

# Learning J

draft 8, August 2002, still not finished

---

## ABOUT THIS BOOK

This book is meant to help the reader to learn the computer-programming language J.

The book is intended to be read with enjoyment by both the beginning programmer and the experienced programmer alike. The only prerequisite is an interest on the part of the reader in learning a programming language.

The emphasis is on making the J language accessible to a wide readership. Care is taken to introduce only one new idea at a time, to provide examples at every step, and to make examples so simple that the point can be grasped immediately. Even so, the experienced programmer will find much to enjoy in the radical simplicity and power of the J notation.

The scope of this book is the core J language common to the many implementations of J available on different computers. The coverage of the core language is meant to be relatively complete, covering (eventually) most of the J Dictionary.

Hence the book does not cover topics such as graphics, plotting, GUI, and database access covered in the J User Guide. It should also be stated what the aims of the book are not: neither to teach principles of programming as such, nor to study algorithms, or topics in mathematics or other subjects using J as a vehicle, nor to provide definitive reference material.

The book is organised as follows. Part 1 is an elementary introduction which touches on a variety of themes. The aim is to provide the reader, by the end of part 1, with an overview and a general appreciation of the J language. The themes introduced in Part 1 are then developed in more depth and detail in the remainder

of the book.

---

## Feedback

Please send comments and criticisms to [rstokes@dial.pipex.com](mailto:rstokes@dial.pipex.com)

---

## Acknowledgements

I am grateful to readers of earlier drafts for encouragement and for valuable criticisms and suggestions.

---

These web pages are also available in a single [downloadable zip file](#). There is a version in PDF format on [Skip Cave's web page](#).

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice, including this provision, is also reproduced.

last updated 16 Aug 2002

# Contents

- 1 [Basics](#)
- 2 [Lists and Tables](#)
- 3 [Defining Functions](#)
- 4 [Scripts and Explicit Functions](#)
- 5 [Building Arrays](#)
- 6 [Indexing](#)
- 7 [Ranks](#)
- 8 [Composing Verbs](#)
- 9 [Trains of Verbs](#)
- 10 [Conditional and Other Forms](#)
- 11 [Tacit Verbs Concluded](#)
- 12 [Explicit Verbs](#)
- 13 [Explicit Operators](#)
- 14 [Gerunds](#)
- 15 [Tacit Operators](#)
- 16 [Rearrangements](#)
- 17 [Patterns of Application](#)
- 18 [Sets, Classes and Relations](#)
- 19 [Numbers](#)
- 20 [Scalar Numerical Functions](#)
- 21 [Factors and Polynomials](#)
- 22 [Vectors and Matrices](#)
- 23 [Names and Locales](#)
- 24 [OOP](#)
- 25 [Script Files](#)
- 26 [Evaluating Expressions](#)

# Chapter 1: Basics

These first four chapters make up Part 1 of this book. Part 1 is a quick tour, giving a first look at some of the main features of J. In later chapters we will take up again themes which are introduced in Part 1.

## 1.1 Interactive Use

The user types a line at the keyboard. This input line may be an expression, such as  $2+2$ . When the line is entered (by pressing the "enter" or "carriage return" key), the value of the expression is computed and displayed on the next line.

```
    2+2
4
```

The user is then prompted for another line of input. The prompt is seen by the cursor being positioned a few spaces from the left margin. Thus in this book, a line indented by a few spaces represents input typed by a user, and a following line, not indented, represents the corresponding output.

## 1.2 Arithmetic

The symbol for multiplication is \* (asterisk).

```
    2*3
6
```

If we try this again, this time typing 2 space \* space 3

```
    2 * 3
6
```

the result is the same as before, showing that the spaces here are optional. Spaces can make an expression more readable.

The symbol for division is % (percent).

$$\begin{array}{r} 3 \% 4 \\ 0.75 \end{array}$$

For subtraction, we have the familiar - symbol:

$$\begin{array}{r} 3 - 2 \\ 1 \end{array}$$

The next example shows how a negative number is represented. The negative sign is a leading \_ (underscore) symbol, with no space between the sign and the digits of the number. This sign is not an arithmetic function: it is part of the notation for writing numbers, in the same way that a decimal point is part of the notation.

$$\begin{array}{r} 2 - 3 \\ \_1 \end{array}$$

The symbol for negation is -, the same symbol as for subtraction:

$$\begin{array}{r} - 3 \\ \_3 \end{array}$$

The symbol for the power function is ^ (caret). 2 cubed is 8:

$$\begin{array}{r} 2 \wedge 3 \\ 8 \end{array}$$

The arithmetic function to compute the square of a number has the symbol \*: (asterisk colon).

$$\begin{array}{r} *: 4 \\ 16 \end{array}$$

## 1.3 Some Terminology: Function, Argument, Application, Value

Consider an expression such as  $2 * 3$ . We say that the multiplication function  $*$  is applied to its arguments. The left argument is 2 and the right argument is 3. Also, 2 and 3 are said to be the values of the arguments.

## 1.4 List Values

Sometimes we may wish to repeat the same computation several times for several different numbers. A list of numbers can be given as  $1\ 2\ 3\ 4$ , for example, written with a space between each number and the next. To find the square of each number in this list we could say:

```
*: 1 2 3 4
1 4 9 16
```

Here we see that the "Square" function ( $*:$ ) applies separately to each item in the list. If a function such as  $+$  is given two list arguments, the function applies separately to pairs of corresponding items:

```
1 2 3 + 10 20 30
11 22 33
```

If one argument is a list and the other a single item, the single item is replicated as needed:

```
1 + 10 20 30
11 21 31
```

```
1 2 3 + 10
11 12 13
```

Sometimes it is helpful, when we are looking at a new function, to see how a pattern in a list of arguments gives rise to a pattern in the list of results.

For example, when 7 is divided by 2 we can say that the quotient is 3 and the remainder is 1. A built-in J function to compute remainders is  $|$  (vertical bar), called the "Residue" function. Patterns in arguments and results are shown by:

```
2 | 0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
```

```

      3 | 0 1 2 3 4 5 6 7
0 1 2 0 1 2 0 1

```

The Residue function is like the familiar "mod" or "modulo" function, except that we write  $(2 \mid 7)$  rather than  $(7 \bmod 2)$

## 1.5 Parentheses

An expression can contain parentheses, with the usual meaning; what is inside parentheses is, in effect, a separate little computation.

```

      (2+1)*(2+2)
12

```

Parentheses are not always needed, however. Consider the J expression:  $3*2+1$ . Does it mean  $(3*2)+1$ , that is, 7, or does it mean  $3*(2+1)$  that is, 9?

```

      3 * 2 + 1
9

```

In school mathematics we learn a convention, or rule, for writing expressions: multiplication is to be done before addition. The point of this rule is that it reduces the number of parentheses we need to write.

There is in J no rule such as multiplication before addition. We can always write parentheses if we need to. However, there is, in J, a parenthesis-saving rule, as the example of  $3*2+1$  above shows. The rule, is that, in the absence of parentheses, the right argument of an arithmetic function is everything to the right. Thus in the case of  $3*2+1$ , the right argument of  $*$  is  $2+1$ . Here is another example:

```

      1 + 3 % 4
1.75

```

We can see that  $\%$  is applied before  $+$ , that is, the rightmost function is applied first.

This "rightmost first" rule is different from, but plays the same role as, the common convention of "multiplication before addition". It is merely a convenience: you can

ignore it and write parentheses instead. Its advantage is that there are, in J, many (something like 100) functions for computation with numbers and it would be out of the question to try to remember which function should be applied before which.

In this book, I will on occasion show you an expression having some parentheses which, by the "rightmost first" rule, would not be needed. The aim in doing this is to emphasize the structure of the expression, by setting off parts of it, so as to make it more readable.

## 1.6 Variables and Assignments

The English-language expression:

let x be 100

can be rendered in J as:

```
x =: 100
```

This expression, called an assignment, causes the value 100 to be assigned to the name x. We say that a variable called x is created and takes on the value 100.

When a line of input containing only an assignment is entered at the computer, then nothing is displayed in response (because you probably don't need to see again right away the value you just typed in.)

A name with an assigned value can be used wherever the value is wanted in following computations.

```
x - 1
99
```

The value in an assignment can itself be computed by an expression:

```
y =: x - 1
```

Thus the variable y is used to remember the results of the computation x-1. To see what value has been assigned to a variable, enter just the name of the variable. This is an expression like any other, of a particularly simple form:

```
y
```



Assignments can be made repeatedly to the same variable; the new value supersedes the current value:

```
z =: 6
z =: 8
z
8
```

The value of a variable can be used in computing a new value for the same variable:

```
z =: z + 1
z
9
```

It was said above that a value is not displayed when a line consisting of an assignment is entered. Nevertheless, an assignment is an expression: it does have a value which can take part in a larger expression.

```
1 + (u =: 99)
100
u
99
```

Here are some examples of assignments to show how we may choose names for variables:

```
x           =: 0
X           =: 1
K9          =: 2
finaltotal  =: 3
FinalTotal  =: 4
average_annual_rainfall =: 5
```

Each name must begin with a letter. It may contain only letters (upper-case or lower-case), numeric digits (0–9) or the underscore character (\_). Note that upper-case and lower-case letters are distinct; `x` and `X` are the names of distinct variables:

```

      x
0
      x
1

```

## 1.7 Terminology: Monads and Dyads

A function taking a single argument on the right is called a monadic function, or a monad for short. An example is "Square", (`* :` ). A function taking two arguments, one on the left and one on the right, is called a dyadic function or dyad. An example is `+`.

Subtraction and negation provide an example of the same symbol (`-`) denoting two different functions. In other words, we can say that `-` has a monadic case (negation) and a dyadic case (subtraction). Nearly all the built-in functions of J have both a monadic and a dyadic case. For another example, recall that the division function is `%`, or as we now say, the dyadic case of `%`. The monadic case of `%` is the reciprocal function.

```

      % 4
0.25

```

## 1.8 More Built-In Functions

The aim in this section is to convey a little of the flavor of programming in J by looking at a small further selection of the many built-in functions which J offers.

Consider the English-language expression: add together the numbers 2, 3, and 4, or more briefly:

add together 2 3 4

We expect a result of 9. This expression is rendered in J as:

```

+ / 2 3 4
9

```

Comparing the English and the J, "add" is conveyed by the `+` and "together" is conveyed by the `/`. Similarly, the expression:

multiply together 2 3 4

should give a result of 24. This expression is rendered in J as

$$\begin{array}{c} * \ / \ 2 \ 3 \ 4 \\ 24 \end{array}$$

We see that  $+ / 2 \ 3 \ 4$  means  $2+3+4$  and  $* / 2 \ 3 \ 4$  means  $2*3*4$ . The symbol  $/$  is called "insert", because in effect it inserts whatever function is on its left between each item of the list on its right. The general scheme is that if  $F$  is any dyadic function and  $L$  is a list of numbers  $a, b, c, \dots, y, z$  then:

$$F \ / \ L \quad \text{means} \quad a \ F \ b \ F \ \dots \ F \ y \ F \ z$$

Moving on to further functions, consider these three propositions:

2 is larger than 1 (which is clearly true)

2 is equal to 1 (which is false)

2 is less than 1 (which is false)

In J, "true" is represented by the number 1 and "false" by the number 0. The three propositions are rendered in J as:

$$\begin{array}{c} 2 \ > \ 1 \\ 1 \end{array}$$

$$\begin{array}{c} 2 \ = \ 1 \\ 0 \end{array}$$

$$\begin{array}{c} 2 \ < \ 1 \\ 0 \end{array}$$

If  $x$  is a list of numbers, for example:

$$x =: 5 \ 4 \ 1 \ 9$$

we can ask: which numbers in  $x$  are greater than 2?

```
      x > 2
1 1 0 1
```

Evidently, the first, second and last, as reported by the 1's in the result of  $x > 2$ . Is it the case that all numbers in  $x$  are greater than 2?

```
      * / x > 2
0
```

No, because we saw that  $x > 2$  is 1 1 0 1. The presence of any zero ("false") means the the multiplication (here  $1*1*0*1$ ) cannot produce 1.

How many items of  $x$  are greater than 2? We add together the 1's in  $x > 2$ :

```
      + / x > 2
3
```

How many numbers are there altogether in  $x$ ? We could add together the 1's in  $x=x$ .

```
      x
5 4 1 9
      x = x
1 1 1 1
      + / x = x
4
```

but there is a built-in function `#` (called "Tally") which gives the length of a list:

```
      # x
4
```

## 1.9 Side By Side Displays

When we are typing J expressions into the computer, expressions and results follow each other down the screen. Let me show you the last few lines again:

```
      x
5 4 1 9
```

```

      x = x
1 1 1 1
      +/ x = x
4
      # x
4

```

Now, sometimes in this book I would like to show you a few lines such as these, not one below the other but side by side across the page, like this:

x	x = x	+/ x = x	# x
5 4 1 9	1 1 1 1	4	4

This means: at this stage of the proceedings, if you type in the expression `x` you should see the response `5 4 1 9`. If you now type in `x = x` you should see `1 1 1 1`, and so on. Side-by-side displays are not a feature of the J system, but merely figures, or illustrations, in this book. They show expressions in the first row, and corresponding values below them in the second row.

When you type in an assignment (`x=:something`), the J system does not show the value. Nevertheless, an assignment is an expression and has a value. Now and again it might be helpful to see, or to be reminded of, the values of our assignments, so I will often show them in these side-by-side displays. To illustrate:

x =: 1 + 2 3 4	x = x	+/ x = x	# x
3 4 5	1 1 1	3	3

Returning now to the built-in functions, suppose we have a list. Then we can choose items from it by taking them in turn and saying "yes, yes, no, yes, no" for example. Our sequence of choices can be represented as `1 1 0 1 0`. Such a list of 0's and 1's is called a bit-string (or sometimes bit-list or bit-vector). The function which applies the choices is the dyadic case of `#` which can take a bit-string as left argument:

<code>y =: 6 7 8 9 10</code>	<code>1 1 0 1 0 # y</code>
<code>6 7 8 9 10</code>	<code>6 7 9</code>

We can select from `y` just those items which satisfy some condition, such as: those which are greater than 7

<code>y</code>	<code>y &gt; 7</code>	<code>(y &gt; 7) # y</code>
<code>6 7 8 9 10</code>	<code>0 0 1 1 1</code>	<code>8 9 10</code>

## 1.10 Comments

In a line of J, the symbol `NB.` (capital N, capital B dot) introduces a comment. Anything following `NB.` to the end of the line is not evaluated. For example

```
NB.    this is a whole line of annotation
```

```
6 + 6    NB. ought to produce 12
12
```

## 1.11 Naming Scheme for Built-In Functions

Each built-in function of J has an informal and a formal name. For example, the function with the formal name `+` has the informal name of "Plus". Further, we have seen that there may be monadic and dyadic cases, so that the formal name `-` corresponds to the informal names "Negate" and "Minus".

The informal names are, in effect, short standard descriptions, usually one word. They are not recognised by the J software, that is, expressions in J use always the formal names. In this book, the informal names will be quoted, thus: "Minus".

Nearly all the built-in functions of J have formal names with one character or two characters. Examples are the `*` and `*:` functions. The second character is always either `:` (colon) or `.` (dot, full stop, or period). A two-character name is meant to suggest some relationship to a basic one-character function. Thus "Square" (`*:`) is related to "Times" (`*`).

Hence the built-in J functions tend to come in families of up to 6 related functions. There are the monadic and dyadic cases, and for each case there are the basic, the colon and dot variants. This will be illustrated for the `>` family.

Dyadic `>` we have already met as "Larger Than".

Monadic `>` we will come back to later.

Monadic `>.` rounds its argument up to an integer. Note that rounding is always upwards as opposed to rounding to the nearest integer. Hence the name: "Ceiling"

```
>. _1.7 1 1.7
_1 1 2
```

Dyadic `>.` selects the larger of its two arguments

```
3 >. 1 3 5
3 3 5
```

We can find the largest number in a list by inserting "Larger Of" between the items, using `/`. For example, the largest number in the list `1 6 5` is found by evaluating `(>. / 1 6 5)`. The next few lines are meant to convince you that this should give 6. The comments show why each line should give the same result as the previous.

```
>. / 1 6 5
6
1 >. 6 >. 5      NB. by the meaning of /
6
1 >. (6 >. 5)    NB. by rightmost-first rule
6
1 >. (6)         NB. by the meaning of >.
6
1 >. 6           NB. by the meaning of ()
6
6               NB. by the meaning of >.
```

Monadic  $>:$  is informally called "Increment". It adds 1 to its argument:

```
>: _2 3 5 6.3
_1 4 6 7.3
```

Dyadic  $>:$  is "Larger or Equal"

```
3 >: 1 3 5
1 1 0
```

This is the end of Chapter 1.

---

Copyright © Roger Stokes 2001. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 22Sep01



# Chapter 2: Lists and Tables

Computations need data. So far we have seen data only as single numbers or lists of numbers. We can have other things by way of data, such as tables for example. Things like lists and tables are called "arrays".

## 2.1 Tables

A table with, say, 2 rows and 3 columns can be built with the `$` function:

```
table =: 2 3 $ 5 6 7 8 9 10
table
5 6 7
8 9 10
```

The scheme here is that the expression `(x $ y)` builds a table. The dimensions of the table are given by the list `x` which is of the form number-of-rows followed by number-of-columns. The elements of the table are supplied by the list `y`.

Items from `y` are taken in order, so as to fill the first row, then the second, and so on. The list `y` must contain at least one item. If there are too few items in `y` to fill the whole table, then `y` is re-used from the beginning.

2 4 \$ 5 6 7 8 9	2 2 \$ 1
5 6 7 8 9 5 6 7	1 1 1 1

The `$` function offers one way to build tables, but there are many more ways: see [Chapter 05 p5](#).

Functions can be applied to whole tables exactly as we saw earlier for lists:

table	10 * table	table + table
5 6 7 8 9 10	50 60 70 80 90 100	10 12 14 16 18 20

One argument can be a table and one a list:

table	0 1 * table
5 6 7 8 9 10	0 0 0 8 9 10

In this last example, evidently the items of the list 0 1 are automatically matched against the rows of the table, 0 matching the first row and 1 the second. Other patterns of matching the arguments against each other are also possible - see [Chapter 07 p7](#).

## 2.2 Arrays

A table is said to have two dimensions (namely, rows and columns) and in this sense a list can be said to have only one dimension.

We can have table-like data objects with more than two dimensions. The left argument of the \$ function can be a list of any number of dimensions. The word "array" is used as the general name for a data object with some number of dimensions. Here are some arrays with one, two and three dimensions:

3 \$ 1	2 3 \$ 5 6 7	2 2 3 \$ 5 6 7 8
1 1 1	5 6 7 5 6 7	5 6 7 8 5 6  7 8 5 6 7 8

---

The 3-dimensional array in the last example is said to have 2 planes, 2 rows and 3 columns and the two planes are displayed one below the other.

Recall that the monadic function # gives the length of a list.

# 6 7	# 6 7 8
2	3

The monadic function \$ gives the list-of-dimensions of its argument:

L =: 5 6 7	\$ L	T =: 2 3 \$ 1	\$ T
5 6 7	3	1 1 1 1 1 1	2 3

Hence, if x is an array, the expression (# \$ x) yields the length of the list-of-dimensions of x, that is, the dimension-count of x, which is 1 for a list, 2 for a table and so on.

L	\$ L	# \$ L	T	\$T	# \$ T
5 6 7	3	1	1 1 1 1 1 1	2 3	2

If we take x to be a single number, then the expression (# \$ x) gives zero.

```
# $ 17
0
```

We interpret this to mean that, while a table has two dimensions, and a list has one,

a single number has none, because its dimension-count is zero. A data object with a dimension-count of zero will be called a scalar. We said that "arrays" are data objects with some number of dimensions, and so scalars are also arrays, the number of dimensions being zero in this case.

We saw that `(# $ 17)` is 0. We can also conclude from this that, since a scalar has no dimensions, its list-of-dimensions (given here by `$ 17`) must be a zero-length, or empty, list. Now a list of length 2, say can be generated by an expression such as `2 $ 99` and so an empty list, of length zero, can be generated by `0 $ 99` (or indeed, `0 $ any number`)

The value of an empty list is displayed as nothing:

<code>2 \$ 99</code>	<code>0 \$ 99</code>	<code>\$ 17</code>
<code>99 99</code>		

Notice that a scalar, (`17` say), is not the same thing as a list of length one (e.g. `1 $ 17`), or a table with one row and one column (e.g. `1 1 $ 17`). The scalar has no dimensions, the list has one, the table has two, but all three look the same when displayed on the screen:

```
S =: 17
L =: 1 $ 17
T =: 1 1 $ 17
```

S	L	T	# \$ S	# \$ L	# \$ T
17	17	17	0	1	2

A table may have only one column, and yet still be a 2-dimensional table. Here `t` has 3 rows and 1 column.

<code>t =: 3 1 \$ 5 6 7</code>	<code>\$ t</code>	<code># \$ t</code>
--------------------------------	-------------------	---------------------

5	3 1	2
6		
7		

## 2.3 Terminology

The property we called "dimension-count" is in J called by the shorter name of "rank", so a single number is said to be a rank-0 array, a list of numbers a rank-1 array and so on. The list-of-dimensions of an array is called its "shape".

The mathematical terms "vector" and "matrix" correspond to what we have called "lists" and "tables" (of numbers). An array with 3 or more dimensions (or, as we now say, an array of rank 3 or higher) will be called a "report".

A summary of terms and functions for describing arrays is shown in the following table.

	Example	Shape	Rank
	x	\$ x	# \$ x
Scalar	6	empty list	0
List	4 5 6	3	1
Table	0 1 2 3 4 5	2 3	2
Report	0 1 2 3 4 5 6 7 8 9 10 11	2 2 3	3

This table above was in fact produced by a small J program, and is a genuine "table", of the kind we have just been discussing. Its shape is 6 4. However, it is evidently not just a table of numbers, since it contains words, list of numbers and

so on. We now look at arrays of things other than numbers.

