency on the one hand, and strict sequencing on the other. Because J is of necessity a precise language you are required to distinguish explicitly between these two cases when you choose to combine verbs, which, as we have already seen, is an inevitable activity in writing your own programs. Thus we look next at how the two J conjunctions, @ and @: are used in programming practice. First, here is a new verb *square* whose symbol is *:

```
*:c
9 1 16 0.25 4
```

Suppose that we wish to total these squared values. The composite operation “total-atop-square” which is a fusion of total and square means that the combined verb is applied to every item in the argument list. Since “total” can be perfectly reasonably applied to a single number *x*, that is *+/x* is just *x*, the effect of “total-atop-square” is no different from square by itself:

```
(+/@*: )c    NB. total atop square for each item
9 1 16 0.25 4
```

The other interpretation, and in practice the more likely one, is that “square” should produce an intermediate result before “total” is applied, that is, as in the children’s narrative, we seek “total-after-square”. In this case @: must be used:

```
(+/@:*)c    NB. total after square for entire list
30.25
```

If you like, you can think of @: as providing a weaker linkage between the verbs than @ does. In order to sum the squares of deviations from the mean, say

```
(+/@(*:@mdev) )c
21.8
```

which provides the definition of a “sum of squares” verb:

```
ssq=.+/@(*:@mdev)
ssq c
21.8
```

Conjunctions are always resolved at the earliest possible point on a left to right reading scan, so that if the parentheses were to omitted in the above definition, the meaning would be *(+/@(*:))@mdev* which, as described above, results in totalling being applied to individual items:

```
ssq1=.+/@*: @mdev
ssq1 c
2.89 0.09 7.29 0.64 10.89
```

Following the discussion of grids in section 1, here is how a monadic verb *grid* might be developed, given its argument is to be a three-item list consisting of start

point, interval width, number of items. First use the hook (**i.*) to construct the intervals, then convert this to a fork using *take* and *drop* in order to make *ti* monadic :

```

    2(*i.)5          NB. 2 times 0 1 2 3 4
0 2 4 6 8
    ti={.( *i.)}.   NB. ti=times index generator
    ti 2 5
0 2 4 6 8

```

Apply this technique a second time to introduce the location parameter, and recognise also the fact that *ti* must be applied after the scale and size parameters have been extracted by *drop* :

```

    grid={.+ti@}.
    grid 97 2 5
97 99 101 103 105

```

It may have struck you in comparing J verb definitions with conventional program writing, that although argument data is excluded from the former, some programs necessarily involve constants. For example, to write a verb to round numbers to a given number of decimal places, use of the number 10 is unavoidable. A possible starting point is a hook which multiplies *x* (left argument) by “10 to the power..” J allows you to “bind” the constant 10 to the power verb with a conjunction *bond* & .

```

    up=.*10&^      NB. x times 10 to the power y
    3.757 4.232 up 2
375.7 423.2

```

Simple rounding, that is to the nearest integer, consists of adding 0.5 to a number and taking its floor. This calls for another *bond* to define a verb *rnd* for simple rounding :

```

    rnd=.<.@(0.5&+)  NB. add a half and round down
    rnd 3.4
3

```

and the next step is to combine rounding with “upping” in an “after” relationship :

```

    3.757 4.232 (rnd@up) 2
376 423

```

By analogy with *up* define

```

    down=.%10&^      NB. x divided by 10 to the y

```

so that the desired result is

```

    (3.757 4.232 (rnd@up) 2) down 2
3.76 4.23

```

Notice that the parameter 2 is used as a right argument twice in the above expression. To permit this reuse J has two verbs *left* and *right* written as [and] respectively which extract the right and left arguments. Using these, the operations embedded in the above line are brought together in the form of a fork :

```
round=.rnd@up down ]
3.757 4.232 round 2
3.76 4.23
```

Explicit programming

When you start to write programs which are appreciably larger than those of the preceding subsection, joining verbs correctly can begin to make demands on mental ingenuity. Accordingly, a more conventional style of defining programs is allowed which allows left and right argument data to be referred explicitly as *x.* and *y.* respectively. Redefining *ssq* in this style should make things clear :

```
ssq=.monad define
+/*: mdev y.
)
ssq i.5
10
ssq c
21.8
```

Notice that *mdev* is still defined tacitly showing that the programmer is free to mix explicit and tacit styles in whatever way he or she find most comfortable. In the case of *round* we have a dyadic verb :

```
round=.dyad define
(rnd x. up y.)down y.
)
3.757 4.232 round 2
3.76 4.23
```

In this case explicit programming avoids the need to use the conjunction @ which was necessary in the earlier tacit definition. If you find conjunctions are tricky to master, then the ability to switch between explicit and tacit styles can be invaluable. J even provides the capability to “translate” an explicit verb automatically into a tacit one as in the following dialogue :

```
9!:3(5)      NB. set system for linear display of verbs

13 : '(rnd x. up y.)down y.'
([: rnd up) down ]
```

9!:3 is a foreign conjunction like 9!:11 which was used earlier to set the print precision. The available parameters are 2, 4 and 5, which instructs the system to set verb display to boxed, tree and linear formats respectively.

13 : is a code which requests the translation of whatever explicit definition string follows in quotes. The result includes a new symbol *cap* [: which is necessary because the translator resolves everything in terms of forks.. When a monadic function like *rnd* is encountered, a dummy symbol is necessary to fill the place of the left tine of the fork.