## 2.4 Arrays of Characters

Characters are letters of the alphabet, punctuation, numeric digits and so on. We can have arrays of characters just as we have arrays of numbers. A list of characters is entered between single quotes, but is displayed without the quotes. For example:

```
title =: 'My Ten Years in a Quandary'
title
My Ten Years in a Quandary
```

A list of characters is called a character-string, or just a string. A single quote in a string is entered as two successive single quotes.

```
'What's new?'
What's new?
```

## 2.5 Some Functions for Arrays

At this point it will be useful to look first at some functions for dealing with arrays. J is very rich in such functions: here we look at a just a few.

### 2.5.1 Joining

The built-in function `,` (comma) is called "Append". It joins things together to make lists.

```
a =: 'rear'
b =: 'ranged'
a,b
rearranged
```

The "Append" function joins lists or single items.

<code>x =: 1 2 3</code>	<code>0 , x</code>	<code>x , 0</code>	<code>0 , 0</code>	<code>x , x</code>
-------------------------	--------------------	--------------------	--------------------	--------------------

1 2 3	0 1 2 3	1 2 3 0	0 0	1 2 3 1 2 3
-------	---------	---------	-----	-------------

The "Append" function can take two tables and join them together end-to-end to form a longer table:

T1=: 2 3 \$ 'catdog'	T2=: 2 3 \$ 'ratpig'	T1,T2
cat dog	rat pig	cat dog rat pig

Now a table can be regarded as a list where each item of the list is a row of the table. This is something we will find useful over and over again, so let me emphasize it: the items of a table are its rows.

With this in mind, we can say that in general  $(x, y)$  is a list consisting of the items of  $x$  followed by the items of  $y$ . For more information about "Append", see [Chapter 05 p5](#).

## 2.5.2 Selecting

Now we look at selecting items from a list. Positions in a list are numbered 0, 1, 2 and so on. The first item occupies position 0. To select an item by its position we use the { (left brace) function.

y =: 'abcd'	0 { y	1 { y	3 { y
abcd	a	b	d

A position-number is called an index. The { function can take as left argument a single index or a list of indices:

y	0 { y	0 1 { y	3 0 1 { y
abcd	a	ab	dab

There is a built-in function `i.` (letter-i dot). The expression `(i. n)` generates  $n$  successive integers from zero.

i. 4	i. 6	1 + i. 3
0 1 2 3	0 1 2 3 4 5	1 2 3

If  $x$  is a list, the expression `(i. # x)` generates all the possible indexes into the list  $x$ .

x =: 'park'	# x	i. # x
park	4	0 1 2 3

With a list argument, `i.` generates an array:

```
i. 2 3
0 1 2
3 4 5
```

There is a dyadic version of `i.`, called "Index Of". The expression `(x i. y)` finds the position, that is, index, of  $y$  in  $x$ .

```
'park' i. 'k'
3
```

The index found is that of the first occurrence of  $y$  in  $x$ .

```
'parka' i. 'a'
1
```



If `y` is not present in `x`, the index found is 1 greater than the last possible position.

```
'park' i. 'j'
```

4

For more about the many variations of indexing, see [Chapter 06 p6](#).

## 2.6 Arrays of Boxes

### 2.6.1 Linking

There is a built-in function `;` (semicolon, called "Link"). It links together its two arguments to form a list. The two arguments can be of different kinds. For example we can link together a character-string and a number.

```
A =: 'The answer is' ; 42
```

A

```
+-----+---+
|The answer is|42|
+-----+---+
```

The result `A` is a list of length 2, and is said to be a list of boxes. Inside the first box of `A` is the string `'The answer is'`. Inside the second box is the number 42. A box is shown on the screen by a rectangle drawn round the value contained in the box.

A	0 { A
+-----+---+  The answer is 42  +-----+---+	+-----+---+  The answer is  +-----+---+

A box is a scalar whatever kind of value is inside it. Hence boxes can be packed into regular arrays, just like numbers. Thus `A` is a list of scalars.

A	\$ A	s =: 1 { A	# \$ s
+-----+---+  The answer is 42  +-----+---+	2	+---+  42  +---+	0

The main purpose of an array of boxes is to assemble into a single variable several values of possibly different kinds. For example, a variable which records details of a purchase (date, amount, description) could be built as a list of boxes:

```
P =: 18 12 1998 ; 1.99 ; 'baked beans'
P
+-----+---+-----+
|18 12 1998|1.99|baked beans|
+-----+---+-----+
```

Note the difference between "Link" and "Append". While "Link" joins values of possibly different kinds, "Append" always joins values of the same kind. That is, the two arguments to "Append" must both be arrays of numbers, or both arrays of characters, or both arrays of boxes. Otherwise an error is signalled.

'answer is'; 42	'answer is' , 42
+-----+---+  answer is 42  +-----+---+	error

On occasion we may wish to combine a character-string with a number, for example to present the result of a computation together with some description. We could "Link" the description and the number, as we saw above. However a smoother presentation could be produced by converting the number to a string, and then Appending this string and the description, as characters.

Converting a number to a string can be done with the built-in "Format" function " : (double-quote colon). In the following example n is a single number, while s, the

formatted value of `n`, is a string of characters of length 2.

<code>n =: 42</code>	<code>s =: ": n</code>	<code># s</code>	<code>'answer is ' , s</code>
42	42	2	answer is 42

For more about "Format", see [Chapter 19 p19](#). Now we return to the subject of boxes.

Because boxes are shown with rectangles drawn round them, they lend themselves to presentation of results on-screen in a simple table-like form.

```
p =: 4 1 $ 1 2 3 4
q =: 4 1 $ 3 0 1 1

2 3 $ ' p ' ; ' q ' ; ' p+q ' ; p ; q ; p+q
+---+---+---+
| p | q | p+q |
+---+---+---+
| 1 | 3 | 4 |
| 2 | 0 | 2 |
| 3 | 1 | 4 |
| 4 | 1 | 5 |
+---+---+---+
```

## 2.6.2 Boxing and Unboxing

There is a built-in function `<` (left-angle-bracket, called "Box"). A single boxed value can be created by applying `<` to the value.

```
< 'baked beans'
+-----+
|baked beans|
+-----+
```

Although a box may contain a number, it is not itself a number. To perform computations on a value in a box, the box must be, so to speak "opened" and the value taken out. The function `>` (right-angle-bracket) is called "Open".

b =: < 1 2 3	> b
+-----+   1 2 3   +-----+	1 2 3

It may be helpful to picture < as a funnel. Flowing into the wide end we have data, and flowing out of the narrow end we have boxes which are scalars, that is, dimensionless or point-like. Conversely for > .

Since boxes are scalars, they can be strung together into lists of boxes with the comma function, but the semicolon function is more convenient because it combines the stringing-together and the boxing:

(< 1 1) , (< 2 2) , (< 3 3)	1 1 ; 2 2 ; 3 3
+---+---+---+   1 1   2 2   3 3   +---+---+---+	+---+---+---+   1 1   2 2   3 3   +---+---+---+

## 2.7 Summary

In conclusion, every data object in J is an array, with zero, one or more dimensions. An array may be an array of numbers, or an array of characters, or an array of boxes (and there are further possibilities).

This brings us to the end of Chapter 2.

---

# Chapter 3: Defining Functions

J comes with a collection of functions built-in; we have seen a few, such as `*` and `+`. In this section we take a first look at how to put together these built-in functions, in various ways, for the purpose of defining our own functions.

## 3.1 Renaming

The simplest way of defining a function is to give a name of our own choice to a built-in function. The definition is an assignment. For example, to define `square` to mean the same as the built-in `*` function:

```
square =: *:
```

```
square 1 2 3 4
1 4 9 16
```

The point here is that we might prefer our own name as more memorable. We can use different names for the monadic and dyadic cases of the same built-in function:

```
Ceiling =: >.
Max      =: >.
```

Ceiling 1.7	3 Max 4
2	4

## 3.2 Inserting

Recall that `(+ / 2 3 4)` means  $2+3+4$ , and similarly `(* / 2 3 4)` means  $2*3*4$ . We can define a function and give it a name, say `sum`, with an assignment:

```
sum =: + /
```

```
sum 2 3 4
9
```

Here, `sum =: +/` shows us that `+/` is by itself an expression which denotes a function.

This expression `+/` can be understood as: "Insert" `( / )` applied to the function `+` to produce a list-summing function.

That is, `/` is itself a kind of function. It takes one argument, on its left. Both its argument and its result are functions.

## 3.3 Terminology: Verbs, Operators and Adverbs

We have seen functions of two kinds. Firstly, there are "ordinary" functions, such as `+` and `*`, which compute numbers from numbers. In J these are called "verbs".

Secondly, we have functions, such as `/`, which compute functions from functions. Functions of this kind will here be called "operators", to distinguish them from verbs.

(The word "operator" is not used in the official J literature. However, in this book, for the purpose of explanation, it will be convenient to have a single word, "operator", to describe any kind of function which is not a verb. Then we may say that every function in J is either a verb or an operator.)

Operators which take one argument are called "adverbs". An adverb always takes its argument on the left. Thus we say that in the expression `( + / )` the adverb `/` is applied to the verb `+` to produce a list-summing verb.

The terminology comes from the grammar of English sentences: verbs act upon things and adverbs modify verbs.

## 3.4 Commuting

Having seen one adverb, `( / )`, let us look at another. The adverb `~` has the effect of

exchanging left and right arguments.

'a' , 'b'	'a' ,~ 'b'
ab	ba

The scheme is that for a dyad  $f$  with arguments  $x$  and  $y$

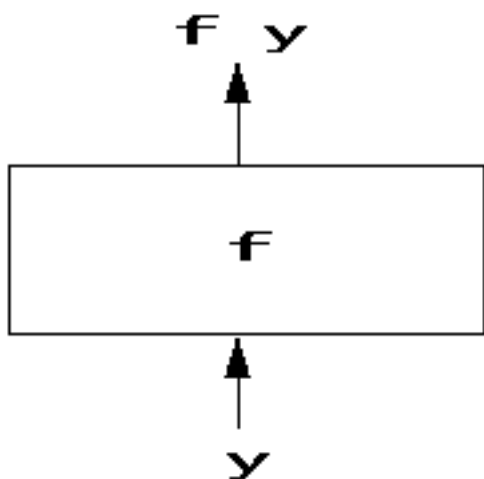
$$x \; f \sim y \qquad \text{means} \qquad y \; f \; x$$

For another example, recall the residue verb  $|$  where  $2 \; | \; 7$  means, in conventional notation, "7 mod 2". We can define a mod verb:

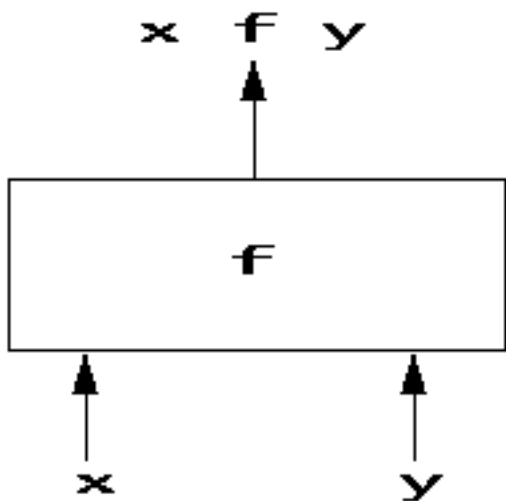
$$\text{mod} \; =: \; | \; \sim$$

7 mod 2	2   7
1	1

Let me draw some pictures. Firstly, here is a diagram of function  $f$  applied to an argument  $y$  to produce a result  $(f \; y)$ . In the diagram the function  $f$  is drawn as a rectangle and the arrows are arguments flowing into, or results flowing out of, the function. Each arrow is labelled with an expression.

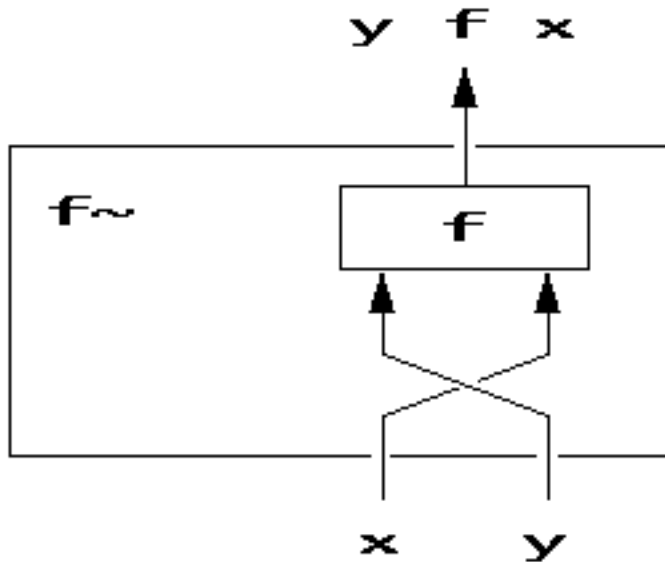


Here is a similar diagram for a dyadic  $\mathfrak{f}$  applied to arguments  $x$  and  $y$  to produce  $(x \mathfrak{f} y)$ .



Here now is a diagram for the function  $(\mathfrak{f} \sim)$ , which can be pictured as containing inside itself the function  $\mathfrak{f}$ , together with a crossed arrangement of arrows.





## 3.5 Bonding

Suppose we wish to define a verb `double` such that `double x` means `x * 2`. That is, `double` is to mean "multiply by 2". We define it like this:

```
double =: * & 2
```

```
double 3
```

6

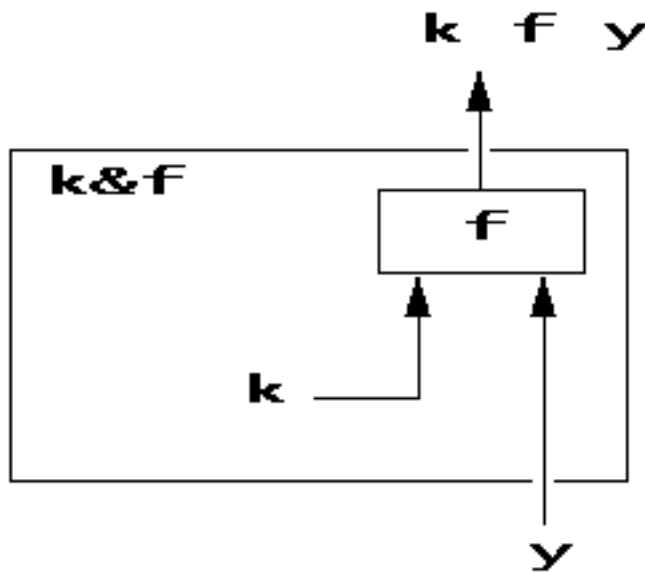
Here we take a dyad, `*`, and produce from it a monad by fixing one of the two arguments at a chosen value (in this case, 2). The `&` operator forms a bond between a function and a value for one argument. The bonding operation is also known as "currying". Instead of fixing the right argument we could fix the left. For example, suppose that the rate of sales tax is 10%, then a function to compute the tax, from the purchase-price is:

```
tax =: 0.10 & *
```

```
tax 50
```

5

Here is a diagram illustrating function `k&f`.



## 3.6 Terminology: Conjunctions and Nouns

The expression ( \* & 2 ) can be described by saying that the & operator is a function which is applied to two arguments (the verb \* and the number 2), and the result is the "doubling" verb.

A two-argument operator such as & is called in J a "conjunction", because it conjoins its two arguments. By contrast, recall that adverbs are operators with only one argument.

Every function in J, whether built-in or user-defined, belongs to exactly one of the four classes: monadic verbs, dyadic verbs, adverbs or conjunctions. Here we regard an ambivalent symbol such as - as denoting two different verbs: monadic negation or dyadic subtraction.

Every expression in J has a value of some type. All values which are not functions are data (in fact, arrays, as we saw in the previous section).

In J, data values, that is, arrays, are called "nouns", in accordance with the English-grammar analogy. We can call something a noun to emphasize that it's not a verb, or an array to emphasize that it may have several dimensions.

## 3.7 Composition of Functions

Consider the English language expression: the sum of the squares of the numbers 1 2 3, that is,  $(1+4+9)$ , or 14. Since we defined above verbs for `sum` and `square`, we are in a position to write the J expression as:

```
sum square 1 2 3
14
```

A single sum-of-the-squares function can be written as a composite of `sum` and `square`:

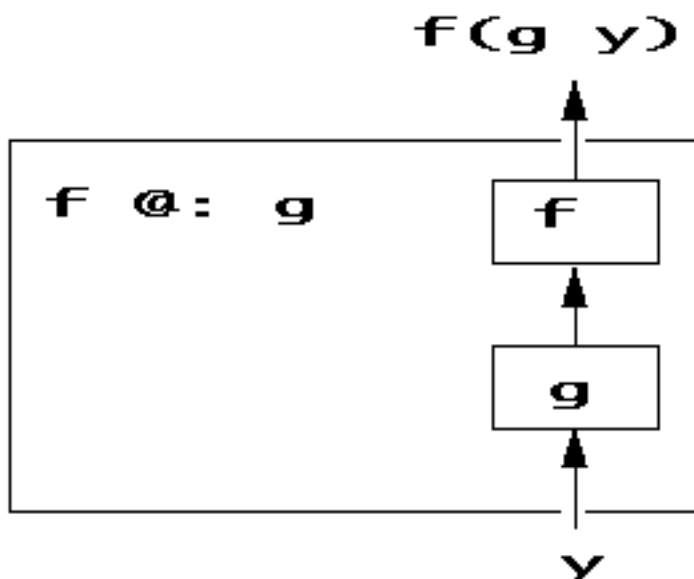
```
sumsq =: sum @: square

sumsq 1 2 3
14
```

The symbol `@:` (at colon) is called a "composition" operator. The scheme is that if `f` and `g` are verbs, then for any argument `y`

$$(f @: g) y \quad \text{means} \quad f (g y)$$

Here is a diagram for the scheme:



At this point, the reader may be wondering why we write  $(f @: g)$  and not simply  $(f g)$  to denote composition. The short answer is that  $(f g)$  means something else, which we will come to.

For another example of composition, a temperature in degrees Fahrenheit can be converted to Celsius by composing together functions  $s$  to subtract 32 and  $m$  to multiply by  $5/9$ .

```
s      =: - & 32
m      =: * & (5%9)
convert =: m @: s
```

s 212	m s 212	convert 212
180	100	100

For clarity, these examples showed composition of named functions. We can of course compose expressions denoting functions:

```
conv =: (* & (5%9)) @: (- & 32)
conv 212
100
```

We can apply an expression denoting a function, without giving it a name:

```
(* & (5%9)) @: (- & 32) 212
100
```

The examples above showed composing a monad with a monad. The next example shows we can compose a monad with a dyad. The general scheme is:

$$x (f @: g) y \quad \text{means} \quad f (x g y)$$

For example, the total cost of an order for several items is given by multiplying

quantities by corresponding unit prices, and then summing the results. To illustrate:

```
P =: 2 3          NB. prices
Q =: 1 100        NB. quantities

total =: sum @: *
```

P	Q	P*Q	sum P * Q	P total Q
2 3	1 100	2 300	302	302

For more about composition, see [Chapter 08 p8](#).

## 3.8 Trains of Verbs

Consider the expression "no pain, no gain". This is a compressed idiomatic form, quite comprehensible even if not grammatical in construction - it is not a sentence, having no main verb. J has a similar notion: a compressed idiomatic form, based on a scheme for giving meaning to short lists of functions. We look at this next.

### 3.8.1 Hooks

Recall the verb `tax` we defined above to compute the amount of tax on a purchase, at a rate of 10%. The definition is repeated here:

```
tax =: 0.10 & *
```

The amount payable on a purchase is the purchase-price plus the computed tax. A verb to compute the amount payable can be written:

```
payable =: + tax
```

If the purchase price is, say, \$50, we see:

tax 50	50 + tax 50	payable 50
5	55	55

In the definition `(payable =: + tax)` we have a sequence of two verbs `+` followed by `tax`. This sequence is isolated, by being on the right-hand side of the assignment. Such an isolated sequence of verbs is called a "train", and a train of 2 verbs is called a "hook".

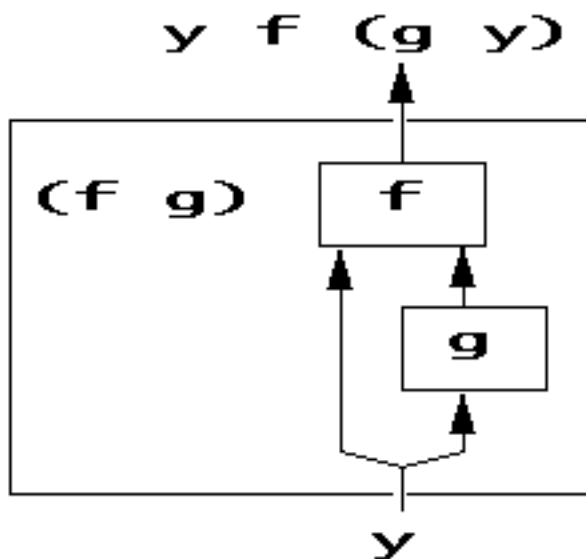
We can also form a hook just by isolating the two verbs inside parentheses:

```
(+ tax) 50
55
```

The general scheme for a hook is that if  $f$  is a dyad and  $g$  is a monad, then for any argument  $y$ :

$(f\ g)\ y$  means  $y\ f\ (g\ y)$

Here is a diagram for the scheme:



For another example, recall that the "floor" verb `<.` computes the whole-number part of its argument. Then to test whether a number is a whole number or not, we

can ask whether it is equal to its floor. A verb meaning "equal-to-its-floor" is the hook (= <.) :

```
wholenumber =: = <.
```

<code>y =: 3 2.7</code>	<code>&lt;. y</code>	<code>y = &lt;. y</code>	<code>wholenumber y</code>
3 2.7	3 2	1 0	1 0

### 3.8.2 Forks

The arithmetic mean of a list of numbers `L` is given by the sum of `L` divided by the number of items in `L`. (Recall that number-of-items is given by the monadic verb `#.`)

<code>L =: 3 5 7 9</code>	<code>sum L</code>	<code># L</code>	<code>(sum L) % (# L)</code>
3 5 7 9	24	4	6

A verb to compute the mean as the sum divided by the number of items can be written as a sequence of three verbs: `sum` followed by `%` followed by `#`.

```
mean =: sum % #
```

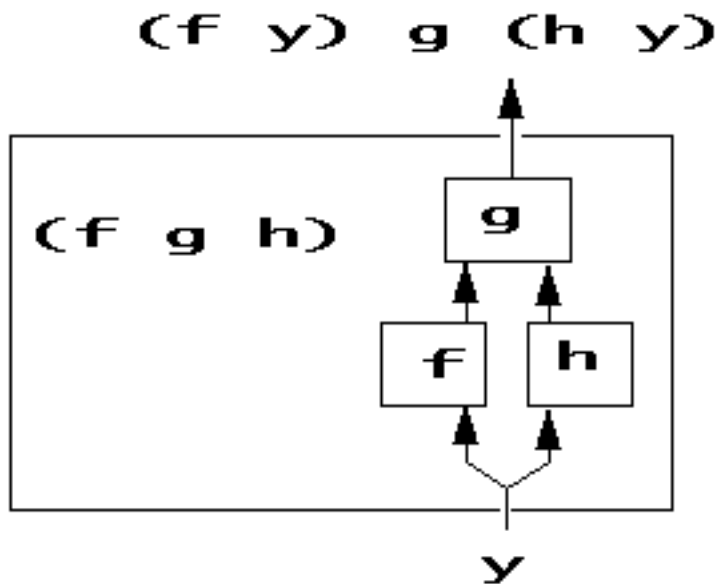
```
mean L
```

6

An isolated sequence of three verbs is called a fork. The general scheme is that if `f` is a monad, `g` is a dyad and `h` is a monad then for any argument `y`,

```
(f g h) y means (f y) g (h y)
```

Here is a diagram of this scheme:



Hooks and forks are sequences of verbs, also called "trains" of verbs. For more about trains, see [Chapter 09 p9](#).