Verbs for sorting

J contains two primitive verbs for sorting, namely *grade up* / : and *grade down* \ : . These return the permutations which would cause the list given as right argument to be sorted in ascending (descending) order :

```

/:c          NB. upward sorting permutation
4 3 1 0 2
  4 3 0 1 2{c
_2 0.5 3 1 4  NB. c sorted upwards

\ :c        NB. downward sorting permutation
2 0 1 3 4
  2 0 1 3 4{c
4 3 1 0.5 _2  NB. c sorted downwards

```

Each of these pairs of steps can be reduced to a hook. But first observe that the arguments of any dyadic verb can be switched from left to right by applying an adverb *reflex*, so that the immediately preceding result could also have been achieved using the verb {~ :

```

c{~2 0 1 3 4
4 3 1 0.5 _2  NB. c sorted downwards

```

Recall that a hook is a composite verb defined by writing two verbs one after the other so that (u v)y is equivalent to y u v(y). Thus combining the verbs {~ and / : gives

```

({~/:)c
_2 0.5 1 3 4  NB. c sorted upwards

```

leading to user-defined verbs

```

sortu={~/: NB. sort upwards
sortu c
_2 0.5 1 3 4

```

and

```

sortd={~\: NB. sort downwards
sortd c
4 3 1 0.5 _2

```

A further primitive verb *nub* removes duplicates from a list :

```
~. 4 6 2 4 4 6 2 5 4
4 6 2 5
```

so that an upwardly sorted list with duplicates removed is given by `sortu atop nub`. As you have already seen `atop` is expressed by the conjunction `@`

```
(sortu@~.)4 6 2 4 4 6 2 5 4
2 4 5 6
```

This can be made into another user defined verb :

```
unub=.sortu @ ~. NB. upwardly sorted nub
unub 4 6 2 4 4 6 2 5 4
2 4 5 6
```

A verb for reversing and shifting

The verb `|.` in its monadic *reverses* a list, and in its dyadic form performs a *shift*, to the left if the left argument is positive, and to the right if it is negative :

```
|.c NB. reverse c
_2 0.5 4 1 3
|.'J is useful.'
.lufesu si J
2|.1 2 3 4 5 NB. shift >:i.5 two places to left
3 4 5 1 2
_1|.1 2 3 4 5 NB. shift >:i.5 one place to right
5 1 2 3 4
```

The principle remains the same even if the argument is a list of lists :

```
]m=.3 5$'rosessmellsweet' NB. list of three words
roses
smell
sweet
1|.m NB. shift words one place up
smell
sweet
roses
```

Suppose that a verb is wanted which removes duplicate blanks from sentences. A first step is to compare the sentence as a character list with itself shifted one place right, and marking where matching items are not equal :

```
test=.~::~~1&|.
test s='how are you today ?'
1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1
```

Then use copy to retain only the marked items :

```
(#~test) s
how are you today ?
```

A flaw in the above verb is that it will remove duplicates of any character, not just space. This can be adjusted for by making a fork whose centre verb is *or +*. and which does a *not equal* *~ :* comparison with space :

```
s1=. 'marry me, harry !'
(#~test+.~:&' ') s1
marry me, harry !
```

so the final defined verb for RemoveDuplicateBlanks is `rdb=.#~test+.~:&' '`.

Conditional structures in J are catered for using a mechanism called a gerund, in which a succession of verbs are separated by the conjunction *tie`*, followed by another conjunction called *agenda @.*, followed by a verb whose result is an integer index into the tied verb list. The overall structure is what is known in programming as a case statement. As a simple example applied to an if-then-else situation, consider how the median of a list of ordered numbers might be programmed. If the tally of the argument list is odd, then the median is the value of the middle number. If it is even the median is the mean of the values of the two middle numbers.

A first step is to obtain the index of the median value or values. Define

```
half=.-:@<:          NB. halve one less than rt. argument
```

If the argument is odd, the result is a single integer. However, if the argument is even the result is a fraction *n.5* and what is wanted is both the *floor* and *ceiling*. The condition “is-odd” is the result of `2&|` whose result must be 0 or 1 which are appropriate indices for the two case verbs and leads to the following definition :

```
medind=.((<.,>.)@half) ` half @.(2&|)
medind # 1 3 4 7
1 2
```

Next apply the index(es) using *from* and finally use `mean` defined above :

```
(mean@:{~medind@#) 1 3 4 7
3.5
```

As a final step the verb `sortu` can be incorporated, so that the requirement that the list be ordered initially can be removed :

```
median=(mean@:{~medind@#)@sortu
median 7 4 1 3
3.5
```

SECTION 4 : FITTING EQUATIONS TO DATA

One of the most powerful mathematical J primitive verbs is *matrix inverse*, which also provides least squares fitting in its dyadic form. As an example, consider the simultaneous equations

$$\begin{aligned}x + 2y - 3z &= 15 \\x + y + z &= 12 \\2x - y - z &= 0\end{aligned}$$

The coefficient matrix of the left hand side is :

```
+m=.3 3$1 2 _3 1 1 1 2 _1 _1
1 2 _3
1 1 1
2 _1 _1
```

Its inverse is

```
%.m
0 0.3333 0.3333
0.2 0.3333 _0.2667
_0.2 0.3333 _0.06667
```

The solution of the equations is :

```
15 12 0%.m
4 7 1
```

The functionality of % . does not stop here. Suppose

```
x, :y          NB. x and y as laminated lists
2.1 2.4 3.6 3.7 4.3 5.1 5.5 5.8 5.9 6.6 7.4 8.2
4.1 6.0 5.5 8.2 7.5 12.6 8.1 10.8 7.2 13.1 11.3 15.6
```

are readings from an experiment in which a best-fitting straight line is to be found :

```
y%.x
1.786
```

says that $y = 1.786x$ is the “best-fitting” line (in the least squares sense) of the form $y = kx$. It is usually more useful to fit a constant as well, that is either $y = kx + c$ or $x = ky + c$. A variant of *append* called *append items* adds 1 to every item in a list :

```
y%.1, .x
1.272 1.563
x%.1, .y
0.8117 0.4624
```

so $y = 1.272 + 1.563x$ and $x = 0.812 + 0.462y$ are the two regression lines.