## 3.9 Putting Things Together

Let us now try a longer example which puts together several of the ideas we saw above.

The idea is to define a verb to produce a simple display of a given list of numbers, showing for each number what it is as a percentage of the total.

Let me begin by showing you a complete program for this example, so you can see clearly where we are going. I don't expect you to study this in detail now, because explanation will be given below. Just note that we are looking at a program of 8 lines, defining a verb called `display` and its supporting functions.

NB. `display` verb: tabulate as percentages

```
percent  =: (100 & *) @: (% +/)
round    =: <. @: (+&0.5)
comp     =: round @: percent
br       =: ,. ; (,. @: comp)
tr       =: ('Data'; 'Percentages') & ,
reshape  =: 2 2 & $
display  =: reshape @: tr @: br
```



To show that this verb displays the data as given and as computed percentages:

```
display 15 30 15
+-----+-----+
|Data|Percentages|
+-----+-----+
|15  |25         |
|30  |50         |
|15  |25         |
+-----+-----+
```

The verb `percent` computes the percentages, dividing each number by the total, with the hook `(% +/)` and then multiplying by 100. To save you looking backwards and forwards, the definition of `percent` is repeated here:

```
percent =: (100 & *) @: (% +/)
```

To illustrate with simple data:

```
data =: 3 5
```

data	data % +/ data	(% +/) data	percent data
3 5	0.375 0.625	0.375 0.625	37.5 62.5

Let us round the percentages to the nearest whole number, by adding 0.5 to each and then taking the floor (the integer part) with the verb `<.`. The verb `round` is:

```
round =: <. @: (+&0.5)
```

Then the verb to compute the displayed values from the data is:

```
comp =: round @: percent
```

data	comp data
3 5	38 63

Now we want to show the data and computed values in columns. To make a 1-column table out of a list, we can use the built-in verb `,.` (comma dot, called "Ravel Items").

data	,. data	,. comp data
3 5	3 5	38 63

To make the bottom row of the display, we define verb `br` as a fork which links together the data and the computed values, both as columns:

```
br =: ,. ; (,. @: comp)
```

data	br data
3 5	<pre> +---+   3   38     5   63   +---+</pre>

To make the top row of the display (the column headings), here is one possible way. The bottom row will be a list of two boxes. On the front of this list we stick two more boxes for the top row, giving a list of 4 boxes. To do this we define a verb `tr`:

```
tr =: ('Data'; 'Percentages') & ,
```

data	br data	tr br data
3 5	<pre> +-----+   3   38     5   63   +-----+ </pre>	<pre> +-----+-----+-----+-----+   Data   Percentages   3   38                             5   63   +-----+-----+-----+-----+ </pre>

All that remains is to reshape this list of 4 boxes into a 2 by 2 table, using the reshape verb defined as:

```
reshape =: 2 2 & $
```

```
reshape tr br data
```

```

+-----+-----+
| Data | Percentages |
+-----+-----+
| 3    | 38             |
| 5    | 63             |
+-----+-----+

```

and so we put everything together:

```
display =: reshape @: tr @: br
```

```
display data
```

```

+-----+-----+
| Data | Percentages |
+-----+-----+
| 3    | 38             |
| 5    | 63             |
+-----+-----+

```

This `display` verb has two aspects: the function `comp` which computes the values (the rounded percentages), and the remainder which is concerned to present the results. By changing the definition of `comp`, we can `display` a tabulation of the values of other functions. Suppose we define `comp` to be the built-in square-root verb `(%:)`.

```
comp =: %:
```

We would also want to change the column-headings in the top row, specified by the `tr` verb:

```
tr    =: ('Numbers'; 'Square Roots') & ,

display 1 4 9 16
+-----+-----+
|Numbers|Square Roots|
+-----+-----+
|  1    |1          |
|  4    |2          |
|  9    |3          |
| 16    |4          |
+-----+-----+
```

In review, we have seen a small J program with some characteristic features of J: bonding, composition and a fork. As with all J programs, this is only one of the many possible ways to write it.

This is the end of Chapter 3.

---

Copyright © Roger Stokes 2001. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 05Jun01

# Chapter 4: Scripts and Explicit Functions

What is called a "script" is a sequence of lines of J where the whole sequence can be replayed on demand to perform a computation. The themes of this chapter are scripts, functions defined by scripts, and scripts in files.

## 4.1 Text

Here is an assignment to the variable `txt`:

```
txt =: 0 : 0
What is called a "script" is
a sequence of lines of J.
)
```

The expression `0 : 0` means "as follows", that is, `0 : 0` is a verb which takes as its argument, and delivers as its result, whatever lines are typed following it, down to the line beginning with the solo right- parenthesis.

The value of `txt` is these two lines, in a single character string. The string contains line-feed (LF) characters, which cause `txt` to be displayed as several lines. `txt` has a certain length, it is rank 1, that is, just a list, and it contains 2 line-feed characters.

```
txt
What is called a "script" is
a sequence of lines of J.
```

\$ txt	# \$ txt	+ / txt = LF
55	1	2

Let us say that `t` is a "text" variable, that is, a character string with zero or more line-feed characters.

## 4.2 Scripts for Procedures

Here we look at computations described as step-by-step procedures to be followed. For a very simple example, the Fahrenheit-to-Celsius conversion can be described in two steps. Given some temperature `T` say in degrees Fahrenheit:

```
T =: 212
```

then the first step is subtracting 32. Call the result `t`, say.

```
t =: T - 32
```

The second step is multiplying `t` by `5%9` to give the temperature in degrees Celsius.

```
t * 5 % 9  
100
```

Suppose we intend to perform this computation several times with different values of `T`. We could record this two-line procedure as a script which can be replayed on demand. The script consists of the lines of J stored in a text variable, thus:

```
script =: 0 : 0  
t =: T - 32  
t * 5 % 9  
)
```

Scripts like this can be executed with the built-in J verb `0 !: 111` which we can call, say, `do`.

```
do =: 0 !: 111
```

```
do script
```

We should now see the lines on the screen just as though they had been typed in

from the keyboard:

```
t =: T - 32
t * 5 % 9
100
```

We can run the script again with a different value for T

```
T =: 32
do script
t =: T - 32
t * 5 % 9
0
```

## 4.3 Explicitly-Defined Functions

Functions can be defined by scripts. Here is an example, the Fahrenheit-to-Celsius conversion as a verb.

```
Celsius =: 3 : 0
t =: y. - 32
t * 5 % 9
)
```

Celsius 32 212	1 + Celsius 32 212
0 100	1 101

Let us look at this definition more closely.

### 4.3.1 Heading

The function is introduced with the expression `3 : 0` which means: "a verb as follows". (By contrast, recall that `0 : 0` means "a character string as follows").

The colon in `3 : 0` is a conjunction. Its left argument (`3`) means "verb". Its right argument (`0`) means "lines following". For more details, see [Chapter 12 p12](#). A function introduced in this way is called "explicitly-defined", or just "explicit".

### 4.3.2 Meaning

The expression `(Celsius 32 212)` applies the verb `Celsius` to the argument `32 212`, by carrying out a computation which can be described, or modelled, like this:

```
y. =: 32 212
t  =: y. - 32
t  * 5 % 9
0 100
```

Notice that, after the first line, the computation proceeds according to the script.

### 4.3.3 Argument Variable(s)

The value of the argument `(32 212)` is supplied to the script as a variable named `y.` (letter-y dot). This "argument variable" is always named `y.` in a monadic function. (In a dyadic function, as we shall see below, the right argument is always named `y.` and the left is `x.`)

### 4.3.4 Local Variables

Here is our definition of `Celsius` repeated:

```
Celsius =: 3 : 0
t =: y. - 32
t * 5 % 9
)
```

We see it contains an assignment to a variable `t`. This variable is used only during the execution of `Celsius`. Unfortunately this assignment to `t` interferes with the value of any other variable also called `t`, defined outside `Celsius`, which we happen to be using at the time. To demonstrate:

```
t =: 'hello'
```



```

    Celsius 212
100

    t
180

```

We see that the variable `t` with original value (`'hello'`) has been changed in executing `Celsius`. To avoid this undesirable effect, we declare that `t` inside `Celsius` is to be a strictly private affair, distinct from any other variable called `t`.

For this purpose there is a special form of assignment, with the symbol `=.` (equal dot). Our revised definition becomes:

```

    Celsius =: 3 : 0
t =. y. - 32
t * 5 % 9
)

```

and we say that `t` in `Celsius` is a local variable, or that `t` is local to `Celsius`. By contrast, a variable defined outside a function is said to be global. Now we can demonstrate that in `Celsius` assignment to local variable `t` does not affect any global variable `t`

```

    t =: 'hello'

    Celsius 212
100

    t
hello

```

The argument-variable `y.` is also a local variable. Hence the evaluation of `(Celsius 32 212)` is more accurately modelled by the computation:

```

    y. =. 32 212
    t =. y. - 32
    t * 5 % 9
0 100

```

### 4.3.5 Dyadic Verbs

Celsius is a monadic verb, introduced with `3 : 0` and defined in terms of the single argument `y.`. By contrast, a dyadic verb is introduced with `4 : 0`. The left and right arguments are always named `x.` and `y.` respectively. Here is an example. The "positive difference" of two numbers is the larger minus the smaller.

```
posdiff =: 4 : 0
larger =. x. >. y.
smaller =. x. <. y.
larger - smaller
)
```

3 posdiff 4	4 posdiff 3
1	1

### 4.3.6 One-Liners

A one-line script can be written as a character string, and given as the right argument of the colon conjunction.

```
PosDiff =: 4 : '(x. >. y.) - (x. <. y.)'
4 PosDiff 3
1
```

### 4.3.7 Flow of Control

In the examples we have seen so far of functions defined by scripts, execution begins with the first line, proceeds to the next, and so on to the last. This straight-through path is not the only path possible. The path can be controlled by conditions which can be tested in the course of the computation.

Here is a simple example, a variation of `PosDiff` where the course of the computation is guided by the presence of what are called the "control words" `if.`

```
do. else. end. .
    POSDIFF =: 4 : 0
if.    x. > y.
do.    x. - y.
else.  y. - x.
```

```
end.  
)  
  3 POSDIFF 4  
1
```

See [Chapter 12 p12](#) for more on control words.

## 4.4 Tacit and Explicit Compared

We have now seen two different styles of function definition. The explicit style, introduced in this chapter, is so called because it explicitly mentions variables standing for arguments. Thus in `POSDIFF` above, the variable `y.` is an explicit mention of an argument.

By contrast, the style we looked at in the previous chapter is called "tacit", because there is no mention of variables standing for arguments. For example, compare explicit and tacit definitions of the positive-difference function:

```
epd =: 4 : '(x. >. y.) - (x. <. y.)'
```

```
tpd =: >. - <.
```

Many functions defined in the tacit style can also be defined explicitly, and vice versa. Which style is preferable depends on what seems most natural, in the light of however we conceive the function to be defined. The choice lies between breaking down the problem into, on the one hand, a scripted sequence of steps or on the other hand into a collection of smaller functions.

The tacit style allows a compact definition. For this reason, tacit functions lend themselves well to systematic analysis and transformation. Indeed, the J system can, for a broad class of tacit functions, automatically compute such transformations as inverses and derivatives.

## 4.5 Functions as Values

A function is a value, and a value can be displayed by entering an expression. An expression can be as simple as a name. Here are some values of tacit and explicit

functions:

```
- & 32
+--+---+
| - | & | 32 |
+--+---+

epd
+--+-----+
| 4 | : | (x. >. y.) - (x. <. y.) |
+--+-----+

Celsius
+--+-----+
| 3 | : | t =. y. - 32 |
| | | t * 5 % 9 |
+--+-----+
```

The value of each function is here represented as a boxed structure. Other representations are available: see Chapter 27.

## 4.6 Script Files

We have seen scripts (lines of J) used for definitions of single variables: text variables or functions. By contrast, a file holding lines of J as text can store many definitions. Such a file is called a script file, and its usefulness is that all its definitions together can be executed by reading the file.

Here is an example. Create a file on your computer called, say, `myscript`. Use a text-editor of your choice to create the file. The file should contain 2 lines of text like the following:

```
squareroot =: %:
z =: 1 , (2+2) , (4+5)
```

Having created this 2-line script file, we can execute it by typing at the keyboard:

```
0! :1 < 'myscript'
```

and we should now see the lines on the screen just as though they had been typed

from the keyboard.

```
squareroot =: %:  
z =: 1 , (2+2) , (4+5)
```

We can now compute with the definitions we have just loaded in from the file:

```
z  
1 4 9  
  
squareroot z  
1 2 3
```

The activities in a J session will be typically a mixture of editing script files, loading or reloading the definitions from script files, and initiating computations at the keyboard. What carries over from one session to another is only the script files. The state, or memory, of the J system itself disappears at the end of the session, along with all the definitions entered during the session. Hence it is a good idea to ensure, before ending a J session, that any script file is up to date, that is, it contains all the definitions you wish to preserve.

At the beginning of a session the J system will automatically load a designated script file, called the "profile". (See [Chapter 26 p25](#) for more details). The profile can be edited, and is a good place to record any definitions of your own which you find generally useful.

We have now come to the end of Chapter 4 and of Part 1. The following chapters will treat, in more depth and detail, the themes we have touched upon in Part 1.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 18 Aug 2002

# Chapter 5: Building Arrays

This chapter is about building arrays. First we look at building arrays from lists, and then at joining arrays together in various ways to make larger arrays.

## 5.1 Building Arrays by Shaping Lists

The dyadic verb `$` (dollar) is called "Shape". The expression `(x $ y)` produces an array of the items of the list `y`, with shape `x`, that is, with dimensions given by the list `x`. For example:

<code>2 2 \$ 0 1 2 3</code>	<code>2 3 \$ 'ABCDEF'</code>
0 1 2 3	ABC DEF

If the list `y` contains fewer than the number of items needed, then `y` is re-used in cyclical fashion to make up the number of items needed. This means that an array can be built to show some simple patterning, such as all elements being the same, for example.

<code>2 3 \$ 'ABCD'</code>	<code>2 2 \$ 1</code>	<code>3 3 \$ 1 0 0 0</code>
ABC DAB	1 1 1 1	1 0 0 0 1 0 0 0 1

Instead of re-using values from `y`, a value to fill the array can be specified with the "Customize" conjunction `(!.)` (exclamation-mark dot).

```
2 3 ($ !. '*' ) 'ab'
ab*
***
```

The "Shape" verb, dyadic \$, has a companion verb, "ShapeOf" (monadic \$), which yields the list-of-dimensions, that is, shape, of its argument. To illustrate:

A =: 2 3 \$ 'ABCDEF'	\$ A	a =: 'pqr'	\$ a
ABC DEF	2 3	pqr	3

An array can be of length zero in any of its dimensions. If EL is a zero-length, or empty, list then it has no items, and so, after appending an item to it, the result will have one item.

EL =: 0\$0	# EL	EL , 1
	0	1

Similarly, if ET is an empty table with no rows, and say, 3 columns, then after adding a row, the result will have one row.

ET =: 0 3 \$ 'x'	\$ ET	\$ ET , 'pqr'
	0 3	1 3

It sometimes occurs that we need to build a scalar. A scalar has no dimensions, that is, its dimension-list is empty. We can give an empty list as the left argument of \$ to make a scalar:

```
(0$0) $ 99
99
```

We said that  $(x \$ y)$  produces an  $x$ -shaped array of the items of  $y$ . If  $y$  is a table, then each item of  $y$  will itself be a list (a row). Hence in general the shape of  $(x\$y)$  will be not just  $x$ , but rather  $x$  followed by the shape of an item of  $y$ .

A	3 \$ A	\$ 3 \$ A
ABC DEF	ABC DEF ABC	3 3

The next sections look at building new arrays by joining together arrays we already have.

## 5.2 Appending, or Joining End-to-End

Recall that any array can be regarded as a list of items, so that for example the items of a table are its rows. The verb `,` (comma) is called "Append". The expression  $(x,y)$  is a list of the items of  $x$  followed by the items of  $y$ .

```
B =: 2 3 $ 'UVWXYZ'
b =:    3 $ 'uvw'
```

a	b	a , b	A	B	A , B
pqr	uvw	pqruvw	ABC DEF	UVW XYZ	ABC DEF UVW XYZ

In the example of  $(A,B)$  above. the items of  $A$  are lists of length 3, and so are the items of  $B$ . Hence items of  $A$  are compatible with, that is, have the same rank and length as items of  $B$ . What if they do not? In this case the "Append" verb will helpfully try to stretch one argument to fit the other, by bringing them to the same rank, padding to length, and replicating scalars as necessary. This is shown the



following examples.

### 5.2.1 Bringing To Same Rank

Suppose we want to append a row to a table. For example, consider appending the 3-character list `b` (above) to the 2 by 3 table `A` (above) to form a new row.

A	b	A , b
ABC DEF	uvw	ABC DEF uvw

Notice that we want the two items of `A` to be followed by the single item of `b`, but `b` is not a 1-item affair. We could do it by reshaping `b` into a 1 by 3 table, that is, by raising the rank of `b`. However, this is not necessary, because, as we see, the "Append" verb has automatically stretched the low-rank argument into a 1-item array, by supplying leading dimension(s) of 1 as necessary.

A	b	A , (1 3 \$ b)	A , b	b , A
ABC DEF	uvw	ABC DEF uvw	ABC DEF uvw	uvw ABC DEF

### 5.2.2 Padding To Length

When the items of one argument are shorter than the items of the other, they will be padded out to length. Characters arrays are padded with the blank character, numerical arrays with zero.

A	A , 'XY'	(2 3 \$ 1) , 9 9

ABC	ABC	1 1 1
DEF	DEF	1 1 1
	XY	9 9 0

### 5.2.3 Replicating Scalars

A scalar argument of "Append" is replicated as necessary to match the other argument. In the following example, notice how the scalar '\*' is replicated, but the vector (1 \$ '\*') is padded.

A	A , '*'	A , 1 \$ '*'
ABC DEF	ABC DEF ***	ABC DEF *

## 5.3 Stitching, or Joining Side-to-Side

The verb , . (comma dot) is called "Stitch". In the expression (x , . y) each item of x has the corresponding item of y appended.

a	b	a , . b	A	B	A , . B
pqr	uvw	pu qv rw	ABC DEF	UVW XYZ	ABCUVW DEFXYZ

## 5.4 Laminating, or Joining Face-to-Face

The verb , : (comma colon) is called "Laminate". In the expression (x , : y), if x and y are, say, two similar tables, then we can imagine the result as one table laid on top of the other to form a 3-dimensional array. Thus the arguments are joined along a new dimension, always of length 2. The result has two items, of which the first is x and the second is y. This means that the new dimension is the first.

a	b	a ,: b	A	B	A ,: B
pqr	uvw	pqr uvw	ABC DEF	UVW XYZ	ABC DEF  UVW XYZ

## 5.5 Linking

The verb ; (semicolon) is called "Link". It is convenient for building lists of boxes.

'good' ; 'morning'	5 ; 12 ; 1995
+-----+-----+  good morning  +-----+-----+	+--+---+-----+  5 12 1995  +--+---+-----+

Notice how the example of 5;12;1995 shows that ( $x;y$ ) is not invariably just ( $<x$ ), ( $<y$ ). Since "Link" is intended for building lists of boxes, it recognises when its right argument is already a list of boxes. If we define a verb which does produce ( $<x$ ), ( $<y$ )

```
foo =: 4 : '(<x.) , (<y.)'
```

we can compare these two:

1 ; 2 ; 3	1 foo 2 foo 3
+--+---+  1 2 3  +--+---+	+--+-----+  1 +--+---+     2 3       +--+---+  +--+-----+

---

## 5.6 Unbuilding Arrays

We have looked at four dyadic verbs: "Append" ( , ), "Stitch" ( , . ), "Laminate" ( , : ) and "Link" ( ; ). Each of these has a monadic case, which we now look at.

### 5.6.1 Razing

Monadic ; is called "Raze". It unboxes elements of the argument and assembles them into a list.

B =: 2 2 \$ 1;2;3;4	; B	\$ ; B
<div>+--++</div> <div>  1   2  </div> <div>+--++</div> <div>  3   4  </div> <div>+--++</div>	1 2 3 4	4

### 5.6.2 Ravelling

Monadic , is called "Ravel". It assembles elements of the argument into a list.