The right argument of % . in the first of these two phrases is equivalent to $x^{/i.2}$ since x^0 is 1 for all values of x . Thus :

```
y % . x^/i.2      NB. Best fitting straight line
1.27181 1.56334
```

This idea can be immediately extended by using powers 0,1,2 in order to give the best-fitting quadratic :

```
y % . x^/i.3      NB. Best fitting quadratic
2.768 0.8803 0.06781
```

Now suppose data are the values of a polynomial, say the cubic function $0.5x^3 + 4x^2 + 5x - 6$. First assign a list of the coefficients in reverse order :

```
coef=._6 5 4 0.5
```

Next supply a set of x values for which you wish to evaluate the polynomial :

```
+x=._4+i.9
_5 _4 _3 _2 _1 0 1 2 3
```

Now use a verb p . which evaluates the polynomial at all values of x.:

```
+y=.coef p. x
38 15 0 _7 _6 3 20 45 78
```

An x,y table for the polynomial is :

```
x, :y
_4 _3 _2 _1 0 1 2 3 4
6 1.5 _4 _7.5 _6 3.5 24 58.5 110
```

The best fitting straight line and quadratic are

```
y % . x^/i.2
20.667 10.9
y % . x^/i.3
_6 10.9 4
```

that is $y = 10.9x + 20.667$ and $y = 4x^2 + 10.9x - 6$, whereas the best fitting cubic recovers the original coefficient values :

```
y % . x^/i.4
_6 5 4 0.5
```

SECTION 5 : BOXES AND RECORDS

The concept of a “box” greatly enhances the variety of data structures that J can handle. The principle is that any list structure, however complex, can be cast into a container called a box, which is itself a scalar. The analogy with records in conventional data processing should be clear, as should the application of the verb `box <` in the following example :

```
fname=. 'Harry'
sname=. 'Potter'
fpubl=.1998

shelfcode=. 'childrens'
(<fname) , (<sname) , (<fpubl) , <shelfcode
+-----+-----+-----+-----+
|Harry|Potter|1998|childrens|
+-----+-----+-----+-----+
```

Without boxing it is not possible to mix character and numeric data :

```
fname, sname, fpubl
|domain error
|  fname, sname      , fpubl
```

Also, boxes are complex scalar objects which are quite different both in character and in permissible operations from their contents. For example, if `m=.i.3 4` :

```
(<m) , <m
+-----+-----+
|0 1  2  3|0 1  2  3|
|4 5  6  7|4 5  6  7|
|8 9 10 11|8 9 10 11|
+-----+-----+
```

The result is a list of two items in a row, each of which is a container for three lists each of four items. Adding the contents of the boxed lists works fine :

```
m+m
0  2  4  6
8 10 12 14
16 18 20 22
```

but if you try adding the containers there is a problem :

```
(<m) +<m
|domain error
|  (<m)      +<m
```

This error message says clearly that adding in the numerical sense is not appropriate for the container domain, and in the same way you may not mix the numeric and character domains :

```

      fname + fpubl
|domain error
|   fname   +fpubl

```

An alternative to boxing items individually to construct a boxed list is to use the verb *link* ; :

```

      fname; sname; fpubl; shelfcode
+-----+-----+-----+-----+
|Harry|Potter|1998|childrens|
+-----+-----+-----+-----+

```

A matching verb *open* undoes boxed structures, but naturally requires type consistency for the unboxed contents :

```

      >fname; sname; shelfcode
Harry
Potter
childrens

```

that is, the result of the above open is three character lists.

A commercial type data set could be built up with further levels of boxing, for example :

```

      rec1=.fname; sname; fpubl; shelfcode
      rec2=. 'Jane'; 'Eyre'; 1847; 'classics'

      (<rec1), :<rec2
+-----+-----+-----+-----+
|+-----+-----+-----+-----+|
||Harry|Potter|1998|childrens||
|+-----+-----+-----+-----+|
+-----+-----+-----+-----+
|+-----+-----+-----+-----+ |
||Jane|Eyre|1847|classics| |
|+-----+-----+-----+-----+ |
+-----+-----+-----+-----+

```

using *lamine* , : to align the records vertically. (n.b. (<rec1), <rec2 would be a list of two single items, and so, as with say 2 3, the items would be displayed side by side.) Subsequent additions to the data set are made with simple *appends* :

```

      rec3=. 'Peter'; 'Rabbit'; 1904; 'childrens'

      ds=. (rec1, :rec2), rec3

```

```

ds
+-----+-----+-----+-----+
|Harry|Potter|1998|childrens|
+-----+-----+-----+-----+
|Jane |Eyre  |1847|classics |
+-----+-----+-----+-----+
|Peter|Pan   |1904|childrens|
+-----+-----+-----+-----+

```

This is a list of three four-item lists, and so

```

#(rec1, :rec2), rec3
3
$(rec1, :rec2), rec3
3 4

```

To extract a list of shelfcodes, use *transpose* and then *from* :

```

scs=.3{|:ds
scs
+-----+-----+-----+
|childrens|classics|childrens|
+-----+-----+-----+

```

This list can now be used to select the rows which correspond to children's books :

```

(scs=<'childrens')#ds
+-----+-----+-----+-----+
|Harry|Potter|1998|childrens|
+-----+-----+-----+-----+
|Peter|Pan   |1904|childrens|
+-----+-----+-----+-----+

```

To sort the records in alphabetical order of surnames use `1` to extract the `surnames` fields, then *grade-up* (introduced in section 3) to find the required permutation :

```

/:1{|:ds
1 0 2

```

Then apply this to make the required selection from the list of records :

```

(/:1{|:ds){ds
+-----+-----+-----+-----+
|Jane |Eyre  |1847|classics |
+-----+-----+-----+-----+
|Peter|Pan   |1904|childrens|
+-----+-----+-----+-----+
|Harry|Potter|1998|childrens|
+-----+-----+-----+-----+

```

SECTION 6 : INVESTIGATING POSSIBILITIES

This section begins with a problem. Suppose we have a situation with four two-way choices, for each of which something must be accepted or rejected. These could be to add (or not add) onions, cheese, sausage or mushrooms to a pizza. The aim is to make a table for the cost for each of the possible selections.

First two relevant verbs *base # .* and *antibase # : .* are introduced which allow numbers to be represented in all manner of different units. For example, converting 4 hrs. 22 mins. 54 secs. to seconds, and then 2 yds. 2 feet 9 inches to inches is achieved by :

```
0 60 60 #. 4 22 54 NB. base for mins., secs., is 60 60
15774
0 3 12 #. 2 2 9 NB. base for ft., ins., is 3 12
105
```

To convert back to the original units use # : :

```
0 60 60 #:15774 NB. secs to hrs. mins. secs.
4 22 54
0 3 12 #:105 NB. inches to yds. ft. ins.
2 2 9
```

As another example, *antibase* generates the individual digits in a decimal integer :

```
10 10 10 #:658 NB. digits in base 10
6 5 8
```

When the right argument is a list, the result is a list of lists :

```
10 10 10 10 10#:4342 8958 4646 243 10 12334
0 4 3 4 2
0 8 9 5 8
0 4 6 4 6
0 0 2 4 3
0 0 0 1 0
1 2 3 3 4
```

With no left argument the default number base is 2, which delivers binary numbers :

```
#:i.8 NB. 8=*/2 2 2, so it is also #:i.* /2 2 2
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

For the purposes of display, it is often convenient to use a verb *transpose* |: which reorganises a list of lists into a new set of lists in which the first list is a list of all first items, the second list is a list of all second items and so on :

```
|:#:i.8
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
```

Thus, returning to the pizza problem, given 0 means absent and 1 means present

```
|:#:i.16
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

represents all the possible choices of toppings.

There is no requirement that all the items in the left argument of dyadic #: are the same. If we wished to allow three possibilities for the second row, say because two types of cheese become available making a total of 24 possible pizzas in total, we can supply 2 3 2 2 as the left argument :

```
|:2 3 2 2#:i.24
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 1 1 1 1 2 2 2 2
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

Now the 24 is just the product of 2 3 2 and 2, or in J terminology */2 3 2 2, and so there is some redundancy in the above expression. J allows us to avoid this by using a hook with #: as the left verb, and i.@(*) as the right verb :

```
poss=.:#:i.@(*)
|: poss 2 3 2 2
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 1 1 1 1 2 2 2 2
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

Now make and display a list of the ingredients, using *open* and *link* described in section 5 :

```
]tops=.>'onions';'cheese';'sausage';'mushrooms '
onions
cheese
sausage
mushrooms
```

The leftmost verb *right*], by virtue of its definition, causes a display of everything to its right. Earlier monadic + and , were used for this purpose; in practice *right* is much the commonest way of doing simultaneous assignment and display because it is independent of type (character or numeric). Now define

```
tab=. |:poss 2 3 2 2
tops,tab
|domain error
| tops ,tab
```

The problem here is that J does not allow mixed types (character and numeric) to be appended. However, J does provide a very convenient verb *format* " : which transforms any numeric object into its character equivalent which looks identical on display :

```
tops, " :tab
onions
cheese
sausage
mushrooms
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 1 1 1 1 2 2 2 2
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

Not quite what we were thinking of - what is needed is *append items* , . which was first met in section 4, and is now used to append the two lists on an item by item basis as opposed to one list after the other. At the same time we have incorporated the dyadic form of format which allows a field size to be specified, in this case 1 :

```
tops,.1 " :tab
onions 00000000000001111111111111
cheese 000011112222000011112222
sausage 001100110011001100110011
mushrooms 010101010101010101010101
```

Now suppose that we want to compute the costs of the various types of pizza, given the costs of the various toppings :

```
cost=.0.80 1.80 2.20 1.50
```

The required calculation is to multiply each of the four lists (rows) in *tab* by the matching element in *cost* and then add “down the columns”, that is a +/ applied to each column. This combination of addition and multiplication, sometimes known as an “inner product” is expressed by the *dot* conjunction, and so

```
cost +/ .* tab
0 1.5 2.2 3.7 1.8 3.3 4 5.5 3.6 5.1 5.8 7.3 0.8 2.3 3 4.5
2.6 4.1 4.8 6.3 4.4 5.9 6.6 8.1
```

gives a list of the 24 possible pizzas.

To display the combinations in the form of a price list use *format* once again, only now applying it dyadically to specify a field width of 1 with no decimals for the first four columns, and 6 with 2 decimal places for the price column :

```
((4#1j0),6j2)":|:tab,cost +/ .* tab
0000 0.00
0001 1.50
0010 2.20
0011 3.70
0100 1.80
0101 3.30
0110 4.00
0111 5.50
.. ..
```

and so on. The leftmost column is not very informative, so use *copy* to make it more meaningful, having first used *append items* monadically to change it from a list of 24 items to a list of 24 single-item lists :

```
$prices=.cost+/. *tab
24
$, .prices
24 1
((|:tab) #'OCSM'),.6j2":,.prices
0.00
M 1.50
S 2.20
SM 3.70
C 1.80
CM 3.30
CS 4.00
CSM 5.50
CC 3.60
CCM 5.10
CCS 5.80
CCSM 7.30
O 0.80
OM 2.30
OS 3.00
OSM 4.50
OC 2.60
OCM 4.10
OCS 4.80
OCSM 6.30
OCC 4.40
OCCM 5.90
OCCS 6.60
OCCSM 8.10
```

SECTION 7 : EDITING, SYSTEM FACILITIES and SCRIPTS

Sooner or later you will want to write “programs” which you cannot express on a single line. While J contains many sophisticated features which make it possible to express a great deal in a single phrase, it can be comforting to know that a more conventional style of conditional and looping programming is also available. We look at three ways in which a user-defined verb for Fibonacci numbers can be constructed. Fibonacci numbers are sequences which are started with two arbitrary numbers (most commonly 0 and 1), and thereafter each succeeding number is the sum of the previous two. Clearly such a series could go on indefinitely, and so for practical purposes one of the parameters of a Fibonacci verb must provide a stopping condition, for example, either a total number of numbers is given, or the series may stop after a given value has been exceeded.

The English of the previous sentence can be rendered in J as `_2&{.` (take the last two) and `then +/@(_2&{.)` (sum the last two). Finally we want to join this to what we started with, which is the verb structure previously recognised as a hook `, +/@(_2&{.)`. Hence a user-defined verb for a single Fibonacci step is

```
fib=., +/@(_2&{.)
fib 2 3
2 3 5
```

Suppose that the stopping condition is that this step has to be performed a given number of times, say 10. J provides a conjunction `power ^:` which allows us to say just this. Its symbol demonstrates the analogy with the power verb which prescribes how often a number is to be multiplied by itself.