B	, B	\$ , B
<div>+--++</div> <div>  1   2  </div> <div>+--++</div> <div>  3   4  </div> <div>+--++</div>	<div>+--+--+--+</div> <div>  1   2   3   4  </div> <div>+--+--+--+</div>	4

### 5.6.3 Ravelling Items

Monadic , . is called "Ravel Items". It separately ravel each item of the argument to form a table.

<code>k =: 2 2 3 \$ i. 12</code>	<code>,. k</code>
<pre> 0  1  2 3  4  5  6  7  8 9 10 11 </pre>	<pre> 0 1 2 3  4  5 6 7 8 9 10 11 </pre>

"Ravel Items" is useful for making a 1-column table out of a list.

<code>b</code>	<code>,. b</code>
<code>uvw</code>	<pre> u v w </pre>

### 5.6.4 Itemizing

Monadic `,:` makes a 1-item array out of any array, by adding a leading dimension of 1.

<code>A</code>	<code>,: A</code>	<code>\$ ,: A</code>
<pre> ABC DEF </pre>	<pre> ABC DEF </pre>	<pre> 1 2 3 </pre>

## 5.7 Arrays Large and Small

For small arrays, where the contents can be listed on a single line, there are alternatives to using `$`, which avoid the need to give the dimensions explicitly.

<code>&gt; 1 2 ; 3 4 ; 5 6</code>	<code>1 2 , 3 4 ,: 5 6</code>
-----------------------------------	-------------------------------

1 2	1 2
3 4	3 4
5 6	5 6

To build large tables, a convenient method is as follows. First, here is a "utility" verb (that is, a verb which is useful for present purposes, but we don't need to study its definition now.)

```
ArrayMaker =: ". ;. _2
```

The purpose of `ArrayMaker` is to build a numeric table row by row from the lines of a script.

```
table =: ArrayMaker 0 : 0
1 2 3
4 5 6
7 8 9
)
```

table	\$ table
1 2 3 4 5 6 7 8 9	3 3

(See [Chapter 17 p17](#) for an explanation of how `ArrayMaker` works). Arrays of boxes can also be entered from a script in the same way:

```
X =: ArrayMaker 0 : 0
'hello' ; 1 2 3 ; 8
'Waldo' ; 4 5 6 ; 9
)
```

X	\$ X
+-----+-----+--+  hello 1 2 3 8  +-----+-----+--+  waldo 4 5 6 9  +-----+-----+--+	2 3

We have reached the end of Chapter 5.

---

Copyright © Roger Stokes 2000. This material may be freely reproduced, provided that this copyright notice and provision is also reproduced.

last updated 17Mar00

# Chapter 6: Indexing

Indexing is the name given to selecting of elements of arrays by position. This topic includes selecting elements, rearranging selected elements to form new arrays, and amending, or updating, selected elements of arrays.

## 6.1 Selecting

The verb `{` (left-brace) is called "From". The expression `(x { y)` selects elements from `y` according to positions given by `x`. For example, recall from Chapter 2 that if `L` is a list, then the positions of items of `L` are numbered 0 1 and so on. The expression `(0 { L)` gives the value of the first item of `L` and `1 { L` gives the second item.

<code>L =: 'abcdef'</code>	<code>0 { L</code>	<code>1 { L</code>
abcdef	a	b

The left argument of `{` is called the "index".

### 6.1.1 Common Patterns of Selection.

Several items may be selected together:

<code>L</code>	<code>0 2 4 { L</code>
abcdef	ace

Items selected from `L` may be replicated and re-ordered:



L	5 4 4 3 { L
abcdef	feed

An index value may be negative: a value of `_1` selects the last item, `_2` selects the next-to-last item and so on. Positive and negative indices may be mixed.

L	<code>_1 { L</code>	<code>_2 1 { L</code>
abcdef	f	eb

A single element of a table at, say, row 1 column 2 is selected with an index `(< 1 ; 2)`.

<code>T =: 3 3 \$ 'abcdefghi'</code>	<code>(&lt; 1 ; 2) { T</code>
abc def ghi	f

We can select from a table all elements in specified rows and columns, to produce a smaller table (called a subarray). To select a subarray consisting of, for example rows 1 and 2 and columns 0 and 1, we use an index `(< 1 2; 0 1)`

T	<code>(&lt; 1 2; 0 1) { T</code>
abc def ghi	de gh

A complete row or rows may be selected from a table. Recall that a table is a list of

items, each item being a row. Thus selecting rows from tables is just like selecting items from lists.

T	1 { T	2 1 { T
abc def ghi	def	ghi def

To select a complete column or columns, a straightforward way is to select all the rows:

T	( < 0 1 2 ; 1 ) { T
abc def ghi	beh

but there are other possibilities: see below.

## 6.1.2 Take, Drop, Head, Behead, Tail, Curtail

Next we look at a group of verbs providing some convenient short forms of indexing. There is a built-in verb { . (left brace dot, called "Take"). The first  $n$  items of list  $L$  are selected by  $(n \{ . L)$

L	2 { . L
abcdef	ab

If we take  $n$  items from  $L$  with  $(n \{ . L)$ , and  $n$  is greater than the length of  $L$ , the result is padded to length  $n$ , with zeros, spaces or empty boxes as appropriate.

For example, suppose we require to make a string of exactly 8 characters from a given string, a description of some kind, which may be longer or shorter than 8. If longer, we shorten. If shorter we pad with spaces.

<code>s =: 'pasta'</code>	<code># s</code>	<code>z =: 8 {. s</code>	<code># z</code>
pasta	5	pasta	8

There is a built-in verb `}.` (right-brace dot, called "Drop"). All but the first  $n$  items of  $L$  are selected by `(n }.) L`.

<code>L</code>	<code>2 }.</code> <code>L</code>
abcdef	cdef

The last  $n$  items of  $L$  are selected by `(-n) {. L`. All but the last  $n$  are selected by `(-n) }.` `L`

<code>L</code>	<code>_2 {. L</code>	<code>_2 }.</code> <code>L</code>
abcdef	ef	abcd

There are abbreviations of Take and Drop in the special case where  $n=1$ . The first item of a list is selected by monadic `{.` (left-brace dot, called "Head"). All but the first are selected by `}.` (right-brace dot, called "Behead").

<code>L</code>	<code>{.</code> <code>L</code>	<code>}.</code> <code>L</code>
abcdef	a	bcdef

The last item of a list is selected by monadic `{:` (left-brace colon, called "Tail").

All but the last are selected by `}: (right-brace colon, called "Curtail".`

L	{ : L	} : L
abcdef	f	abcde

## 6.2 General Treatment of Selection

It will help to have some terminology. In general we will have an n-dimensional array, but consider a 3-dimensional array. A single element is picked out by giving a plane- number, a row-number and a column-number. We say that the planes are laid out in order along the first axis, and similarly the rows along the second axis, and the columns along the third.

There is no special notation for indexing; rather the left argument of `{` is a data structure which expresses, or encodes, selections and rearrangements. This data structure can be built in any way convenient. What follows is an explanation of how to build it.

### 6.2.1 Independent Selections

The general expression for indexing is of the form `index { array`. Here `index` is an array of scalars. Each scalar in `index` gives rise to a separate independent selection, and the results are assembled together.

L	0 1 { L
abcdef	ab

### 6.2.2 Shape of Index

The shape of the results depends on the shape of `index`.

L	index =: 2 2 \$ 2 0 3 1	index { L
abcdef	2 0 3 1	ca db

The indices must lie within the range  $-\#L$  to  $(\#L) - 1$ :

L	#L	_7 { L	6 { L
abcdef	6	error	error

### 6.2.3 Scalars

Each scalar in `index` is either a single number or a box (and of course if one is a box, all are.) If the scalar is a single number it selects an item from `array`.

A =: 2 3 \$ 'abcdef'	1 { A
abc def	def

If the scalar in `index` is a box however then it contains a list of selectors which are applied to successive axes. To show where a box is used for this purpose, we can use the name `SuAx`, say, for the box function.

`SuAx =: <`

The following example selects from `A` the element at row 1, column 0.

A	(SuAx 1 0) { A

abc def	d
------------	---

## 6.2.4 Selections on One Axis

In a list of selectors for successive axes, of the form  $(\text{SuAx } p, r, c)$  say, each of  $p$ ,  $r$  and  $c$  is a scalar. This scalar is either a number or a box (and if one is boxed, all are). A number selects one thing on its axis: one plane, row or column as appropriate, as in the last example.

However, if the selector is a box it contains a list of selections all applicable to the same axis. To show where a box is used for this purpose we can use the name `Sel`, say, for the box function.

```
Sel =: <
```

For example, to select from `A` elements at row 1, columns 0 2:

A	$(\text{SuAx } (\text{Sel } 1), (\text{Sel } 0\ 2)) \{ A$
abc def	df

## 6.2.5 Excluding Things

Instead of selecting things on a particular axis, we can exclude things, by supplying a list of thing-numbers enclosed in yet another level of boxing. To show where a box is used for this purpose we can use the name `Excl`, say, for the box function.

```
Excl =: <
```

For example, to select from `A` elements at row 0, all columns excluding column 1:

A	$(\text{SuAx } (\text{Sel } 0), (\text{Sel } (\text{Excl } 1))) \{ A$
---	---

abc def	ac
------------	----

We can select all things on a particular axis by excluding nothing, that is, giving an empty list (0\$0) as a list of thing-numbers to exclude. For example, to select from A elements at row 1, all columns:

A	(SuAx (Sel 1),(Sel (Excl 0\$0))) { A
abc def	def

### 6.2.6 Simplifications

The expression (Excl 0\$0) denotes a boxed empty list. There is a built-in J abbreviation for this, namely (a:) (letter-a colon), which in this context we can think of as meaning "all".

A	(SuAx (Sel 1),(Sel a:)) { A
abc def	def

If in any index of the form (SuAx p,q,..., z), the last selector z is the "all" form, (Sel (Excl 0\$0)) or (Sel a:), then it can be omitted.

A	(SuAx (Sel 1),(Sel a:)) {A	(SuAx (Sel 1)) {A
abc def	def	def

If in any index of the form (SuAx (Sel p),(Sel q),...), the "all" form is

entirely absent, then the index can be abbreviated to  $(\text{SuAx } p;q;\dots)$ . For example, to select elements at row 1, columns 0 and 2:

A	$(\text{SuAx } (\text{Sel } 1), (\text{Sel } 0\ 2)) \{A$	$(\text{SuAx } 1;0\ 2) \{A$
abc def	df	df

Finally, as we have already seen, if selecting only one thing on each axis, a simple unboxed list is sufficient. For example to select the element at row 1, column 2:

A	$(\text{SuAx } 1;2) \{ A$	$(\text{SuAx } 1\ 2) \{ A$
abc def	f	f

### 6.2.7 Shape of the Result

Suppose that  $B$  is a 3-dimensional array:

```
B =: i. 3 3 3
```

and we define  $p$  to select planes along the first axis of  $B$ , and  $r$  to select rows along the second axis, and  $c$  to select columns along the third axis:

```
p =: 1 2
r =: 1 2
c =: 0 1
```

We see that, selecting with  $p;r;c$ , the shape of the result  $R$  is the concatenation of the shapes of  $p$ ,  $r$  and  $c$

B	$R =: (< p;r;c) \{ B$	$\$ R$	$(\$p),(\$r),(\$c)$
---	-----------------------	--------	---------------------



0 1 2 3 4 5 6 7 8  9 10 11 12 13 14 15 16 17  18 19 20 21 22 23 24 25 26	12 13 15 16  21 22 24 25	2 2 2	2 2 2
--	--------------------------------------	-------	-------

`B` is 3-dimensional, and so is `R`. As we would expect, this concatenation-of-shapes holds when a selector (`r`, say) is a list of length one:

<code>r =: 1 \$ 1</code>	<code>S =: (&lt; p;r;c){B</code>	<code>\$ S</code>	<code>(\$p),(\$r),(\$c)</code>
1	12 13 21 22	2 1 2	2 1 2

and the concatenation-of-shapes holds when selector `r` is a scalar:

<code>r =: 1</code>	<code>T =: (&lt; p;r;c){B</code>	<code>\$ T</code>	<code>(\$p),(\$r),(\$c)</code>	<code>\$ r</code>
1	12 13 21 22	2 2	2 2	

In this last example, `r` is a scalar, so the shape of `r` is an empty list, and so the axis corresponding to `r` has disappeared, and so the result `T` is 2-dimensional.

## 6.3 Amending (or Updating) Arrays

Sometimes we need to compute an array which is the same as an existing array

except for new values at a comparatively small number of positions. We may speak of 'updating' or 'amending' an array at selected positions. The J function for amending arrays is } (right brace, called "Amend").

### 6.3.1 Amending with an Index

To amend an array we need three things:

- the original array
- a specification of the position(s) at which the original is to be amended. This can be an index exactly like the index we have seen above for selection with {.
- new values to replace existing elements at specified positions.

Consequently the J expression to perform an amendment may have the general form:

```
newvalues index } original
```

For example: to amend list L to replace the first item (at index 0) with '\*':

L	new=: '*'	index=:0	new index } L
abcdef	*	0	*bcdef

} is an adverb, which takes index as its argument to yield the dyadic amending verb (index }).

```
ReplaceFirst =: 0 }
'*' ReplaceFirst L
*bcdef
```

(index }) is a verb like any other, dyadic and yielding a value in the usual way. Therefore to change an array by amending needs the whole of the result to be reassigned to the old name. Thus amendment often takes place on the pattern:

```
A =: new index } A
```

The J system ensures that this is an efficient computation with no unnecessary movement of data.

To amend a table at row 1 column 2, for example:

A	'*' (< 1 2) } A
abc def	abc de*

To amend multiple elements, a list of new values can be supplied, and they are taken in turn to replace a list of values selected by an index

L	'*#' 1 2 } L
abcdef	a*#def

## 6.3.2 Amending with a Verb

Suppose that  $y$  is a list of numbers, and we wish to amend it so that all numbers exceeding a given value  $x$  are replaced by  $x$ . (For the sake of this example, we here disregard the built-in J verb  $(<.)$  for this function.)

The indices at which  $y$  is to be amended must be computed from  $x$  and  $y$ . Here is a function  $f$  to compute the indices:

```
f =: 4 : '(y. > x.) # (i. # y.)'
```

x =: 100	y =: 98 102 101 99	y > x	x f y
----------	--------------------	-------	-------

100	98 102 101 99	0 1 1 0	1 2
-----	---------------	---------	-----

The amending is done, in the way we have seen above, by supplying indices of ( $x$   $f$   $y$ ):

$y$	$x (x f y) \}$ $y$
98 102 101 99	98 100 100 99

The "Amend" adverb  $\}$  allows the expression ( $x (x f y) \}$   $y$ ) to be abbreviated as ( $x f \}$   $y$ ).

$x (x f y) \}$ $y$	$x f \}$ $y$
98 100 100 99	98 100 100 99

Since  $\}$  is an adverb, it can accept as argument either the indices ( $x f y$ ) or the verb  $f$ .

```
cap =: f }
      10 cap 8 9 10 11
8 9 10 10
```

Note that if verb  $f$  is to be supplied as argument to adverb  $\}$ , then  $f$  must be a dyad, although it may ignore  $x$  or  $y$ .

### 6.3.3 Linear Indices

We have just looked at amending lists with a verb. The purpose of the verb is to find the places at which to amend, that is, to compute from the values in a list the indices at which to amend. With a table rather than a list, the indices would have to be 2- dimensional, and the task of the verb in constructing the indices would be

correspondingly more difficult. It would be easier to flatten a table into a linear list, amend it as a list, and rebuild the list into a table again.

For example, suppose we have a table:

$M =: 2 \ 2 \ \$ \ 3 \ 12 \ 11 \ 4$

Then, using our index-finding verb  $f$ , the flattening, amending and rebuilding is shown by:

$M$	$LL =: ,M$	$Z =: 10 \ f \ } \ LL$	$(\$M) \ \$ \ Z$
$\begin{array}{cc} 3 & 12 \\ 11 & 4 \end{array}$	$3 \ 12 \ 11 \ 4$	$3 \ 10 \ 10 \ 4$	$\begin{array}{cc} 3 & 10 \\ 10 & 4 \end{array}$

However, there is a better way. First note that our index-finding verb  $f$  takes as argument, not  $M$  but  $(LL =: , M)$ . Thus information about the original shape of  $M$  is not available to the index-finder  $f$ . In this example, this does not matter, but in general we may want the index-finding to depend upon both the shape and the values in  $M$ . It would be better if  $f$  took the whole of  $M$  as argument. In this case  $f$  must do its own flattening. Thus we redefine  $f$ :

```
f =: 4 : 0
Y. =. , Y.
(Y. > x.) # (i. # Y.)
)
```

$M$	$10 \ f \ M$
$\begin{array}{cc} 3 & 12 \\ 11 & 4 \end{array}$	$1 \ 2$

Now the index finder  $f$  takes an array as argument, and delivers indices into the flattened array, so-called "linear indices". The amending process, with this new  $f$ ,

is shown by:

M	( \$M ) \$ 10 ( 10 f M ) } ( , M )
3 12 11 4	3 10 10 4

Finally, provided  $\epsilon$  delivers linear indices, then  $(\})$  allows the last expression to be abbreviated as:

M	10 $\epsilon$ } M
3 12 11 4	3 10 10 4

## 6.4 Tree Indexing

So far we have looked at indexing into rectangular arrays. There is also a form of indexing into boxed structures, which we can picture as "trees" having branches and leaves. For example:

```
branch =: <
leaf   =: <

branch0 =: branch (leaf 'J S'), (leaf 'Bach')
branch1 =: branch (leaf 1), (leaf 2), (leaf 1777)
tree    =: branch0, branch1
tree
```

```
+-----+-----+
|+---+---+|+---+---+| | | | | | |
||J S|Bach||1|2|1777||
|+---+---+|+---+---+|
+-----+-----+
```

Then data can be fetched from the tree by specifying a path from the root. The path is a sequence of choices, given as left argument to the verb  $\{ : :$  (left-brace colon

colon,called "Fetch") The path 0 will fetch the first branch, while the path 0;1 fetches the second leaf of the first branch:

0 {:: tree	(0;1) {:: tree
<pre> +---+-----+  J  S Bach  +---+-----+ </pre>	Bach

The monadic form {:: tree is called the "Map" of tree. it has the same boxed structure as tree and shows the path to each leaf.

```

{:: tree
+-----+-----+-----+-----+
|+-----+-----+|+-----+-----+-----+| | | | | | | | | | | | | | | |
||+--+|+--+|+--+|+--+|+--+|+--+|
||0|0||0|1|||1|0||1|1||1|2||
||+--+|+--+|+--+|+--+|+--+|+--+|
|+-----+-----+|+-----+-----+-----+|
+-----+-----+-----+-----+

```

This is the end of Chapter 6.

---

Copyright © Roger Stokes 2001. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 4 Sep 01

# Chapter 7: Ranks

Recall that the rank of an array is its number of dimensions. A scalar is of rank 0, a list of numbers is of rank 1, a table of rank 2, and so on.

The subject of this chapter is how the ranks of arguments are taken into account when verbs are applied.

## 7.1 The Rank Conjunction

First, some terminology. An array can be regarded as being divided into "cells" in several different ways. Thus, a table such as

```
M =: 2 3 $ 'abcdef'
M
abc
def
```

may be regarded as being divided into 6 cells each of rank 0, or divided into 2 cells each of rank 1, or as being a single cell of rank 2. A cell of rank  $k$  will be called a  $k$ -cell.

### 7.1.1 Monadic Verbs

The box verb (monadic `<`) applies just once to the whole of the argument, to yield a single box, whatever the rank of the argument.

<code>L =: 2 3 4</code>	<code>&lt; L</code>	<code>M</code>	<code>&lt; M</code>
2 3 4	<pre>+-----+   2 3 4   +-----+</pre>	<pre>abc def</pre>	<pre>+----+   abc     def   +----+</pre>



However, we may choose to box each cell separately. Using the rank-conjunction " (double-quote), we write (`< " 0`) to box each scalar, that is, each 0-cell.

M	<code>&lt; " 0 M</code>	<code>&lt; " 1 M</code>	<code>&lt; " 2 M</code>
abc def	<pre> +-+--+  a b c  +-+--+  d e f  +-+--+ </pre>	<pre> +----+----+  abc def  +----+----+ </pre>	<pre> +----+  abc   def  +----+ </pre>

The general scheme is that in the expression (`u " k y`), the monadic verb `u` is applied separately to each `k`-cell of `y`.

We can define a verb to exhibit the `k`-cells of an array, each cell in its own box::

```
cells =: 4 : '< " x. y.'
```

M	<code>0 cells M</code>	<code>1 cells M</code>
abc def	<pre> +-+--+  a b c  +-+--+  d e f  +-+--+ </pre>	<pre> +----+----+  abc def  +----+----+ </pre>

## 7.1.2 Dyadic Verbs

Given a table, how do we multiply each row by a separate number? We multiply with the verb (`* " 1 0`) which can be understood as "multiply 1-cells by 0-cells". For example,

<code>x =: 2 2 \$ 0 1 2 3</code>	<code>y =: 2 3</code>	<code>x (* " 1 0) y</code>
----------------------------------	-----------------------	----------------------------

0 1 2 3	2 3	0 2 6 9
------------	-----	------------

The general scheme is that the expression

$x \ (u \ " \ (L,R)) \ y$

means: apply dyad  $u$  separately to each pair consisting of an L-cell from  $x$  and the corresponding R-cell from  $y$ .

To multiply each column by its own number, we combine each 1-cell of  $x$  with the solitary 1-cell of  $y$

$x$	$y$	$x \ (* \ " \ 1 \ 1) \ y$
0 1 2 3	2 3	0 3 4 9

## 7.2 Intrinsic Ranks

In J, every verb has what might be called a natural, or intrinsic, rank for its argument(s). Here are some examples to illustrate. For the first example, consider:

$*: \ 2$	$*: \ 2 \ 3 \ 4$
4	4 9 16

Here, the arithmetic function "square" naturally applies to a single number (a 0-cell). When a rank-1 array (a list) is supplied as argument, the function is applied separately to each 0-cell of the argument. In other words, the natural rank of (monadic)  $*:$  is 0.