```
fib^:10(0 1)
0 1 1 2 3 5 8 13 21 34 55 89
```

We can even write this series all in a single line without any named verb, although the expression could be criticised as becoming a little bit hard to disentangle :

```
(, +/@(_2&{.))^:10(0 1)
0 1 1 2 3 5 8 13 21 34 55 89
```

Next, here, side by side, are two alternative ways in which we could have written and executed this program, the second shows incidentally that J supports recursion :

```
Fib=.dyad define
r=.y. [ i=.0
while. i<x. do.
  i=.i+1
  r=.r, +/_2{.r
end.
)

Fib=.dyad define
if.x.>0 do.
  r=. (x.-1)Fib y.
  r=.r, +/_2{.r
else. r=.y.
end.
)
```

```
10 Fib 0 1
0 1 1 2 3 5 8 13 21 34 55 89
```

In the body of the code `x.` and `y.` are used to reference the left and right arguments as in Section 3. and the control words (`while.` `do.` `end.` `if.` `else.`) must be terminated with dots. The code itself is terminated with a right parenthesis at which point the session reverts to execution mode, as opposed to object definition mode. The first code line in the left hand program could have been written as two separate lines – the left bracket acts as a statement separator, although you might have recognised it as the verb *left* whose value is what lies to its left, ignoring any right argument. Its role as a statement separator is a happy consequence of this definition.

The opening line of the above definition could also have been written `Fib=.4 :0`, in which case `Fib` would be a strictly dyadic verb. Another possibility which allows both a monadic and a dyadic definition is :

```
Fib=.3 : 0
10 Fib y.
:
r=.y. [ i=.0
.. etc.
```

The first line states that it is a dyadic verb (that is an object of class 3) which is to be defined. The 0 means that the subsequent input lines are to be made from the keyboard. The colon on the second line separates the monadic and dyadic definitions, and in the present case establishes a default left argument of 10.

More System Facilities

You have already seen how the foreign conjunction is used to get and set print precision. With different integer arguments it has many other uses for bridging the gap between programs and the underlying operating system. For example, a left argument of 1 is associated with reading and writing. `1!:1(1)` means “read from the keyboard” :

```
g=.1!:1(1)
I am now typing a message ...      NB. On the keyboard
g
I am now typing a message ...      NB. From the computer
```

Thus

```
ask=.monad define
1!:1(1)
)
h=.ask 'what''s the score?'
round about 20                      NB. From the keyboard
h
round about 20                      NB. From the computer
```

You can ask how many verbs there are presently in the workspace by

```

4! : 1 (3)
+---+---+---+
|Fib|ask|fib|
+---+---+---+

```

For nouns (that is constants and variables), adverbs and conjunctions replace 3 above with 0, 1 and 2 respectively.

You can ask the time of day with `6! : 0 (1)`, the elapsed time since start of session in microseconds by `6! : 1 (1)`, together with much, much more which is fully documented within the J system help facilities.

Scripts

Naturally you do not want to repeat the typing of input every time you want to use a sequence of user-defined verbs, so J provides for scripts, which are text files, (usually with qualifier `.ijs` although any qualifier except `.ijx` is acceptable) into which you can save objects for later use. The extension `.ijx` is reserved for executable J sessions which run under the control of software known as the “session manager”. It is this software which, for example, allows you to run an arrow up the screen and re-execute a previously submitted line. Assume that you have saved work from executing sessions into a script files. When you want to reuse the script file you can either

(a) load the file explicitly by `load'c:\j406\temp\fib.ijs'`

(b) Use **File/Open** to open the script file, which results in the file appearing in a new window. Once the script has been opened, use **Run/Window** or **Run/Line** from a drop-down menu;

(c) with the script file open, press `Ctrl-W` which is equivalent to (a).

You will find that there are already many existing scripts in your J system, and by loading these you are able to take advantage of a great deal of other people’s past work and experience. An example is the statistics package which consists of three separate scripts obtainable by **Open/System/stats.ijs**. This script contains three lines

```

script_z_ <'system\packages\stats\random.ijs'
script_z_ <'system\packages\stats\statfns.ijs'
script_z_ <'system\packages\stats\statdist.ijs'

```

Do **Run/Line** on the second of these and you can now execute all the verbs in the script, for example `median` which we laboured to program in section 3 is just one of several statistics immediately available :

```

median 6 7 9 0 _2 1 5 7
5.5

```

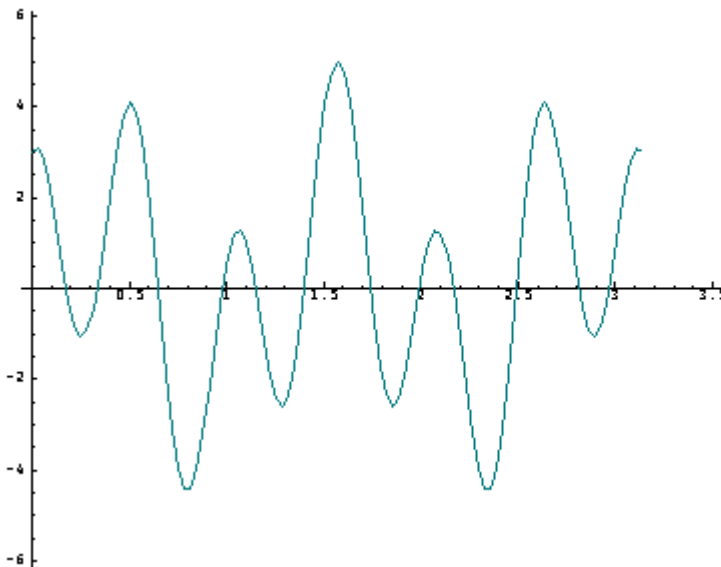
SECTION 8 : DRAWING GRAPHS

One feature of the J package which you are bound to welcome is the ease with which you can draw graphs. First do the following