For another example, there is a built-in verb  $\#.$  (hash dot called "Base Two"). Its argument is a bit-string (a list) representing a number in binary notation, and it

computes the value of that number. For example, 1 0 1 in binary is 5

```
#. 1 0 1
5
```

The verb `#.` applies naturally to a list of bits, that is, to a 1-cell. When a rank-2 array (a table) is supplied as argument, the verb is applied separately to each 1-cell, that is, to each row of the table.

t =: 3 3 \$ 1 0 1 0 0 1 0 1 1	#. t
1 0 1 0 0 1 0 1 1	5 1 3

Thus the natural rank of monadic `#.` is 1.

For a third example, as we have already seen, the monadic case of `<` applies just once to the whole of its argument, whatever the rank of its argument. The natural rank of `<` is thus an indefinitely large number, that is, infinity, denoted by `_`.

These examples showed monadic verbs. In the same way every dyadic verb will have two natural ranks, one for each argument. For example, the natural ranks of dyadic `+` are 0 0 since `+` takes a number (rank-0) on left and right.

In general, a verb has both a monadic and a dyadic case, and hence altogether 3 ranks, called its "intrinsic ranks". For any verb, its intrinsic ranks can be seen by applying the utility adverb `RANKS` (defined below), which gives the ranks in the order: monadic, left, right.

```
RANKS =: 1 : 'x. b. 0'
```

*: RANKS	#. RANKS	< RANKS
0 0 0	1 1 1	_ 0 0

For convenience, the rank conjunction can accept a right argument consisting of a single rank (for a monad) or two ranks (for a dyad) or three ranks (for an ambivalent verb).

One rank or two are automatically expanded to three as shown by:

(<"1) RANKS	(<"1 2) RANKS	(<"1 2 3) RANKS
1 1 1	2 1 2	1 2 3

## 7.3 Frames

Suppose  $u$  is a verb which sums all the numbers in a table. Evidently  $u$  has monadic rank 2.

```
u =: (+/) @: (+/) " 2
```

w =: 4 5 \$ 1	u w	u RANKS
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	20	2 2 2

Suppose a four-dimensional array  $A$  has shape 2 3 4 5.

```
A =: 2 3 4 5 $ i. 7
```

We can regard  $A$  as a 2-by-3 array of 2-cells, each cell being 4-by-5. Now consider computing  $(u\ A)$ . The verb  $u$ , being of rank 2, applies separately to each 2-cell, giving us a 2-by-3 array of results.

Each result is a scalar (because  $u$  produces scalars), and hence the overall result will

be 2 by 3 scalars.

<code>u A</code>	<code>\$ u A</code>
57 58 59 60 61 62	2 3

The shape 2 3 is called the "frame" of A with respect to its 2-cells, or its 2-frame for short. The k-frame of A is given by dropping the last k dimensions from the shape of A, or equivalently, as the shape of the array of k-cells of A.

```
frame =: 4 : '$ x. cells y.'
```

<code>\$ A</code>	<code>2 frame A</code>
2 3 4 5	2 3

In general, suppose that verb `u` has rank `k`, and from each `k`-cell it computes a cell of shape `s`. (The same `s`, we are supposing, for each cell). Then the shape of the overall result `(u A)` is: the `k`-frame of A followed by the shape `s`.

To demonstrate that this is the case, we can find `k` from `u`, as the first (monadic) rank of `u`:

```
k =: 0 { u RANKS
```

We can find the shape `s` by applying `u` to a typical `k`-cell of A, say the first.

```
s =: $ u 0 { > (, k cells A)
```

In this example, the shape `s` is an empty list, because `u` produces scalars.

<code>k</code>	<code>s</code>	<code>kfr =: k frame A</code>	<code>kfr, s</code>	<code>\$ u A</code>
----------------	----------------	-------------------------------	---------------------	---------------------

2		2 3	2 3	2 3
---	--	-----	-----	-----

Here we supposed that verb *u* gives the same-shaped result for each cell in its argument. This is not necessarily the case - see the section on "Reassembly of Results" below.

### 7.3.1 Agreement

A dyad has two intrinsic ranks, one for the left argument, one for the right. Suppose these ranks are *L* and *R* for a verb *u*.

When *u* is applied to arguments *x* and *y*, *u* is applied separately to each pair consisting of an *L*-cell from *x* and the corresponding *R*-cell from *y*. For example, suppose dyad *u* has ranks (0 1). It combines a 0-cell from *x* and a 1-cell from *y*.

```
u =: < @ , " 0 1
x =: 2 $ 'ab'
y =: 2 3 $ 'ABCDEF'
```

x	y	x u y
ab	ABC DEF	+-----+-----+   aABC   bDEF   +-----+-----+

Notice that here the 0-frame of *x* is the same as the 1-frame of *y*. These two frames are said to agree.

x	y	\$x	\$y	0 frame x	1 frame y
ab	ABC DEF	2	2 3	2	2

What if these two frames are not the same? They can still agree if one is a prefix of the other. That is, if one frame is the vector  $f$ , and the other frame can be written as  $(f, g)$  for some vector  $g$ . Here is an example. With

$$\begin{aligned} x &=: 2\ 3\ 2\ \$\ i.\ 12 \\ y &=: 2\ \quad\quad \$\ 0\ 1 \end{aligned}$$

and a dyad such as  $+$ , with ranks  $(0\ 0)$ , we are interested in the 0-frame of  $x$  and the 0-frame of  $y$ .

x	y	0 frame x	0 frame y	x+y
<div> 0 1 2 3 4 5 6 7 8 9 10 11 </div>	<div> 0 1 </div>	<div> 2 3 2 </div>	<div> 2 </div>	<div> 0 1 2 3 4 5 6 7 8 9 10 11 12 13 </div>

We see that the two frames are 2 and 2 3 2 and their difference  $g$  is therefore 3 2.

Here  $y$  has the shorter frame. Then each cell of  $y$  corresponds to, not just a single cell of  $x$ , but rather a 3 2-shaped array of cells. In such a case, a cell of  $y$  is automatically replicated to form a 3 2-shaped array of identical cells. In effect the shorter frame is made up to length, so as to agree with the longer. Here is an example. The expression  $(3\ 2\ \&\ \$)\ "0\ y$  means "a 3 by 2 replication of each 0-cell of  $y$ ".

x	y	yyy =: (3 2&\$)"0 y	x + yyy	x + y

0 1	0 1	0 0	0 1	0 1
2 3		0 0	2 3	2 3
4 5		0 0	4 5	4 5
6 7		1 1	7 8	7 8
8 9		1 1	9 10	9 10
10 11		1 1	11 12	11 12

What we have seen is the way in which a low-rank argument is automatically replicated to agree with a high-rank argument, which is possible provided one frame is a prefix of the other. Otherwise there will be a length error. The frames in question are determined by the intrinsic dyadic ranks of the verb.

The general scheme for automatically replicating one argument is: for arguments  $x$  and  $y$ , if  $u$  is a dyad with ranks  $L$  and  $R$ , and the  $L$ -frame of  $x$  is  $f, g$  and the  $R$ -frame of  $y$  is  $f$  (supposing  $y$  to have the shorter frame)

then  $(x \ u \ y)$  is computed as  $(x \ u \ (g\& \$))"R \ y)$

## 7.4 Reassembly of Results

We now look briefly at how the results of the computations on the separate cells are reassembled into the overall result. Suppose that the frame of application of a verb to its argument(s) is  $f$ , say. Then we can visualise each individual result as being stuffed into its place in the  $f$ -shaped framework of results. If each individual result-cell has the same shape,  $s$  say, then the shape of the overall result will be  $(f, s)$ . However, it is not necessarily the case that all the individual results are the same shape. For example, consider the following verb  $R$ , which takes a scalar  $y$  and produces a rank- $y$  result.

```
R =: (3 : '(y. $ y.) $ y.) " 0
```

R 1	R 2
1	2 2 2 2



When  $R$  is applied to an array, the overall result may be explained by envisaging each separate result being stuffed into its appropriate box in an  $f$ -shaped array of boxes. Then everything is unboxed all together. Note that it is the unboxing which supplies padding and extra dimensions if necessary to bring all cells to the same shape.

$(R\ 1) ; (R\ 2)$	$> (R\ 1) ; (R\ 2)$	$R\ 1\ 2$
<pre> +---+---+   1   2   2         2   2   +---+---+ </pre>	<pre> 1 0 0 0  2 2 2 2 </pre>	<pre> 1 0 0 0  2 2 2 2 </pre>

Consequently the shape of the overall result is given by  $(f, m)$  where  $m$  is the shape of the largest of the individual results.

## 7.5 More on the Rank Conjunction

### 7.5.1 Relative Cell Rank

The rank conjunction will accept a negative number for a rank. Thus the expression  $(u\ \<\_1\ y)$  means that  $u$  is to be applied to cells of rank 1 less than the rank of  $y$ , that is, to the items of  $y$ .

$x$	$\$x$	$<\_1\ x$	$<\_2\ x$
<pre> 0  1 2  3 4  5  6  7 8  9 10 11 </pre>	<pre> 2 3 2 </pre>	<pre> +---+---+---+   0 1   6  7     2 3   8  9     4 5  10 11   +---+---+---+ </pre>	<pre> +---+---+---+---+   0 1   2 3   4 5    +---+---+---+---+   6 7   8 9  10 11   +---+---+---+---+ </pre>

## 7.5.2 User-Defined Verbs

The rank conjunction has a special significance for user-defined verbs. The significance is that it allows us to define a verb considering only its "natural" rank: we ignore the possibility that it may be applied to higher-rank arguments. In other words, we can write a definition assuming the verb will be applied only to arguments of the natural rank. Afterwards, we can then put the finishing touch to our definition with the rank conjunction. Here are two examples.

The factorial of a number  $n$  is the product of the numbers from 1 to  $n$ . Hence (disregarding for the moment J's built-in verb `!`) we could define factorial straightforwardly as

```
f =: */ @: >: @: i.
```

because `i. n` gives the numbers  $0\ 1\ \dots\ (n-1)$ , and `>: i. n` gives  $1\ 2\ \dots\ n$ . We see:

f 2	f 3	f 4	f 5
2	6	24	120

Will `f` work as expected with a vector argument?

```
f 2 3
4 10 18
```

Evidently not. The reason is that `(f 2 3)` begins by computing `(i. 2 3)`, and `(i. 2 3)` does NOT mean `(i. 2)` followed by `(i. 3)`. The remedy is to specify that `f` applies separately to each scalar (rank-0 cell) in its argument:

```
f =: (* / @: (>: @: i.)) " 0

f 2 3 4 5
2 6 24 120
```

For a second example of the significance of the rank-conjunction we look at

explicitly defined verbs. The point being made here is, to repeat, that it is useful to be able to write a definition on the assumption that the argument is a certain rank say, a scalar, and only later deal with extending to arguments of any rank.

Two features of explicitly defined verbs are relevant. First, for any explicit verb, its intrinsic ranks are always assumed to be infinite. (This is because the J system does not look at the definition until the verb is executed.) Second, since the rank is infinite, the whole argument of an explicit verb is always treated as a single cell (or pair of cells for a dyad) and there is no automatic extension to deal with multiple cells.

For example, the absolute value of a number can be computed by the verb:

```
abs =: 3 : 'if. y. < 0 do. - y. else. y. end.'
```

abs 3	abs _3
3	3

We see that `abs`, being explicitly defined, has infinite rank:

```
abs RANKS
```

```
-- --
```

This means that if `abs` is applied to an array `y`, of any rank, it will be applied just once, and we can see from the definition that the result will be `y` or `-y`. There are no other possibilities. It is indeed the case that if `y` is a vector then `(y. < 0)` yields a vector result, but the expression `(if. y. < 0)` makes ONE decision. (This decision will in fact be based, not on the whole of `y > 0` but only on its leading item. See Ch X for more details). Hence if the argument contains both positives and negatives, this decision must be wrong for some parts of the argument.

```
abs 3 _3
3 _3
```

Hence with `abs` defined as above, it is important to limit its application to scalars.

Thus a better definition for abs would be:

```
abs =:(3 : 'if. y. < 0 do. -y. else. y. end.')"0  
  
abs 3 _3  
3 3
```

This brings us to the end of Chapter 7.

---

Copyright © Roger Stokes 2001. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 28 Jan 2002

# Chapter 8: Composing Verbs

This chapter is concerned with operators which combine two verbs to produce new composite verbs.

## 8.1 Composition of Monad and Monad

Recall the composition operator `@:` from [Chapter 03 p3](#). Given verbs `sum` and `square` we can define a composite verb, `sum-of-the-squares`.

```
sum      =: +/
square   =: *:
```

<code>sumsq =: sum @: square</code>	<code>sumsq 3 4</code>
<code>sum@:square</code>	25

The general scheme is that if `f` and `g` are monads then

`(f @: g) y` means `f (g y)`

Note in particular that `f` is applied to the whole result `(g y)`. To illustrate, suppose `g` applies separately to each row of a table, so we have:

```
g =: sum " 1
f =: <
```

<code>y =: 2 2 \$ 1 2 3 4</code>	<code>g y</code>	<code>f g y</code>	<code>(f @: g) y</code>

1 2 3 4	3 7	+----+   3 7   +----+	+----+   3 7   +----+
------------	-----	-----------------------------	-----------------------------

We have just seen the most basic of kind of composition. Now we look at some variations.

## 8.2 Composition: Monad And Dyad

If  $f$  is a monad and  $g$  is a dyad, then  $(f @: g)$  is a dyadic verb such that  
 $x (f @: g) y$  means  $f (x g y)$

For example, the sum of the product of two vectors  $x$  and  $y$  is called the "scalar product".

`sp =: +/ @: *`

<code>x =: 1 2</code>	<code>y =: 2 3</code>	<code>x * y</code>	<code>+/(x * y)</code>	<code>x sp y</code>
1 2	2 3	2 6	8	8

The last example showed that, in the expression  $(x (f @: g) y)$  the verb  $f$  is applied once to the whole of  $(x g y)$

## 8.3 Composition: Dyad And Monad

The conjunction `&:` (ampersand colon, called "Appose") will compose dyad  $f$  and monad  $g$ . The scheme is:

$x (f &: g) y$  means  $(g x) f (g y)$

For example, we can test whether two lists are equal in length, with the verb `(= &:`

```
#)
  eqlen =: = &: #
```

x	y	#x	#y	(#x) = (#y)	x eqlen y
1 2	2 3	2	2	1	1

Here  $f$  is applied once to the whole of  $(g\ x)$  and  $(g\ y)$ .

## 8.4 Ambivalent Compositions

To review, we have seen three different schemes for composition. These are:

$$\begin{aligned}
 (f\ @: g)\ y &= f\ (g\ y) \\
 x\ (f\ @: g)\ y &= f\ (x\ g\ y) \\
 x\ (f\ \&: g)\ y &= (g\ x)\ f\ (g\ y)
 \end{aligned}$$

There is a fourth scheme,

$$(f\ \&: g)\ y = f\ (g\ y)$$

which is, evidently, the same as the first. This apparent duplication is useful for the following reason. Suppose verb  $g$  is ambivalent, that is, has both a monadic and dyadic case. It follows from the first two schemes that the composition  $(f\ @: g)$  is also ambivalent. Similarly, if verb  $f$  is ambivalent, it follows from the third and fourth schemes that  $(f\ \&: g)$  is ambivalent.

To illustrate, let  $g$  be the ambivalent built-in verb  $(|.)$  with  $(|.\ y)$  being the reverse of  $y$  and  $x\ |.\ y$  being the rotation of  $y$  by  $x$  places.

y =: 'abcdef'	(< @:  .) y	1 (< @:  .) y
---------------	-------------	---------------

abcdef	<pre> +-----+  fedcba  +-----+ </pre>	<pre> +-----+  bcdefa  +-----+ </pre>
--------	---------------------------------------	---------------------------------------

For an example of ambivalent  $(f \ \& \ g)$ , let  $f$  be the verb  $\%$  - reciprocal or divide.

$\% \ * \colon 2$	$\% \ \& \colon \ * \colon 2$	$(\ * \colon 3)\%(\ * \colon 2)$	$3 \ \% \ \& \colon \ * \colon 2$
0.25	0.25	2.25	2.25

## 8.5 More on Composition: Monad Tracking Monad

The conjunction  $@$  is a variation of the  $@ \colon$  conjunction. Here is an example to show the difference between  $(f \ @ \colon g)$  and  $(f \ @ \ g)$ .

`y =: 2 2 $ 0 1 2 3`

y	f	g	$(f \ @ \colon g) \ y$	$(f \ @ \ g) \ y$
<pre> 0 1 2 3 </pre>	<	sum"1	<pre> +----+   1 5   +----+ </pre>	<pre> +----+   1   5   +----+ </pre>

We see that with  $(f \ @ \colon g)$  verb  $f$  is applied once. However, with  $(f@g)$ , for each separate application of  $g$  there is a corresponding application of  $f$ . We could say that applications of  $f$  track the applications of  $g$ .

Suppose that the monadic rank of  $g$  is  $G$ . Then  $(f \ @ \ g)$  means  $(f \ @ \colon g)$  applied separately to each  $G$ -cell, that is,  $(f \ @ \colon g) \ "G$ .

`RANKS =: 1 : 'x. b. 0'`



<code>G =: 0 { g RANKS</code>	<code>(f @ g) y</code>	<code>(f @: g) "G y</code>
1	<pre> +---+   1   5   +---+ </pre>	<pre> +---+   1   5   +---+ </pre>

and so the general scheme is:

`(f @ g) y` means `(f @: g) " G y`

There is also the `&` operator. For reasons of symmetry, as with the ambivalent functions mentioned above, `(f&g) y` means the same as `(f@g) y`.

## 8.6 Composition: Monad Tracking Dyad

Next we look at the composition `(f @ g)` for a dyadic `g`. Suppose `f` and `g` are defined by:

```

f =: <
g =: |. " 0 1 NB. dyadic

```

Here `x g y` means: rotate vectors in `y` by corresponding scalars in `x`. For example:

<code>x=: 1 2</code>	<code>y=: 2 3 \$ 'abcdef'</code>	<code>x g y</code>
1 2	abc def	bca fde

Here now is an example to show the difference between `f @: g` and `f @ g`

<code>f (x g y)</code>	<code>x (f @: g) y</code>	<code>x (f @ g) y</code>
------------------------	---------------------------	--------------------------

<pre> +----+  bca   fde  +----+ </pre>	<pre> +----+  bca   fde  +----+ </pre>	<pre> +----+----+  bca fde  +----+----+ </pre>
--	--	--

We see that with  $(f @: g)$  verb  $f$  is applied once. With  $(f@g)$ , for each separate application of  $g$  there is a corresponding application of  $f$ .

Suppose that the left and right ranks of dyad  $g$  are  $L$  and  $R$ . Then  $(f @ g)$  means  $(f @: g)$  applied separately to each pair of an  $L$ -cell from  $x$  and corresponding  $R$ -cell from  $y$ . That is,  $(f@g)$  means  $(f @: g) "G$  where  $G = L, R$ .

$G =: 1\ 2\ \{ \ g\ RANKS$	$x\ (f @:g) " G\ y$	$x\ (f @ g) y$
0 1	<pre> +----+----+  bca fde  +----+----+ </pre>	<pre> +----+----+  bca fde  +----+----+ </pre>

The scheme is:

$$x\ (f@g) y = x\ (f@:g) " G y$$

# 8.7 Composition: Dyad Tracking Monad

here we look at the composition  $(f \& g)$  for dyadic  $f$ .

Suppose  $g$  is the "Square" function, and  $f$  is the "comma" function which joins two lists.

```

f =: ,
g =: *:

```

$x =: 1\ 2$	$y =: 3\ 4$	$g\ x$	$g\ y$
-------------	-------------	--------	--------

1 2	3 4	1 4	9 16
-----	-----	-----	------

Here now is an example to show the difference between  $(f \&: g)$  and  $(f \& g)$

$(g \ x) \ f \ (g \ y)$	$x \ (f \&: g) \ y$	$x \ (f \& g) \ y$
1 4 9 16	1 4 9 16	1 9 4 16

We see that in  $(f \&: g)$  the verb  $f$  is applied just once, to join the two lists of squares. By contrast, in  $(f \& g)$  each separate pair of squares is combined with a separate application of  $f$

The scheme is that

$x \ (f \& g) \ y$  means  $(g \ x) \ (f \ " \ G, G) \ (g \ y)$

where  $G$  is the monadic rank of  $g$ . Here  $f$  is applied separately to each combination of a  $G$ -cell from  $x$  and a corresponding  $G$ -cell from  $y$ . To illustrate:

$G =: 0 \ \{ \ g \ \text{RANKS}$	$(g \ x) \ (f \ " \ (G, G)) \ (g \ y)$	$x \ (f \& g) \ y$
0	1 9 4 16	1 9 4 16

## 8.8 Summary

Here is a summary of the 8 cases we have looked at so far.

$@: \quad (f \ @: g) \ y = f \ (g \ y)$   
 $@: \quad x \ (f \ @: g) \ y = f \ (x \ g \ y)$   
  
 $\&: \quad (f \ \&: g) \ y = f \ (g \ y)$   
 $\&: \quad x \ (f \ \&: g) \ y = (g \ x) \ f \ (g \ y)$

$$\begin{array}{lcl}
@ & (f @ g) & y = (f @: g) " G y \\
@ & x (f @ g) & y = x (f @: g) " LR y \\
\\
& (f \& g) & y = (f @: g) " G y \\
& x (f \& g) & y = (g x) (f " (G,G)) (g y)
\end{array}$$

where  $G$  is the monadic rank of  $g$  and  $LR$  is the vector of left and right ranks of  $g$ .

## 8.9 Inverses

The "Square" verb,  $(*:)$ , is said to be the inverse of the "Square-root" verb  $(\%:)$ . The reciprocal verb is its own inverse.

$*: 2$	$\%: 4$	$\% 4$	$\% 0.25$
4	2	0.25	4

Many verbs in J have inverses. The adverb  $(^: _1)$  produces the inverse verb of its argument verb. Let us call this adverb `INV`. `INV` produces "Square-root" from "Square":

<code>INV =: ^: _1</code>	$\%: 16$	$*: INV 16$
$^: _1$	4	4

`INV` can automatically find inverses, not only of built-in verbs, but of user-defined verbs such as compositions. For example, the inverse of  $(1 + \text{the square-root})$  of  $y$  is  $(\text{the square of } 1 \text{ minus})y$ .

<code>foo =: (1&amp;+) @: %:</code>	<code>foo 16</code>	<code>foo INV 5</code>
$(1\&+ )@: \%:$	5	16

## 8.10 Composition: Verb Under Verb

We now look at composition with the conjunction  $\&.$  (ampersand dot, called "Under"). The idea is that the composition " $f$  Under  $g$ " means: apply  $g$ , then  $f$ , then the inverse of  $g$ .

For an example, suppose first that  $f$  is the verb which rounds a number to the nearest integer:

$f =: < . @ (0.5 \& +)$	$f \ 1.2 \ 1.8$
$< . @ ( (0.5) \& + )$	$1 \ 2$

A number can be rounded to the nearest 10, say, by dividing by 10, rounding to nearest integer, then multiplying by 10 again.

Let  $g$  be division by 10, and then  $(g \text{ INV})$  will be the inverse, multiplication by 10.

$g =: \% \& 10$

$g \ 28$	$f \ g \ 28$	$(g \text{ INV}) \ f \ g \ 28$	$f \ \& . \ g \ 28$
$2.8$	$3$	$30$	$30$

The general scheme is that

$(f \ \& . \ g) \ y$  means  $(g \text{ INV}) \ f \ g \ y$

This is the end of Chapter 8.

provision is also reproduced.

last updated 10 March 00

# Chapter 9: Trains of Verbs

In this chapter we continue the topic of trains of verbs begun in Chapter 3. Recall that a train is an isolated sequence of functions, written one after the other, such as  $(+ * -)$ .

## Preparations

It will be convenient to have a few definitions ready to hand for the examples to come.

```
x    =: 1 2 3 4  NB. a list of numbers
sum  =: + /       NB. verb: sum of a list
min  =: <. /      NB. verb: smallest item of a list
max  =: >. /      NB. verb: largest item of a list
```

## 9.1 Review: Monadic Hooks and Forks

Recall from Chapter 3 the monadic hook, with the scheme:

$(f\ g)\ y$  means  $y\ f\ (g\ y)$

Here is an example, as a brief reminder: a whole number is equal to its floor:

$y =: 2.1\ 3$	$<.\ y$	$y = <.\ y$	$(= <.)\ y$
2.1 3	2 3	0 1	0 1

Recall also the monadic fork, with the scheme:

$(f\ g\ h)\ y$  means  $(f\ y)\ g\ (h\ y)$

For example: the mean of a list of numbers is the sum divided by the number-of-

items:

```
mean =: sum % #
```

x	sum x	# x	(sum x) % (# x)	mean x
1 2 3 4	10	4	2.5	2.5

Now we look at some further variations.

## 9.2 Dyadic Hooks

3 hours and 15 minutes is 3.25 hours. A verb `hr`, such that `(3 hr 15)` is 3.25, can be written as a hook. We want `x hr y` to be `x + (y%60)` and so the hook is:

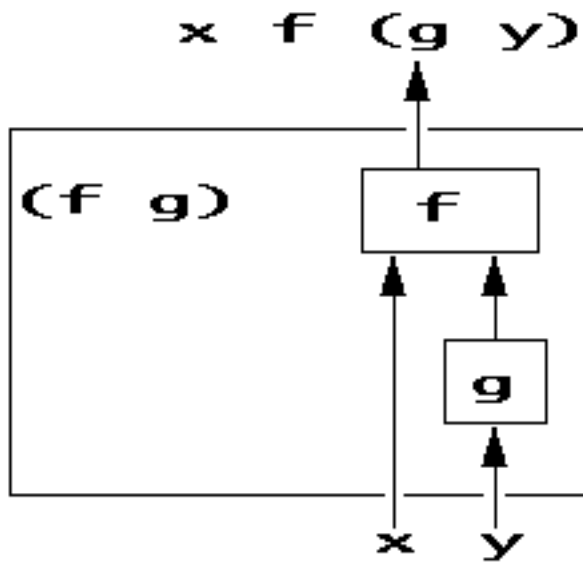
```
hr =: + (%&60)
3 hr 15
3.25
```

The scheme for dyadic hook is:

```
x (f g) y means x f (g y)
```

with the diagram:





## 9.3 Dyadic Forks

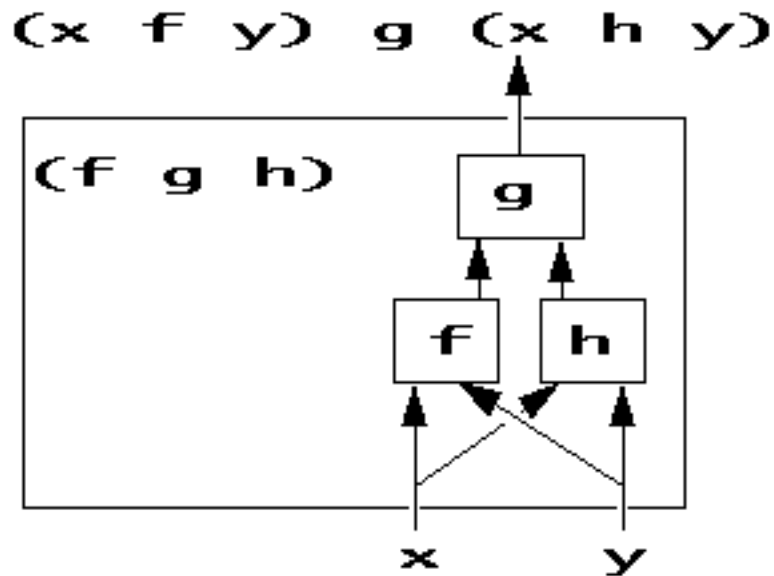
Suppose we say that the expression "10 plus or minus 2" is to mean the list 12 8. A verb to compute  $x$  plus-or-minus  $y$  can be written as the fork  $(+, -)$ :

(10+2) , (10-2)	10 (+, -) 2
12 8	12 8

The scheme for a dyadic fork is:

$x (f g h) y$  means  $(x f y) g (x h y)$

Here is a diagram for this scheme:



## 9.4 Review

There are four basic patterns of verb-trains. It may help to fix them in the memory by recalling these four verbs:

mean	<code>=: sum % #</code>	NB. monadic fork
plusminus	<code>=: + , -</code>	NB. dyadic fork
wholenumber	<code>=: = &lt;.</code>	NB. monadic hook
hr	<code>=: + (%&amp;60)</code>	NB. dyadic hook

## 9.5 Longer Trains

Now we begin to look at ways in which to broaden the class of functions which can be defined as trains.

In general a train of any length can be analysed into hooks and forks. For a train of 4 verbs, `e f g h`, the scheme is that

`e f g h` means `e (f g h)`

that is, a 4-train `(e f g h)` is a hook, where the first verb is `e` and the second is the fork `(f g h)`. For example, if `y` a list of numbers:

```
y =: 2 3 4
```

then the "norm" of  $y$  is  $(y - \text{mean } y)$ , where  $\text{mean}$  is defined above as  $(\text{sum } \% \#)$ . We see that the following expressions for the norm of  $y$  are all equivalent:

```
y - mean y
_1 0 1
```

```
(- mean) y      NB. as a hook
_1 0 1
```

```
(- (sum % #)) y  NB. by definition of mean
_1 0 1
```

```
(- sum % #) y    NB. as 4-train
_1 0 1
```

A certain amount of artistic judgement is called for with long trains. This last formulation as the 4-train  $(- \text{sum } \% \#)$  does not bring out as clearly as it might that the key idea is subtracting the mean. The formulation  $(- \text{mean})$  is clearer.

For a train of 5 verbs  $d \ e \ f \ g \ h$  the scheme is:

$d \ e \ f \ g \ h$  means  $d \ e \ (f \ g \ h)$

That is, a 5-train  $(d \ e \ f \ g \ h)$  is a fork with first verb  $d$ , second verb  $e$  and third verb the fork  $(f \ g \ h)$  For example, if we write a calendar date in the form day month year:

```
date =: 28 2 1999
```

and define verbs to extract the day month and year separately:

```
Da =: 0 & {
Mo =: 1 & {
Yr =: 2 & {
```

the date can be presented in different ways by 5-trains:

(Da , Mo , Yr) date	(Mo ; Da ; Yr) date
28 2 1999	<pre> +---+-----+   2   28   1999   +---+-----+ </pre>

The general scheme for a train of verbs (a b c ...) depends upon whether the number of verbs is even or odd:

even: (a b c ...) means hook (a (b c ...))

odd : (a b c ...) means fork (a b (c ...))

## 9.6 Identity Functions

There is a verb [ (left bracket) which gives a result identical to its argument.

[ 99	[ 'a b c'
99	a b c

There is a dyadic case, and also a similar verb ]. Altogether we have these schemes

[ y means y

x [ y means x

] y means y

x ] y means y

[ 3	2 [ 3	] 3	2 ] 3
-----	-------	-----	-------

3	2	3	3
---	---	---	---

The expression `(+ % ])` is a fork; for arguments `x` and `y` it computes

`(x+y) % (x ] y)`

that is, `(x+y) % y`

<code>2 ] 3</code>	<code>(2 + 3) % (2 ] 3)</code>	<code>2 (+ % ]) 3</code>
3	1.66667	1.66667

Another use for the identity function `[` is to cause the result of an assignment to be displayed. The expression `foo =: 42` is an assignment while the expression `[ foo =: 42` is not: it merely contains an assignment.

```
foo =: 42      NB.  nothing displayed
[ foo =: 42
42
```

Yet another use for the `[` verb is to allow several assignments to be combined on one line.

<code>a =: 3 [ b =: 4 [ c =: 5</code>	<code>a,b,c</code>
3	3 4 5

Since `[` is a verb, its arguments must be nouns, (that is, not functions). Hence the assignments combined with `[` must all evaluate to nouns.

### 9.6.1 Example: Hook as Abbreviation

The monadic hook `(g h)` is an abbreviation for the monadic fork `([ g h)`. To

demonstrate, suppose we have:

```
g =: ,
h =: *:
y =: 3
```

Then each of the following expressions is equivalent.

```
([ g h) y      NB. a fork
3 9
([ y) g (h y)   NB. by defn of fork
3 9
y g (h y)       NB. by defn of [
3 9
(g h) y         NB. by defn of hook
3 9
```

## 9.6.2 Example: Left Hook

Recall that the monadic hook has the general scheme

$$(f\ g)\ y = y\ f\ (g\ y)$$

How can we write, as a train, a function with the scheme

$$( \ ? \ )\ y = (f\ y)\ g\ y$$

There are two possibilities. One is the fork  $(f\ g\ ])$ :

```
f =: *:
g =: ,

(f g ]) y      NB. a fork
9 3
(f y) g ([ y)   NB. by meaning of fork
9 3
(f y) g y       NB. by meaning of ]
9 3
```

For another possibility, recall the  $\sim$  adverb with its scheme  $(x\ f\sim\ y) = (y\ f\ x)$ .

Our train can be written as the hook  $(g \sim f)$ .

```

(g ~ f) y      NB. a hook
9 3
y (g ~) (f y)  NB. by meaning of hook
9 3
(f y) g y      NB. by meaning of ~
9 3

```

### 9.6.3 Example: Dyad

There is a sense in which `[` and `]` can be regarded as standing for left and right arguments.

```

f =: 'f' & ,
g =: 'g' & ,

```

<code>foo =: (f @: [) , (g @: ])</code>	<code>'a' foo 'b'</code>
<code>(f@:[) , (g@:])</code>	<code>fagb</code>

## 9.7 The Capped Fork

The class of functions which can be written as unbroken trains can be widened with the aid of the "Cap" verb `[ :` (leftbracket colon)

The scheme is: for verbs `f` and `g`, the fork `[ : f g` means `f @: g`. For example, let

```

f =: 'f' & ,
g =: 'g' & ,
y =: 'y'

```

then `[ :` is illustrated by:

$f \ g \ y$	$(f \ @: \ g) \ y$	$([: \ f \ g) \ y$
fgy	fgy	fgy

Notice how the sequence of three verbs  $([: \ f \ g)$  looks like a fork, but with this "capped fork" it is the MONADIC case of the middle verb  $f$  which is applied.

The  $[:$  verb is valid ONLY as the left-hand verb of a fork. It has no other purpose: as a verb it has an empty domain, that is, it cannot be applied to any argument. Its usefulness lies in building long trains. Suppose for example that:

$h =: 'h' \&$ ,

then the expression  $(f \ , \ [: \ g \ h)$  is a 5-train which denotes a verb:

$(f \ , \ [: \ g \ h) \ y$                       NB. a 5-train  
fyghy

$(f \ y) \ , \ (([: \ g \ h) \ y)$     NB. by meaning of 5-train  
fyghy

$(f \ y) \ , \ (g \ @: \ h \ y)$         NB. by meaning of  $[:$   
fyghy

$(f \ y) \ , \ (g \ h \ y)$             NB. by meaning of  $@:$   
fyghy

$'fy' \ , \ 'ghy'$                     NB. by meaning of  $f \ g \ h$   
fyghy

## 9.8 Constant Functions

Here we continue looking at ways of broadening the class of functions that we can write as trains of verbs. There is a built-in verb  $0:$  (zero colon) which delivers a value of zero regardless of its argument. There is a monadic and a dyadic case:



0: 99	0: 2 3 4	0: 'hello'	88 0: 99
0	0	0	0

As well as 0: there are similar functions 1: 2: 3: and so on up to 9: and also the negative values: \_9: to \_1:

1: 2 3 4	_3: 'hello'
1	_3

0: is said to be a constant function, because its result is constant. Constant functions are useful because they can occur in trains at places where we want a constant but must write a verb, (because trains of verbs, naturally, contain only verbs).

For example, a verb to test whether its argument is negative (less than zero) can be written as (< & 0) but alternatively it can be written as a hook:

negative =: < 0:

x =: _1 0 2	0: x	x < (0: x)	negative x
_1 0 2	0	1 0 0	1 0 0

## 9.9 The "Constant" Conjunction

The constant functions \_9: to 9: offer more choices for ways of defining trains. Nevertheless they are limited to single-digit scalar constants. We look now at a more general way of writing constant functions. Suppose that *k* is the constant in question:

`k =: 'hello'`

An explicit verb written as `(3 : 'k')` will give a constant result of `k`:

<code>k</code>	<code>(3 : 'k') 1</code>	<code>(3 : 'k') 1 2</code>
hello	hello	hello

Since `k` is explicit, its rank is infinite: to apply it separately to scalars we need to specify a rank `R` of 0:

<code>k</code>	<code>R =: 0</code>	<code>((3 : 'k') " R) 1 2</code>
hello	0	hello hello

The expression `((3 : 'k') " R)` can be abbreviated as `(k " R)` with the aid of the Constant conjunction `"` (double quote)

<code>k</code>	<code>R</code>	<code>((3 : 'k') " R) 1 2</code>	<code>'hello' " R 1 2</code>
hello	0	hello hello	hello hello

Note that if `k` is a noun, then the verb `(k"R)` means: the constant value `k` produced for each rank-`R` cell of the argument. By contrast, if `v` is a verb, then the verb `(v"R)` means: the verb `v` applied to each rank-`R` cell of the argument.

The general scheme can be represented as:

`k " R` means `(3 : 'k') " R`

This is the end of Chapter 9.

---

Copyright © Roger Stokes 1999. This material may be freely reproduced, provided that this copyright notice and provision is also reproduced.

last updated 10 September 1999

# Chapter 10: Conditional and Other Forms

Tacit verbs were introduced in [Chapter 03 p3](#). Continuing this theme, in [Chapter 08 p8](#) we looked at the use of composition-operators and in [Chapter 09 p9](#) at trains of verbs.

The plan for this chapter is to look at further ways of defining verbs tacitly:

- Conditional forms
- Recursive forms
- Iterative forms
- Generating tacit definitions from explicit definitions

## 10.1 Conditional Forms

Think of a number (some positive whole number). If it is odd, multiply by 3 and then add 1. Otherwise, halve the number you thought of. This procedure computes from 1 the new number 4, and from 4 the new number 2.

To write a function to compute a new number according to this procedure, we start with three verbs, say `halve` to halve, `mult` to multiply-and-add, and `odd` to test for an odd number:

```
halve =: -:
mult  =: 1: + (* 3:)
odd   =: 2 & |
```

halve 6	mult 6	odd 6
3	19	0

Now our procedure for a new number can be written as an explicit verb:

```
NEW =: 3 : 0
if. odd y. do. mult y.
else.      halve y.
end.
)
```

and equivalently as a tacit verb:

```
new =: (halve ` mult) @. odd
```

NEW 1	new 1
4	4

In the definition of `new`, the symbol ``` (backquote) is called the "Tie" conjunction. It ties together `halve` and `mult` to make a list of two verbs. (Such a list is called a "gerund" and we look at more uses of gerunds in [Chapter 14 p14](#)).

In evaluating `new y` the value of `odd y` is used to index the list `(halve`mult)`. Then the selected verb is applied to `y`. That is, `halve y` or `mult y` is computed accordingly as `odd y` is 0 or 1.

In this example, we have two cases to consider: the argument is odd or not. In general, there may be several cases. The general scheme is, if `u0`, `u1`, ... `un` are verbs, and `t` is a verb computing an integer in the range `0 .. n`, then the verb:

```
foo =: u0 ` u1 ` ... ` un @. t
```

can be modelled by the explicit verb:

```
FOO =: 3 : 0
if.    (t y.) = 0 do. u0 y.
elseif. (t y.) = 1 do. u1 y.
...
elseif. (t y.) = n do. un y.
end.
)
```

That is, verb `t` tests the argument `y` and then `u0` or `u1` or ... is applied to `y` according to whether `(t y)` is 0 or 1 or .... head3 'Example with 3 Cases' pp 0  
 Suppose that, each month, a bank pays or charges interest according to the balances of customers' accounts as follows. There are three cases:

- If the balance is \$100 or more, the bank pays interest of 0.5%
- If the balance is negative, the bank charges interest at 2%.
- Otherwise the bank pays or charges nothing.

Three verbs, one for each of the three cases, could be:

```
pi =: * & 1.005      NB. pay interest
ci =: * & 1.02       NB. charge interest
dn =: * & 1          NB. do nothing
```

pi 1000	ci _100	dn 50
1005	_102	50

Now we want a verb to compute, from a given balance, 0 or 1 or 2 as the appropriate index into a list of three verbs containing `pi`, `ci` and `dn`. A somewhat heavy-handed but general method is to write verbs to recognise each of the three cases, tried in order:

```
rpi =: >: & 100 NB. equal to or greater than 100
rci =: < & 0    NB. otherwise, less than 0
rdn =: 1:       NB. otherwise
```

and combine them into a case-recognising verb as follows:

```
recognise =: (i. & 1 ) @: (rpi, rci, rdn)

recognise " 0 (1000 _100 50)
0 1 2
```

Now we can put everything together: the processing of a balance can be represented by the verb `PB` say:

```
PB =: pi ` ci ` dn @. recognise
```

PB 1000	PB _100	PB 50
1005	_102	50

The argument of `PB` is expected to fall under exactly one of the three possible cases, in order to select exactly one verb (`pi` or `ci` or `dn`) to apply to the whole argument.

Hence, if the argument is a list such that different items fall under different cases, then the `PB` function must be applied separately to each item of its argument.

PB 99 100	(PB "0 ) 99 100
100.98 102	99 100.5

## 10.2 Recursion

To compute the sum of a list of numbers, we have seen the verb `+/` but let us look at another way of defining a summing verb.

The sum of an empty list of numbers is zero, and otherwise the sum is the first item plus the sum of the remaining items. If we define three verbs, to test for an empty list, to take the first item and to take the remaining items:

```
empty =: # = 0:
first  =: {.
rest   =: }.
```

then the two cases to consider are:

- an empty list, in which case we apply the 0: function to return zero
- a non-empty list, in which case we want the first plus the sum of the rest:

```
Sum =: (first + Sum @ rest) ` 0: @. empty
```

```
Sum 1 1 2
```

4

Here we see that the verb "Sum" recurs in its own definition and so the definition is said to be recursive.

In such a recursive definition, the name which recurs can be written as \$: (dollar colon), meaning "this function". This enables us to write a recursive function as an expression, without assigning a name. Here is the "Sum" function as an expression:

```
((first + $: @ rest) ` 0: @. empty) 1 2 3
```

6

## 10.2.1 Ackermann's Function

Ackermann's function is celebrated for being extremely recursive. Textbooks show it in a form something like this explicit definition of a dyad:

```
Ack =: 4 : 0
if.      x. = 0 do. y. + 1
elseif.  y. = 0 do. (x. - 1) Ack 1
elseif.  1      do. (x. - 1) Ack (x. Ack y. -1)
end.
)
```

```
2 Ack 3
```

9

A tacit version is due to Roger Hui (Vector, Vol 9 No 2, Oct 1992, page 142):

```
ack =: c1 ` c1 ` c2 ` c3 @. (#. @(&*))
```

```
c1 =: >:@] NB. 1 + y
```

```
c2 =: <:@[ ack 1: NB. (x-1) ack 1
```

```
c3 =: <:@[ ack [ack <:@] NB. (x -1) ack x ack y -1
```



```

2 ack 3
9

```

Notice that in the line defining `c2` the function is referred to as `ack`, not as `$:`, because here `$:` would mean `c2`.

Here is yet another version. The tacit version can be made to look a little more conventional by first defining `x` and `y` as the verbs `[` and `]`. Also, we test for only one case on a line.

```

x =: [
y =: ]

ACK =: A1 ` (y + 1:) @. (x = 0:)
A1 =: A2 ` ((x - 1:) ACK 1:) @. (y = 0:)
A2 =: (x - 1:) ACK (x ACK y - 1:)

2 ACK 3
9

```

## 10.3 Iteration

### 10.3.1 The Power Conjunction

Think of a number, double it, double that result, double again. The result of three doublings is eight times the original number. The built-in verb `+:` is "double", and the verb "three doublings" can be written using the "Power" conjunction (`^:`) as `+: ^: 3`

```
+: ^: 3
```

<code>+: +: +: 1</code>	<code>(+: ^: 3 ) 1</code>
8	8

The general scheme is that for a verb `f` and an integer `n`

$$(f \ ^: n) y \text{ means } f \ f \ f \ \dots \ f \ f \ f \ f \ y$$

<--- n f's --->

Notice that  $f^{\wedge} 0 y$  is just  $y$  and then  $f^{\wedge} 1 y$  is  $f y$ . For example, recall the new verb "halve or multiply-by-3-and-add-1 if odd".