```
load 'plot'  
plot c=.3 1 4 0.5 _2
```

and you should observe a new window containing a plot of these five values plotted against $i.5$ on the x axis. Any one-dimensional sequence can be plotted in this way; you will find that there are verbs in another script which help you draw, for example, trig series. To draw $y = 2\sin(5x) + 3\cos(12x)$ from 0 to 2π :

```
load 'numeric trig'  
x=.steps 0,(o.1),100      NB. o. means 'pi times'  
cos=.2&o. [ sin=.1&o.     NB. dyadic o. supplies the  
                          NB. trig ratios sin,cos,etc  
plot x;(2*sin 5*x)+3*cos(12*x)
```



To print this graph and simultaneously save it as a file, create the following verb :

```
pg=.dyad define  
'res fn'=.x.;y.      NB. resolution / filename  
require'opengl'  
gfile fn  
glsavebmp res  
printbmp_jzopengl_ fn  
)
```

Then, with the graph displayed in the plot window, issue the following :

```
500 300 pg 'c:\j406\temp\plot1.bmp'
```

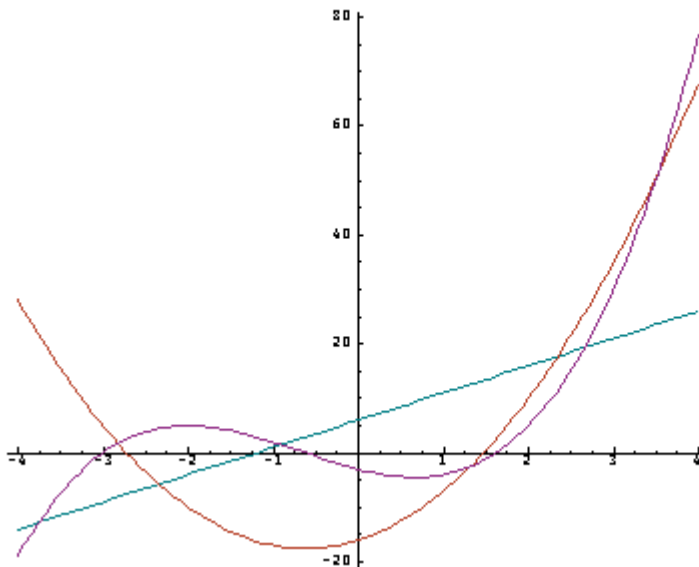
Graphs can be divided into two categories, those which are essentially algebraic, and those which are inherently geometrical. One example of each is given.

To start with, the polynomial $y = 4x^2 + 5x - 16$ (parabola) is plotted from -4 to 4 by

```
x=.steps _4 4 100
plot x;_16 5 4 p.x
```

In order to make the graph a little more interesting the second item in the link is changed into a list of three lists, corresponding to a straight line, a parabola and a cubic respectively :

```
plot x;((6 5 p.x),:_16 5 4 p.x),_3 _4 2 1 p.x
```



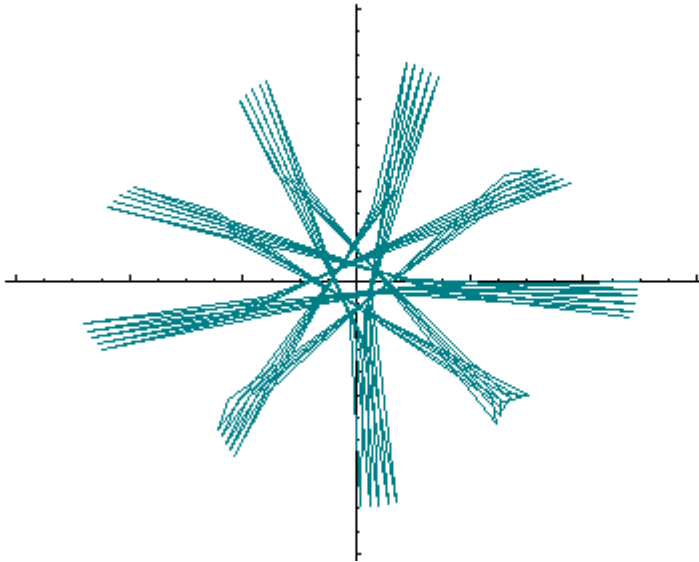
The example to illustrate geometric graphs involves one possible parametrisation of the curves known as epicycloids and hypocycloids. The first line below illustrates incidentally a technique for making multiple assignments on a single line :

```
'r R a b'=.2.6 2.36 _40.1 50
x=(.01*.2)*i.101      NB. x from 0 to 2pi
p=(r*cos(a*x))+R*cos(b*x)
q=(r*sin(a*x))+R*sin(b*x)
```

The command to make the plot is

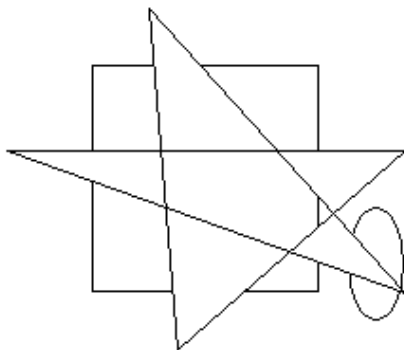
```
'labels 0'plot p;q
```

The left argument of `plot` is a character string which describes just one of the many possible drawing options, in this case it says “do not display axis labels”.



Facilities are also provided which allow the construction of a wide range of drawings. A selection of the basic drawing tools is given below. The graphics window is assumed to be 2 units high and 2 units deep with the origin (0,0) at the centre.

```
load 'graph'
gdopen''
gdirect _0.4 _0.4 0.8 0.8      NB. x y width height
gdellipse 0.5 _0.5 0.2 0.4    NB. centre, axis lens.
gdpolygon t=.0.1*7 1 _7 1 7 _4 _2 6 _1 _6 7 1
                                NB. t is six coordinate pairs in succession
gdshow''      NB. window displayed at this point
```



The scope for drawing graphics and plots is endless, and ample guidance to all the facilities can be found in the Graph Utilities Lab which is part of the J system.

SECTION 9 : RANK

If you have successfully followed the preceding sections, you are now ready to encounter what is one of the most important concepts in J, namely rank. In section 1 an object called `matrix` was defined which, although it looked like a matrix, was in fact a list of three four-lists.