<code>(new ^: 0) 6</code>	<code>(new ^: 1) 6</code>	<code>new 6</code>
6	3	3

With the Power conjunction we can generate a series by applying `new` 0 times, once, twice and so on, starting with 6 for example

```
(new ^: 0 1 2 3 4 5 6 ) 6
6 3 10 5 16 8 4
```

## 10.3.2 Iterating Until No Change

The expression  $f^{\wedge} \_$  where the Power conjunction is given a right argument of infinity ( $\_$ ), is a verb where  $f$  is applied until a result is reached which is the same as the previous result. The scheme is:

```
f ^: _ y      means
                r such that r = f f ... f f y
                and r = f r
```

Here is an example. Suppose function `P` is defined as:

```
P =: 3 : '2.8 * y. * (1 - y.)'
```

Then if we repeatedly apply the function to an argument in the neighbourhood of 0.5, after 20 or so iterations the result will settle on a value of about 0.643

```
(P ^: 0 1 2 3      19 20 _) 0.5
0.5 0.7 0.588 0.6783 0.6439 0.642 0.6429
```

and this value,  $r$  say, is called a fixed point of  $P$  because  $r = P\ r$

<code>r =: (P ^: _) 0.5</code>	<code>P r</code>
0.6429	0.6429

### 10.3.3 Iterating While

The right argument of the "Power" conjunction can be a verb which computes the number of iterations to be performed. The scheme is:

`(f ^: g) y` means `f ^: (g y) y`

If `g y` computes 0 or 1, then `f` will be applied 0 times or 1 time: For example, here is a verb which halves an even number and leaves an odd number alone:

```
halve =: -:
even   =: 0: = 2 & |
```

<code>foo =: halve ^: even</code>	<code>(foo " 0) 1 2 3 4</code>
<code>halve^:even</code>	1 1 3 2

Now consider the function

```
w =: (halve ^: even) ^: _
```

This means "halve if even, and keep doing this so long as the result keeps changing".

```
w (3 * 16)
```

The scheme is that a function written  $(f \wedge g \wedge \_)$  can be modelled by an explicit definition:

```

model =: 3 : 0
while. (g y.)
  do. y. =. f y.
end.
y.
)

```

```

f =: halve
g =: even

```

$(f \wedge g \wedge \_) \ 3 \ * \ 16$	model 3*16
3	3

### 10.3.4 Iterating A Dyadic Verb

Adding 3, twice, to 0 gives 6

```

((3&+) ^: 2) 0
6

```

This expression can be abbreviated as:

```

3 (+ ^: 2) 0
6

```

The given left argument (3) is fixed at the outset, so the iterated verb is the monad  $3\&+$ . The general scheme is:

$x \ (u \wedge w) \ y$  means  $((x\&u) \wedge w) \ y$

where  $w$  is a noun or verb.

## 10.4 Generating Tacit Verbs from Explicit

Suppose that `e` is a verb, defined explicitly as follows:

```
e =: 3 : ' (+/ y.) % # y.'
```

The right argument of the colon conjunction we can call the "body". Then a tacit verb, `t` say, equivalent to `e`, can be produced by writing `13 :` instead of `3 :` with the same body.

```
t =: 13 : ' (+/ y.) % # y.'
```

e	t	e 1 2 3	t 1 2 3
3 : ' (+/ y.) % # y.'	+/ % #	2	2

Here now is an example of an explicit dyad.

```
ed =: 4 : 'y. % x.'
```

The equivalent tacit dyad can be generated by writing `13 :` rather than `4 :` with the same body.

```
td =: 13 : 'y. % x.'
```

ed	td	2 ed 6	2 td 6
4 : 'y. % x.'	] % [	3	3

We can conclude that if we write `13 : body`, and `body` contains `y.` (but not `x.`) then the result is a tacit verb of which the monadic case is equivalent to `3 : body`.

On the other hand, if `body` contains both `x.` and `y.` then the result is a tacit verb of which the dyadic case is equivalent to `4 : body`.

For the purpose of generating tacit functions, the body is restricted to being a single string or one line.

Recall that with `3 : body`, the body is not evaluated when the definition is entered. However, with `13 : body`, then in effect the body is evaluated. For example:

<code>k =: 99</code>	<code>p =: 3 : 'y.+k'</code>	<code>q =: 13 : 'y.+k'</code>	<code>p 6</code>	<code>q 6</code>
99	<code>3 : 'y.+k'</code>	<code>] + 99"_</code>	105	105

We see that `p` is defined in terms of `k` while `q` is not. While `p` and `q` are at present equivalent, any subsequent change in the value of `k` will render them no longer equivalent.

<code>k =: 0</code>	<code>p 6</code>	<code>q 6</code>
0	6	105

A name with no assigned value is assumed to denote a verb. In the following example, note that `f` is unassigned, `c` is a predefined conjunction and `g` is a predefined verb.

```
C =: @:
g =: %:
```

<code>f oo =: 13 : '(f C f y.) , g y.'</code>	<code>f =: *:</code>	<code>f oo 4</code>
<code>f@:f , [: g ]</code>	<code>*:</code>	<code>256 2</code>

This is the end of Chapter 10

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 7 Aug 2002

# Chapter 11: Tacit Verbs

## Concluded

In this chapter we consider some general points in writing expressions for tacit verbs.

Here is an example of a tacit verb. It multiplies its argument by 3:

<code>v =: * &amp; 3</code>	<code>v 4</code>
<code>*&amp;3</code>	12

Recall from [Chapter 03 p3](#) that the bonding operator `&` produces a monad from a dyad by fixing one of the arguments of the dyad. The scheme is that if `N` is a noun and `v` a dyadic verb, then:

$(N \& V) \ y$  means  $N \ V \ y$   
 $(V \& N) \ y$  means  $y \ V \ N$

We take the bonding operator `&` as an example of a typical operator, where arguments may be nouns or verbs. In general, `N` can be an expression denoting a noun, and `v` an expression denoting a verb. We look now at how these expressions get evaluated. The general rules are set out formally in [Chapter 31 p26](#) but here we take an informal first look at a few of the main points.

## 11.1 If In Doubt, Parenthesize

Here is another tacit verb. Its general form is `v&N`. It multiplies its argument by `5%4`, that is, by `1.25`

<code>scale =: * &amp; (5 % 4)</code>	<code>scale 8</code>
---------------------------------------	----------------------



* & 1.25	10
----------	----

Are the parentheses around 5 % 4 necessary here? If we omit them, we see:

SCALE =: \* & 5 % 4  
SCALE  
1.25

so they evidently make a difference. SCALE is a number, not a verb. The result of 1.25 is produced by applying the verb \* & 5 to the argument % 4 (the reciprocal of 4)

% 4	( * & 5 ) ( % 4 )	* & 5 % 4
0.25	1.25	1.25

We have a general rule: informally we can say that conjunctions get applied before adjacent verbs. Thus in the expression \* & 5 % 4 the first step is to apply the & operator to its arguments \* and 5.

Why is the right argument of & just 5 and not 5%4? Because of another general rule: the right argument of a conjunction is as short as possible. We say that a conjunction has a "short right scope". (By contrast, we say that a verb has a "long right scope" to express what we earlier called the "rightmost first" rule for verbs.

What about the left argument of an operator? An adverb or conjunction is said to have "long left scope", that is, as much as possible. For example, here is a verb z which adds 3 to the square of its argument. 3 plus the square of 2 is 7.

z =: 3 & + @: *:	z 2
3&+@: *:	7

We see that the left argument of @: is the whole of 3&+.

If we are in doubt in any particular case we can always make our intention clear. We can write parentheses around a part of an expression, that is, around a function - verb or operator - together with its intended argument(s). For example, verb  $z$  can be written with parentheses as:

$z =: (3 \ \& \ +) \ @: \ *:$	$z \ 2$
$3\&+@: *:$	$7$

Sometimes parentheses are necessary and sometimes not, but, let me emphasize, if in doubt, parenthesize.

## 11.2 Names of Nouns Are Evaluated

In an expression of the general form  $N\&V$  or  $V\&N$ , the the names of any nouns occurring in  $N$  are evaluated right away. Here is an example of a function  $f$  to multiply by five-fourths. The numerical value is given as  $a\%b$  where  $a$  and  $b$  are nouns.

$a =: 5$	$b =: 4$	$f =: * \ \& \ (a \ \% \ b)$	$f \ 8$
$5$	$4$	$*\&1.25$	$10$

We see that function  $f$  contains the computed number  $1.25$  so that  $a\%b$  has been evaluated.

## 11.3 Names of Verb Are Not Evaluated

In  $N\&V$  the verb-expression  $v$  is not necessarily fully evaluated. If expression  $v$  is the name of a verb, then the name is enough:

<code>w =: *</code>	<code>g =: w &amp; (a % b)</code>	<code>g 8</code>
<code>*</code>	<code>w&amp;1.25</code>	<code>10</code>

## 11.4 Unknowns are Verbs

When a new name is encountered, it is assumed to be a yet-to-be-defined verb if it possibly can be.

<code>h =: ytbdd &amp; (a%b)</code>	<code>ytbdd =: *</code>	<code>h 8</code>
<code>ytbdd&amp;1.25</code>	<code>*</code>	<code>10</code>

Any sequence of hitherto-unknown names is assumed to be a train of verbs:

```
Ralph Waldo Emerson
Ralph Waldo Emerson
```

Consequently, a verb can be defined in "top-down" fashion, that is, with detail presented later. For example, here is a Celsius-to-Fahrenheit converter presented top-down.

```
ctof =: shift @ scale
      shift =: + & 32
      scale =: * & (9 % 5)
```

<code>ctof</code>	<code>ctof 0 100</code>
<code>shift@scale</code>	<code>32 212</code>

We can see that `ctof` is defined solely in terms of (the names) `scale` and `shift`.

Hence if we now change `scale` or `shift` we will effectively change the definition of `ctof`.

```

ctof 100
212
scale =: * & 2
ctof 100
232
scale =: * & (9 % 5)
ctof 100
212

```

The possibility of changing the definition of a function simply by changing one of its subordinate functions, may or may not be regarded as desirable. It is useful, in so far as we can correct a definition just by changing a small part. However, it may be a source of error: we may introduce a new verb `scale` say forgetting that `scale` is already defined as subordinate in `ctof`.

There are ways to protect `ctof` against accidental redefinition of its subordinate functions. Firstly, we can put a wrapper of explicit definition around it, making `scale` and `shift` local, thus:

```

CTOF =: 3 : 0
shift =. + & 32
scale =. * & (9 % 5)
shift @ scale y.
)
CTOF 100
212

```

A second method is to, so to speak, "freezing" or "fixing" the definition of `ctof`, with the "Fix" adverb `f.` (letter-f dot). Observe the difference between the values of the verbs `ctof` and `(ctof f.)`

<code>ctof</code>	<code>ctof f.</code>
<code>shift@scale</code>	<code>+&amp;32@(*&amp;1.800000000000000004)</code>

We see that adverb `f.` applied to a tacit verb replaces names by definitions, giving

an equivalent verb defined only in terms of built-in functions. Here is yet another definition of `ctof`.

```
scale =: * & (9 % 5)
shift =: + & 32
ctof   =: (shift @ scale) f.
```

ctof	ctof 0 100
+&32@(*&1.800000000000000004)	32 212

After this definition, the names `scale` and `shift` are still defined, but are no longer important in the definition of `ctof`.

## 11.5 Parametric Functions

The following example shows the consequences of nouns being evaluated and verbs not in an expression for a tacit verb.

A curve may be specified by an equation such as, for example:

$$y = \text{lambda} * x * (1 - x)$$

This equation describes a family of similar parabolic curves, and different members of the family are picked out by choosing different values for the number `lambda`.

A function to correspond to this equation might be written explicitly as verb `P`:

```
P =: 3 : 'lambda * y. * (1-y.)'
```

Here `lambda` is not an argument to function `P`, but a variable, a number, which makes a difference to the result. We say that `lambda` is a parameter, or that function `P` is parametric.

x=:0.6	lambda=: 3.0	P x	lambda=: 3.5	P x
--------	--------------	-----	--------------	-----

0.6	3	0.72	3.5	0.84
-----	---	------	-----	------

Now, can we write a tacit version of  $P$  taking `lambda` as a parameter?

`lambda` is currently 3.5. If we now generate a tacit form of  $P$

```

tP =: 13 : 'lambda * y. * (1-y.)'
tP
3.5"_ * ] * 1: - ]

```

then we see that `lambda` is treated as a constant, not a parameter. This is not what we want. We try again, this time ensuring that `lambda` is not specified beforehand, by erasing it:

```

erase <'lambda' 1 tP =: 13 : 'lambda * y. * (1-y.)' tP [:
lambda [: * ] * 1: - ]

```

Now we see that `tP` is a train of verbs, where `lambda` (being unknown) is assumed to be a verb. This assumption conflicts with the intended meaning of `lambda` as a number. Hence with `lambda` as a number, we get an error:

lambda=: 3.5	tP x
3.5	error

Whether or not `lambda` is specified in advance, it appears that a fully tacit exact equivalent to  $P$  is not possible. However we can come close.

One possibility is to compromise on "fully tacit". Here `tP` is a train of verbs, where the first is explicitly-defined to deliver the value of `lambda` regardless of its argument.

tP =: (3 : 'lambda') * ] * (1: - ])	tP x
3 : 'lambda' * ] * 1: - ]	0.84

Another possibility is to compromise on "exact equivalent". Here we take parameter `lambda` to be, not a number, but a constant function (see [Chapter 09 p9](#)) which delivers a number.

For example, a value for the parameter could be written as

```
lambda =: 3.5 " 0
```

and `tP` could be defined as:

<code>tP =: lambda * ] * (1: - ])</code>	<code>tP x</code>
<code>lambda * ] * 1: - ]</code>	0.84

Now we can vary the parameter without redefining the function:

<code>lambda =: 3.75 " 0</code>	<code>tP x</code>
3.75"0	0.9

This is the end of Chapter 11

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 7 Aug 2002

# Chapter 12: Explicit Verbs

This chapter continues from [Chapter 04 p4](#) the theme of the explicit definition of verbs.

## 12.1 The Colon Conjunction

Recall from [Chapter 04 p4](#) the example of an explicit monad: the Fahrenheit-to-Celsius converter:

```
ftoc =: 3 : 0
z =. y. - 32
z * 5 % 9
)
```

The general scheme for an explicitly-defined function is to provide two arguments for the colon-conjunction, in the form

```
type : body
```

The type is a number: type-3 functions are monadic verbs or ambivalent verbs, while type-4 functions are strictly dyadic verbs (that is, with no monadic case). The complete set of types is as follows:

0	noun
1	explicit adverb
2	explicit conjunction
3	explicit verb (monad or ambivalent)
4	explicit verb (dyad)
13	generated tacit verb

Rather than numbers, some people may prefer to use conventional pre-assigned names. The pre-assigned names and corresponding values are:



noun	adverb	conjunction	verb	monad	dyad
0	1	2	3	3	4

and additionally:

def	define
<pre> +-+  :  +-+ </pre>	<pre> +-+--+  : 0  +-+--+ </pre>

Thus the `ftoc` example could be also written as:

```

ftoc =: verb define
z =. y. - 32
z * 5 % 9
)

```

In this chapter, we will be concerned only with types 3 and 4. For details of types 1 and 2 see [Chapter 13 p13](#) and for type 13 see [Chapter 10 p10](#).

## 12.1.1 Body Styles

The body of an explicit definition consists of one or more lines of text. There are several ways to provide the body. The example above, `ftoc`, shows what is often most convenient: lines introduced by a right argument of zero for the colon operator. A variation is where the body has only one line. Here the body is written as a string:

```
ftoc2 =: 3 : '(y. - 32) * 5 % 9'
```

Another variation allows a multi-line function to be written compactly by embedding line-feeds. `LF` is predefined to be the line-feed character. Notice that the

whole body must be parenthesized.

```
ftoc3 =: 3 : ('z =: y. - 32', LF, 'z * 5 % 9')
ftoc3
+--+-----+
|3|:|z =: y. - 32|
| | |z * 5 % 9   |
+--+-----+
```

Another variation uses a boxed list of lines (again with the body parenthesized):

```
ftoc4 =: 3 : ('z =. y. - 32' ; 'z * 5 % 9')
ftoc4
+--+-----+
|3|:|z =. y. - 32|
| | |z * 5 % 9   |
+--+-----+
```

A character array is also possible. Notice that these are not variations of syntax, but rather alternative expressions for constructing a data-structure acceptable as the right-argument of the colon operator.

An ambivalent function is presented by separating the monadic case from the dyadic with a line consisting of a solo colon. For example:

```
log =: 3 : 0
^ . y.      NB. monad
:
x. ^ . y. NB. dyad
)
```

log	log 2.7182818	10 log 100
<pre>+--+-----+  3 : ^ . y.      NB. monad       :                 x. ^ . y. NB. dyad   +--+-----+</pre>	1	2

## 12.2 Assignments

In this section we consider assignments, which are of significance in defining explicit functions.

### 12.2.1 Local and Global Variables

Consider the example

```
foo =: 3 : 0
L =. y.
G =: y.
L
)
```

Here, the assignment of the form

```
L =. expression
```

causes the value of `expression` to be assigned to a local variable named `L`. Saying that `L` is local means that `L` exists only while the function `foo` is executing, and furthermore this `L` is distinct from any other variable named `L`. By contrast, the assignment of the form

```
G =: expression
```

causes the value of `expression` to be assigned to a global variable named `G`. Saying that `G` is global means that the unique variable `G` exists independently, in its own right.

To illustrate, we define two GLOBAL variables called `L` and `G`, then execute `foo` to show that the `L` mentioned in `foo` is not the same as global `L`, while the `G` mentioned in `foo` is the same as global `G`:

```
L =: 'old L'
G =: 'old G'
```

foo	foo 'new'	L	G
<pre> +---+-----+  3 : L =.  y.       G =:  y.       L        +---+-----+ </pre>	new	old L	new

## 12.2.2 Local Functions

A local variable may be a noun, as we have seen, or it may be a locally defined function. A local function may be tacit or explicit, as in the following example

```

foo =: 3 : 0
Square =. *:
Cube   =. 3 : 'y. * y. * y.'
Square y. + Cube y.
)

```

```

foo 2
100

```

However, what we can't have is a local function defined by an inner script. Recall that a script is terminated by a solo right parenthesis, so we cannot have one script inside another. Instead, we could use an alternative form for the body of an inner function, such as `scale` in the following example:

```

FTOC =: 3 : 0
line1 =. 'k =. 5 % 9'
line2 =. 'k * y.'
scale =. 3 : (line1 ; line2)
scale y. - 32
)

```

```

FTOC 212
100

```

One final point on the topic of inner functions. A local variable is either strictly local or strictly global. Consider the following:

```

K =: 'hello '

```

```

    zip =: 3 : 0
K =. 'goodbye '
zap =. 3 : 'K , y.'
zap y.
)

```

```

    zip 'George'
hello George

```

We see that there is a global `κ` and a local `κ`. The inner function `zap` uses the global `κ` because the `κ` which is local to `zip` is not local to `zap`.

## 12.2.3 Multiple and Indirect Assignments

`J` provides a convenient means of unpacking a list by assigning different names to different items.

'day mo yr' =: 16 10 95	day	mo	yr
16 10 95	16	10	95

Instead of a simple name to the left of the assignment, we have a string with names separated by spaces.

A variation uses a boxed set of names:

( 'day' ; 'mo' ; 'yr' ) =: 17 11 96	day	mo	yr
17 11 96	17	11	96

The parentheses around the left hand of the assignment force evaluation as a set of names, to give what is called "indirect assignment". To illustrate:

```

N =: 'DAY' ; 'MO' ; 'YR'

```

(N) =: 18 12 97	DAY	MO	YR
18 12 97	18	12	97

As a convenience, a multiple assignment will automatically remove one layer of boxing from the right-hand side:

(N) =: 19; 'Jan'; 98	DAY	MO	YR
+---+---+---+   19   Jan   98   +---+---+---+	19	Jan	98

# 12.2.4 Unpacking the Arguments

Every J function takes exactly one or exactly two arguments - not zero and not more than two. This may appear to be a limitation but in fact is not. A collection of values can be packaged up into a list, or boxed list, to form in effect multiple arguments to the J function. However, the J function must unpack the values again. A convenient way to do this is with the multiple assignment. For example, the familiar formula to find the roots of a quadratic  $(a \cdot x^2) + (b \cdot x) + c$ , given the vector of coefficients  $a, b, c$  might be:

```
rq =: 3 : 0
'a b c' =: y.
(( -b) (+, -) %: (b^2) - 4*a*c) % (2*a)
)
```

rq 1 1 _6	rq 1 ; 1 ; _6
-----------	---------------

2 _3	2 _3
------	------

## 12.3 Flow of Control

In an explicit definition, the sequence of execution of the lines is often the first line, then the second, and so on through to the last. The result of the whole function is the result computed by the last line to be executed. This sequence may be varied by the presence of CONTROL WORDS, such as `if.` or `while.` .

### 12.3.1 if.

Here is an example of a function in which a choice is made about which lines to execute. The function classifies the temperature of porridge.

```
CTP =: 3 : 0
if.    y. > 80
do.    'too hot'
else.  'OK'
end.
)
```

```
CTP 70
OK
```

This example shows the pattern:

```
if. T do. B1 else. B2 end.
```

meaning: if the expression `T` evaluates to "true", then execute the expression `B1`, and otherwise execute the expression `B2`.