```
matrix
3 1 4 0.5
_2 3 1 4
0.5 _2 3 1
```

The fact that it is a list of three lists is made explicit by *tally* :

```
#matrix
3
```

and the fact that each of these three lists is a four-list by *shape* :

```
$matrix
3 4
```

Tallying the shape

```
#$matrix
2
```

indicates that `matrix` is a list of lists, that is after penetrating two list levels primitive objects, in this case numbers, are reached. This can be expressed more concisely by saying that “`matrix` is an object of rank 2”, and leads to the definition of a verb

```
rank=.#@$
```

It is very important to get the order of verbs right in `rank` since `$$@#` means *tally* first which always results in a scalar, whose *shape* is an empty list. An alternative and equivalent definition of `rank` uses *cap* which was encountered earlier in section 3 :

```
rank=.[:#$
```

The structure involved here is called a capped fork, although a capped hook might be a more appropriate name because the effect is to make the verb pair `u v` return `u v(y)` rather than `y u v(y)` which it would do under the hook rule. In either case :

```
rank matrix
2
rank c=.3 1 4 0.5 _2
1
rank 7
0
```

What makes rank a particularly important concept is that verbs, both user-defined and primitive, can be made to operate at different rank levels. For example we have seen how *box <* can make any object into a scalar. Thus :

```
rank <matrix
0
```

However, if *box* operates at rank 1, that is 1 level in, then it is only the lowest level rank 1 objects which are boxed :

```
<"1 matrix
+-----+-----+-----+
|3 1 4 0.5|_2 3 1 4|0.5 _2 3 1|
+-----+-----+-----+
rank <"1 matrix
1
```

Because it joins a verb and a noun, the symbol " is a conjunction, and because of what it does it is called the *rank conjunction*. When we come to *insert* verbs like *+/* rank becomes important and useful in enabling different types of subtotalling to be performed. To “add-insert” two lists without any reference to rank, means, in a reasonably obvious sense, add their corresponding items :

```
+/matrix
1.5 2 8 5.5
```

that is, in matrix terms, add “down the columns”. To “add-insert” at rank 1 means to add at the lowest list level, that is to add the items in each of the boxes above :

```
+"/"1 m
8.5 6 2.5
```

in matrix terms, this is adding “along the rows” :

To find the grand total of all the items in matrix we have two options using *atop* :

```
gt1=.+/@(+/) or gt2=.+/@, NB. sum atop ravel
gt1 matrix gt2 matrix
17 17
```

and two using *cap* :

```
gt3=.[:+// gt4=.[:+/, NB. sum after ravel
gt3 matrix gt4 matrix
17 17
```

Which verb you choose is entirely a matter of personal taste. If using the *atop* form, notice that parentheses are essential in the definition of *gt1* . Without them *+/@+ /*

means “insert the verb +/@+” which, following the discussion in section 3, is just the same as inserting the verb + between each of the three top level lists, which in turn, as we saw above, is the same as adding down the columns :

```
(+/@+)/ matrix
1.5 2 8 5.5
```

To attach column totals to a matrix use a hook :

```
ct=.,+/
ct matrix
3 1 4 0.5
_2 3 1 4
0.5 _2 3 1
1.5 2 8 5.5
```

and to attach both row and column totals :

```
ct ct"1 matrix
3 1 4 0.5 8.5
_2 3 1 4 6
0.5 _2 3 1 2.5
1.5 2 8 5.5 17
```

To make this into a user-defined verb there are, as with `gt` , two options, you may use either a conjunction with parentheses or *cap* :

<code>totals1=.ct@:(ct"1)</code>	or	<code>totals2=.[:ct ct"1</code>
<code>totals1 m</code>		<code>totals2 m</code>
3 1 4 0.5 8.5		3 1 4 0.5 8.5
_2 3 1 4 6		_2 3 1 4 6
0.5 _2 3 1 2.5		0.5 _2 3 1 2.5
1.5 2 8 5.5 17		1.5 2 8 5.5 17

In `totals1` note again the importance of both the parentheses and the need for *at* `@:` as opposed to *atop* `@.` *at* is vital in this case since the grand total is itself the sum of totals, namely the row totals, and so it is necessary that column totalling does not commence until the row totalling and *appending* is complete, a form of sequencing which is implicit in *cap*.

Another example of a verb operating at different rank levels is taken from section 3 and involves the verb *reverse* :

```
m=.3 5$'rosessmellsweet'
w
roses
smell
sweet
```

```

      |.w                |."1 w
sweet                sesor
smell                llems
roses                teews

```

Similarly

```

      sortu m
      _2 3 1 4
0.5 _2 3 1
      3 1 4 0.5

```

sorts the rows in ascending order of the size of the first item, whereas

```

      sortu"1 m
0.5 1 3 4
      _2 1 3 4
      _2 0.5 1 3

```

sorts each rows into ascending order individually.

Here are the effects of rank with a deeper (that is, rank 3) list :

```

      i.2 2 3                +/"1 i.2 2 3
0 1 2                        3 12
3 4 5                        21 30
                                +/"2 i.2 2 3
6 7 8                          3 5 7
9 10 11                         15 17 19

```

`i.2 2 3` is a two-list, each item of which is a two-list of three lists. Summing at rank 1, the result is the sums of each of the inner two-lists of which there are four altogether. Summing at rank 2, the result is the sums of each of the two pairs of three-lists. Without the rank conjunction the result is the sum of the two outer two-lists item by item, exactly as with `matrix`:

```

      +/i.2 2 3
6 8 10
12 14 16

```

Armed with the rank conjunction the potential of J for expressing data manipulations is truly formidable.

Here the present investigation of J must cease. As hinted in the introduction there is much, much more left for you to discover. If you have found what you have met so far has stimulated you, then you will now proceed at a very fast rate under your own direction.

Index

- Absolute value, 8
- Adverb, 9, 21, 35
- Agenda, 23
- Alphabet, 11
- Alternating sum, 9
- Anti-base, 29
- Append, 7, 27, 29, 41
- Append items, 24, 31
- Arguments, 6
- Arithmetic functions, 2
- ASCII characters, 11
- Assignment, 3
- At, 17, 41
- Atop, 17, 40

- Base, 29
- Best fitting
 - polynomials, 25
- Bond, 19
- Box, 26, 40

- Cap, 21, 39, 41
- Capped fork, 39
- Cards, 15
- Case statement, 23
- Ceiling, 7, 23
- Character list, 5, 12
- Combinations, 15
- Comments, 4
- Complex numbers, 4
- Concurrency, 17
- Conjunctions, 17, 35
- Control words, 34
- Copy, 6, 9

- Deal, 13
- Decrement, 13
- Dice, 13
- Domain errors, 14, 26, 31
- Dot conjunction, 31
- Drawing, 36
- Drop, 12, 19
- Dyadic, 6

- Empty list, 10
- Epicycloids, 37
- Explicit Programming, 20
- Factorial, 15
- Fibonacci numbers, 33
- Fill items, 12
- Floor, 7, 23
- Foreign conjunction, 3, 20, 34

- Fork, 16, 19
- Format, 31
- From, 5, 13, 23
- Gerund, 23
- Grade down, 21
- Grade up, 21, 28
- Graphs, 36
- Grid, 11, 18

- Halve, 23
- Hook, 16, 19
- Hypocycloids, 37

- Identity values, 10
- Increment, 13
- Index generator, 10, 11, 29, 41
- Index of, 12
- Inner product, 31
- Insert, 9, 40
- Keyboard input, 34

- Laminate, 15, 24, 27
- Least squares, 24
- Left, 20, 34
- Link, 27
- Lists, 4
- Logarithms, 4
- Logical verbs, 10

- Matrix, 6, 24, 39
- Matrix inversion, 24
- Maximum, 7
- Mean, 16
- Median, 23, 35
- Membership, 9
- Minimum, 7
- Monadic, 6
- Multiple choice, 14

- Negate, 7
- Not equal, 23
- Noun, 11, 33
- Nub, 22

- Open, 27
- Or, 23
- Out of, 15
- Overtaking, 12

- Pascal's Triangle, 15
- Permutations, 21, 28
- Pi, 36
- Polynomials, 25, 37

- Power, 2, 33
- Primitives, 16
- Print precision, 3
- Quadratics, 25

- Range, 16
- Rank, 39
- Ravel, 15
- Reciprocal, 7
- Recursion, 33
- Reflex, 21
- Regression lines, 25
- Relational verbs, 9
- Remove blanks, 22
- Reshape, 6
- Residue, 8
- Reverse, 22, 41
- Right, 20, 31
- Roll, 13
- Rounding, 19

- Scalars, 3
- Scripts, 35
- Sequential
 - programming, 17
- Session Manager, 35
- Shape of, 6, 39
- Shift, 22
- Signum, 7
- Sorting, 21, 41
- Simultaneous
 - equations, 24
- Square Root, 4
- Square, 18
- System facilities, 3, 20, 34
- Tables, 11, 15
- Tacit programming, 16
- Tail, 12
- Take, 12
- Tally, 5, 39
- Tie, 23
- Time, 35
- Tossing coins, 13
- Transpose, 30
- Trigonometry, 36

- Underbar, 3

- Vectors, 4
- Verbs, 2

- Workspace, 3
- Wrap around, 6

Vocabulary

The table below gives the J symbols which are used in this booklet. A few other symbols have also been included there are obvious analogies with those which are used in the booklet. Where two meanings are given, these are monadic/dyadic.

Verbs:

+	<i>conjugate/plus</i>	+.	<i>or (dyadic)</i>	+:	<i>double/not-or</i>
-	<i>negate/minus</i>	-.	<i>not (monadic)</i>	-:	<i>halve</i>
*	<i>signum/times</i>	*.	<i>and (dyadic)</i>	*:	<i>square/not-and</i>
%	<i>reciprocal/divide</i>	%.	<i>matrix inverse (monadic)</i>	%:	<i>square root</i>
^	<i>power</i>	^.	<i>natural logarithm/logarithm</i>		
!	<i>factorial/out of</i>				
	<i>absolute value/residue</i>	.	<i>reverse/shift</i>	:	<i>transpose</i>
=	<i>equals</i>				
?	<i>roll/deal</i>				
#	<i>tally/copy</i>	#.	<i>base</i>	#:	<i>antibase</i>
\$	<i>shape of/reshape</i>				
{	<i>from</i>	{.	<i>take</i>	{:	<i>drop</i>
,	<i>ravel/append</i>	,.	<i>append items</i>	,:	<i>laminare</i>
<	<i>box/less than</i>	<.	<i>floor/minimum</i>	<:	<i>decrement/less than or equal</i>
>	<i>open/greater than</i>	>.	<i>ceiling/maximum</i>	>:	<i>increment/greater than or equal</i>
				~:	<i>not equals</i>
;	<i>link</i>				
[<i>left</i>			[:	<i>cap</i>
]	<i>right</i>				
				/:	<i>grade-up</i>
				\:	<i>grade-down</i>
				":	<i>format</i>
		e.	<i>membership</i>		
		i.	<i>index generator/index of</i>		
		o.	<i>multiply by pi</i>		
		p.	<i>polynomial</i>		

Adverbs :

/ *insert/table*
~ *reflex*

Conjunctions :

@	<i>atop (strong linkage)</i>	@.	<i>agenda</i>	@:	<i>after (weak linkage)</i>
&	<i>bond</i>				
.	<i>dot</i>				
`	<i>tie</i>			^:	<i>power</i>
				!:	<i>foreign conjunction</i>

Noun :

a. *alphabet*

Special symbol :

=. *assignment*

Control words : *if. do. else. while. end.*