More generally, `T`, `B1` and `B2` may be what are called BLOCKS. A block is a sequence of zero, one, or more expressions, the sequence being surrounded by control words. Thus in the example above, the block `(y. > 80)` is delimited by the control words `if.` and `do.`. Here is another example, to form the sum of a list, where the `T`-block and the `B2`-block each consist of a sequence.

```
sum =: 3 : 0
if.    length =. # y.
      length = 0
```

```

do.    0
else.  first =. {. y.
      rest  =. }. y.
      first + sum rest
end.
)

      sum 1 2 3
6

```

Here we see that the value of the T-block (true or false) is the value of the last expression in the sequence, (length = 0)

The expressions in a block may themselves be (inner) blocks, as shown by another function to classify the temperature of porridge:

```

      ClaTePo =: 3 : 0

if. y. > 80 do.      'too hot'
else.
      if. y. < 60 do. 'too cold'
      else.          'just right'
      end.
end.
)

      ClaTePo 70
just right

```

This example also shows that control-words serve to terminate J expressions just as end-of-line terminates J expressions. Hence control-words allow some freedom in laying out a definition for the most pleasing appearance.

A neater variation of the last example is:

```

      CLATEPO =: 3 : 0
if.      y. > 80 do. 'too hot'
elseif. y. < 60 do. 'too cold'
elseif. 1      do. 'just right'
end.
)

      CLATEPO 70
just right

```



The second scheme for `if.` is:

```
if.      T1 do. B1
elseif.  T2 do. B2
...
elseif.  Tn do. Bn
end.
```

Notice that according to this scheme, if all of the tests `T1 ... Tn` fail, then none of the blocks `B1 .. Bn` will be executed. Consequently we may wish to make `Tn` a catch-all test, with the constant value 1, as in the example of CLATEPO above.

## 12.3.2 while. and whilst.

In the general pattern

```
while. T do. B end.
```

the block `B` is executed repeatedly so long as block `T` evaluates to true. Here is an example, a version of the factorial function:

```
fact =: 3 : 0
r =. 1
while. y. > 1
do.    r =. r * y.
      y. =. y. - 1
end.
r
)
```

```
fact 5
120
```

The variation `whilst. T do. B end.` means

```
B
while. T do. B end.
```

that is, block B is executed once, and then repeatedly so long as block T is true.

### 12.3.3 for

The pattern

```
for_a. A do. B. end.
```

means: for each item a in array A, execute block B. Here a may be any name; the variable a takes on the value of each item of A in turn. For example, to sum a list:

```
Sum =: 3 : 0
r =. 0
for_term. y. do. r =. r+term end.
r
)

Sum 1 2 3
6
```

In addition to the variable a for the value of an item, the variable a\_index is available to give the index of the item. For example, this function numbers the items:

```
f3 =: 3 : 0
r =. 0 2 $ 0
for_item. y. do. r =. r , (item_index; item) end.
r
)

f3 'ab';'cdef';'gh'
++-----+
|0|ab  |
++-----+
|1|cdef|
++-----+
|2|gh  |
++-----+
```

Another variation is the pattern `for. A. do. B end.` which is similar except that the variables a and a\_index are not available.

## 12.3.4 try.

Here we look at a way of handling errors. The scheme is that:

```
try. B1 catch. B2 end.
```

means: execute block B1. If for any reason B1 fails, then B1 is abandoned and B2 executed instead. If B1 succeeds, then B2 is not executed. The following example is a function which tests that the argument supplied is valid.

```
foo =: 3 : 0
try. *: y. catch. 'argument must be numeric' end.
)
```

foo 2	foo 'hello'
4	argument must be numeric

## 12.3.5 goto and label

Given any name, such as `qwerty`, then two control-words may be constructed: `label_qwerty.` and `goto_qwerty.` . (Notice that both end with a dot).

The meaning of `label_qwerty.` is that it provides a name for a block which begins at the point where `label_qwerty.` occurs. This block ends at the end of the whole explicit definition. The purpose of naming a block in this way is that `goto_qwerty.` means that the named block is to be executed next. Here is an example: yet another factorial function.

```
facto =: 3 : 0
r =. 1
label_again.
  if. y. < 2 do. goto_done. end.
  r =. r * y.
  y. =. y. - 1
  goto_again.
label_done.
r
)
```

## **12.3.6 break, continue and return**

to be supplied

This is the end of Chapter 12.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 16 Aug 2002

# Chapter 13: Explicit Operators

This chapter covers explicit definition of operators, that is, adverbs and conjunctions defined with the colon conjunction. The scheme is:

```
1 : body      is an adverb
2 : body      is a conjunction
```

where `body` is one or more lines of text. The possibilities for the result produced by an operator are: a tacit verb, an explicit verb, a noun or another operator. We will look at each case in turn.

## 13.1 Operators Generating Tacit Verbs

Recall from [Chapter 07 p7](#) the built-in rank conjunction `"`. For any verb `u`, the expression `u"0` is a verb which applies `u` to the 0-cells (scalars) of its argument.

Now suppose we aim to define an adverb `A`, to generate a verb according to the scheme: for any verb `u`

```
u A      is to be      u " 0
```

Adverb `A` is defined explicitly like this:

<code>A =: 1 : 'u. " 0'</code>	<code>f =: &lt; A</code>	<code>f 1 2</code>
<code>1 : 'u. " 0'</code>	<code>&lt;"0</code>	<code>+--++</code> <code>  1   2  </code> <code>+--++</code>

In the definition `(A =: 1 : 'u. " 0')` the left argument of the colon is `1`, meaning "adverb".

The right argument is the string 'u. " 0'. This string has the form of a tacit verb, where u. stands for whatever verb will be supplied as argument to the adverb A.

Adverbs are so called because, in English grammar, adverbs modify verbs. In J by contrast, adverbs and conjunctions in general can take nouns or verbs as arguments. In the following example, adverb w is to generate a verb according to the scheme: for integer u

u w is to be < " u

that is, u w boxes the rank-u cells of the argument. The definition of w is shown by:

W =: 1 : '< " u.'	0 W	z =: 'abc'	0 W z	1 W z
1 : '< " u.'	<"0	abc	+--+--+  a b c  +--+--+	+----+  abc  +----+

For another example of an adverb, recall the dyad # where x # y selects items from y according to the bitstring x.

y =: 1 0 2 3	1 0 1 1 # y
1 0 2 3	1 2 3

To select items greater than 0, we can apply the test-verb (>&0)

y	>&0 y	(>&0 y) # y
1 0 2 3	1 0 1 1	1 2 3

A tacit verb to select items greater than 0 can be written as a fork f:

f =: >&0 # ]	f y
>&0 # ]	1 2 3

This fork can be generalised into an adverb, B say, to generate a verb to select items according to whatever verb is supplied in place of the test >&0.

B =: 1 : 'u. # ]'

If we supply >&1 as a test-verb:

g =: (>&1) B	y	g y
>&1 # ]	1 0 2 3	2 3

We see that the body of B is the fork to be generated, with u. standing for the argument-verb to be supplied.

Conjunctions, taking two arguments, are defined with (2 : '...'). The left argument is u. and the right is v.

For example, consider a conjunction THEN, to apply one verb and then apply another to the result, that is, a composition. The scheme we want is:

u THEN v is to be v @: u

and the definition of THEN is:

THEN =: 2 : 'v. @: u.'	h =: *: THEN <	h 1 2 3
2 : 'v. @: u.'	<@:*	+-----+   1 4 9   +-----+

---

For another example, consider counting (with #) those items of a list which are greater than 0. A verb to do this might be:

<code>foo =: # @: (&gt;0 # ])</code>	<code>y</code>	<code>foo y</code>
<code>#@: (&gt;0 # ])</code>	<code>1 0 2 3</code>	<code>3</code>

We can generalize `foo` to apply a given verb `u` to items selected by another given verb `v`. We define a conjunction `C` with the scheme

NB. `u C v` is to be `u @: (v # ])`

and the definition of `C` is straightforwardly:

<code>C =: 2 : 'u. @: (v. # ])'</code>	<code>f =: # C (&gt;0)</code>	<code>y</code>	<code>f y</code>
<code>2 : 'u. @: (v. # ])'</code>	<code>#@: (&gt;0 # ])</code>	<code>1 0 2 3</code>	<code>3</code>

### 13.1.1 Multiline Bodies

The right argument of colon we may call the body of the definition of our operator. In the examples so far, the body was a string, a schematic tacit verb, for example `'v .@: u.'`. This is the verb to be delivered by our operator. More generally, the body can be several lines. The idea is that, when the operator is applied to its argument, the whole body is executed. That is, each line is evaluated in turn and the result delivered is the value of the last line evaluated. This is exactly analogous to explicit verbs, except that here the result is a value of type "function" rather than of type "array".

Here is an example of a multi-line body, the previous example done in two steps. To apply `u` to items selected by `v`, a scheme for conjunction `D` could be written:

`u D v` is to be `(u @: select)` where `select` is `v # ]`



and `D` defined by

```
D =: 2 : 0
select =: v. # ]
u. @: select
)
```

Again counting items greater than 0, we have

<code>f =: # D (&gt;0)</code>	<code>y</code>	<code>f y</code>
<code>#@:select</code>	<code>1 0 2 3</code>	<code>3</code>

The first line of `D` computes an inner function `select` from the right argument. The second line composes `select` with the left argument, and this is the result-verb delivered by `D`.

Now this definition has an undesirable feature: we see that `select` is defined as a global (with `=:`). It would be better if `select` were local.

However, we can see, by looking at the value of the verb `f` above, that `select` must be available when we apply `f`. If `select` is local to `D`, it will not be available when needed.

We can in effect make `select` local by using the "Fix" adverb (`f.`) (letter-f dot.) The effect of applying "Fix" to a verb is to produce an equivalent verb in which names are replaced by their corresponding definitions. That is, "Fix" resolves a tacit verb into its primitives. For example:

<code>p =: +</code>	<code>q =: *</code>	<code>r =: p,q</code>	<code>r f.</code>
<code>+</code>	<code>*</code>	<code>p , q</code>	<code>+ , *</code>

Here is how we use Fix to enable `select` to be local. In the example below, notice that we Fix the result-expression on the last line:

```

E =: 2 : 0
select =. v. # ]
(u. @: select) f.
)

```

Now a verb to count greater-than-0 items can be written:

<code>g =: # E (&gt;&amp;0)</code>	<code>y</code>	<code>g y</code>
<code>#@: (&gt;&amp;0 # ])</code>	<code>1 0 2 3</code>	<code>3</code>

We see that `g`, unlike `f`, has no local names.

## 13.2 Explicit Operators Generating Explicit Verbs

### 13.2.1 Adverb Generating Monad

Consider the following explicit monadic verb, `e`. It selects items greater than 0, by applying the test-verb `>&0`.

<code>e =: 3 : '(&gt;&amp;0 y.) # y.'</code>	<code>y</code>	<code>e y</code>
<code>3 : '(&gt;&amp;0 y.) # y.'</code>	<code>1 0 2 3</code>	<code>1 2 3</code>

We can generalise `e` to form an adverb, `F` say, which selects items according to a supplied test-verb. The scheme we want is: for any verb `u`:

`u F` is to be `3 : '(u y.) # y.'`

Adverb  $F$  is defined by:

```
F =: 1 : '(u. y.) # y.'
```

Now the verb `>1 F` will select items greater than 1:

y	>1 F y
1 0 2 3	2 3

In the body of  $F$  the variable  $u.$  stands for a verb to be supplied as argument to adverb  $F$ . If this argument is say `>1`, then  $y.$  stands for an argument to the generated explicit verb `3 : '(>1 y.) # y.'`

That is, our method of defining the generated verb is to write out the body of an explicit definition, with  $u.$  at places where a supplied verb is to be substituted.

## 13.2.2 Conjunction Generating Monad

A conjunction takes two arguments, called  $u.$  and  $v.$ . Here is an example of a conjunction to generate an explicit monad.

As before, we specify the generated verb, by writing out the body of an explicit verb. Here  $y.$  stands for the argument of the generated verb and  $u.$  and  $v.$  stand for argument-verbs to be supplied to the conjunction. In this example the body is multi-line. As before,  $u.$  will be applied to items selected by  $v.$

```
G =: 2 : 0
selected =. (v. y.) # y.
u. selected
)
```

Now a verb to count greater-than-zero items can be written as `# G (>0)`:

y	# G (>0) y
---	------------

1 0 2 3	3
---------	---

### 13.2.3 Generating a Dyad

For the next example, let us define a conjunction generating an explicit dyad.  
 Suppose we want a conjunction  $H$  such that, schematically,

$u\ H\ v$  is to be  $4 : '(u\ x.) + (v\ y.)'$

Now it is a fact that all the generated verbs are defined in terms of  $3 :$ , not  $4 :$ .  
 We can write a dyad with  $3 :$  by beginning a multi-line body with the solo colon which separates the monadic case from the dyadic.

Thus, schematically, we have to say, for verbs  $u$  and  $v$ :

$u\ H\ v$  is to be  $3 : 0$   
 $:$   
 $(u\ x.) + (v\ y.)$   
 $)$

The explicit definition of  $H$  follows straightforwardly:

$H =: 2 : 0$   
 $:$   
 $(u.\ x.) + (v.\ y.)$   
 $)$

We see:

$(*: 2) + (\%: 16)$	$2 (*: H \%:) 16$
8	8

### 13.2.4 Review

So far, we have seen that for operators introduced with 1 : or 2 :, there are two kinds of definition.

- The first kind generates a tacit function. The body of the operator is executed (that, is evaluated) to compute the value of the result-function. Notice that the argument-variables occurring in the body are  $u.$  or  $v.$
- The second kind generates an explicit function. The body of the operator IS the body of the generated function, after substitution of arguments. Notice that the argument-variables occurring in the body are  $u.$  or  $v.$  or  $x.$  or  $y.$ . The J system recognises which kind is intended by determining which of the argument-variables  $u.$   $v.$   $x.$   $y.$  occur in the the body.

If we have ONLY  $u.$  or  $v.$  or both, then the generated function is tacit.

If we have BOTH ( $u.$  or  $v.$ ) AND ( $x.$  or  $y.$ ) then the generated function is explicit.

On this basis, the cases we have considered are:

1 : ' $.. u. ..$ ' tacit-generating adverb

2 : ' $.. u. v. ..$ ' tacit-generating conjunction

1 : ' $.. u. y. ..$ ' explicit-monad-generating adverb

1 : ' $.. u. x. y. ..$ ' explicit-dyad-generating adverb

2 : ' $.. u. v. y. ..$ ' explicit-monad-generating conjunction

2 : ' $.. u. v. x. y. ..$ ' explicit-dyad-generating conjunction

## 13.2.5 Alternative Names for Argument-Variables

The arguments to operators may be nouns or verbs. There is a way of constraining arguments to be nouns only, that is, to cause verbs to be signalled as errors. To

impose the constraint, we write the argument-variables as `m.` and `n.` rather than as `u.` and `v.`. For example, without the constraint we could write:

<code>P =: 1 : '+ &amp; u.'</code>	<code>* P</code>	<code>7 P</code>
<code>1 : '+ &amp; u.'</code>	<code>+&amp;*</code>	<code>+&amp;7</code>

With the constraint we write:

<code>Q =: 1 : '+ &amp; m.'</code>	<code>* Q</code>	<code>7 Q</code>
<code>1 : '+ &amp; m.'</code>	error	<code>+&amp;7</code>

We said above that with ONLY `u.` or `v.` or both occurring as argument variables, we get a tacit verb generated. For the sake of completeness, we should add `m.` and `n.` to this list.

Furthermore, if the only argument variables are `x.` or `y.` or both, we get a tacit verb, not an explicit verb. That is, in the absence of `u.` or `v.` or `m.` or `n.` then `x.` and `y.` are equivalent to `u.` and `v.`.

## 13.2.6 Executing the Body (Or Not)

To demonstrate when the body gets executed (or evaluated), we can use a utility verb which displays its argument on-screen:

```
display =: (1 !: 2) & 2
```

Now insert `display 'hello'` into a tacit-generating operator:

```
R =: 2 : 0
display 'hello'
select =. v. # ]
(u. @: select) f.
)
```

When  $R$  is applied to its argument, the body is demonstrably executed:

```
f =: # R (>&0)
hello
```

```
f 1 0 2 0 3
3
```

By contrast, if we do the same with an explicit-generating operator:

```
s =: 2 : 0
display 'hello'
selected =. (v. y.) # y.
u. selected
)
```

we see that the body of  $s$  is NOT executed when the operator is applied to its argument, but it IS executed when the generated verb  $g$  is applied.

```
g =: # S (>&0)
g 1 0 2 0 3
hello
3
```

## 13.3 Operators Generating Nouns

Operators can generate nouns as well as verbs. Here is an example.

A fixed point of a function  $f$  is a value  $p$  such that  $(f\ p) = p$ . If we take  $f$  to be

```
f =: 3 : '2.8 * y. * (1-y.)'
```

then we see that 0.642857 is a fixed-point of  $f$

```
f 0.642857
0.642857
```

Not every function has a fixed point, but if there is one we may be able to find it by

repeatedly applying the function (with  $\wedge: \_$ ) to a suitable starting value. A crude fixed-point-finder can be written as an adverb `FPF` which takes the given function as argument, with `0.5` for a starting value.

<code>FPF =: 1 : '(u. ^: _ ) 0.5'</code>	<code>p =: f FPF</code>	<code>f p</code>
<code>1 : '(u. ^: _ ) 0.5'</code>	<code>0.642857</code>	<code>0.642857</code>

## 13.4 Operators Generating Operators

Here is an example of an adverb generating an adverb.

First note that (as covered in [Chapter 15 p15](#)) if we supply one argument to a conjunction we get an adverb. The expression `(@: *: )` is an adverb which means "composed with square". To illustrate:

<code>CS =: @: *: </code>	<code>- CS</code>	<code>- CS 2 3</code>	<code>- *: 2 3</code>
<code>@: *: </code>	<code>-@: *: </code>	<code>_4 _9</code>	<code>_4 _9</code>

Now back to the main example of this section. We aim to define an explicit adverb,  $\kappa$  say, which generates an adverb according to the scheme: for a verb  $u$

$u \ \kappa$  is to be  $@: u$

Adverb  $\kappa$  can be defined as below. We see that adverb  $\kappa$  delivers as a result adverb  $L$ :

<code>K =: 1 : '@: u.'</code>	<code>L =: *: K</code>	<code>- L</code>	<code>- L 2 3</code>
<code>1 : '@: u.'</code>	<code>@: *: </code>	<code>-@: *: </code>	<code>_4 _9</code>



This is the end of Chapter 13.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 17 Aug 2002

# Chapter 14: Gerunds

What is a gerund, and what is it good for? Briefly, a gerund represents a list of verbs. It is useful, in the main, for supplying a list of verbs as a single argument to an operator.

The plan for this chapter is:

- to introduce gerunds
- to look at some built-in operators which can take gerunds as arguments
- to look at user-defined operators taking gerund arguments

## 14.1 Making Gerunds: The Tie Conjunction

Recall from [Chapter 10 p10](#) how we defined a verb with several cases. Here is a small example as a reminder. To find the absolute value of a number  $x$  we compute  $(+x)$ , or  $(-x)$  if the number is negative, thus:

<code>abs =: + ` - @. (&lt; &amp; 0)</code>	<code>abs _3</code>
<code>+ ` -@. (&lt;&amp;0)</code>	3

The expression  $(+ \text{ ` } -)$  looks like a list of verbs. Here the two verbs  $+$  and  $-$  are tied together with the "Tie" conjunction (```, backquote, different from `'`) to produce a gerund.

```

+ ` -
+-+--+
|+|-|
+-+--+

```

We see that the gerund (+ ` -) is a list of two boxes, each of which contains a representation of a verb. A gerund is a noun - a list of boxes. Here is another gerund which represents three verbs:

```
G =: + ` - ` abs
G
+--+---+
|+|-|abs|
+--+---+
```

Inside each box there is a data structure which represents, or encodes, a verb. Here we will not be concerned with the details of this representation, which will be covered in Chapter 27.

## 14.2 Recovering the Verbs from a Gerund

The verbs packed into a gerund can be unpacked again with the built-in adverb "Evoke Gerund" which is denoted by the expression (`: 6). Let us call this EV.

```
EV =: `: 6
```

Adverb EV applied to a gerund yields a train of all the verbs in the gerund. In the next example, the result foo is a 3-train, that is a fork.

```
f =: 'f' & ,
g =: 'g' & ,
```

H=: f ` , ` g	foo =: H EV	foo 'o'
+--+---+  f , g  +--+---+	f , g	fogo

Individual verbs can be unpacked by indexing the boxed list H and then applying EV.

H	2{H	vb =: (2{H) EV	vb 'o'
+--+--+  f , g  +--+--+	++  g  ++	g	go

Shorter trains can be unpacked from a gerund, again by indexing.

H	1 2 { H	tr =: (1 2 { H) EV	tr 'a'
+--+--+  f , g  +--+--+	++  , g  ++	, g	aga

Now we come to the uses of gerunds.

## 14.3 Gerunds As Arguments to Built-In Operators

A major use of gerunds is that they can be supplied to operators as a single argument containing multiple verbs. We look first at further built-in operators taking gerund arguments, and then at examples of home-made operators.

### 14.3.1 Gerund as Argument to APPEND Adverb

There is a built-in adverb called "APPEND", denoted by the expression (`` : 0`). It applies a list of verbs to a single argument to give a list of results. For example:

```
APPEND =: ` : 0
sum     =: +/
count   =: #
mean    =: sum % count
G1      =: count ` sum ` mean
```

G1	foo =: G1 APPEND	foo 1 2 3
<pre> +-----+-----+-----+  count sum mean  +-----+-----+-----+ </pre>	count`sum`mean`:0	3 6 2

The adverb is called `APPEND` because the results of the individual verbs in the gerund are appended, that is formed into a list. The general scheme is that for verbs `u, v, w, ...` then

`(u`v`w...) APPEND y` means `(u y), (v y), (w y) , ...`

Here is another example, showing that a gerund can be, not just a one-dimensional list, but an array of verbs. The list of verbs `G1` formed by "Tie" can be reshaped into an array, a table say, and the shape of the result is the same.

G2 =: 2 2 \$ G1	G2 APPEND 4 5
<pre> +-----+-----+  count sum    +-----+-----+  mean  count  +-----+-----+ </pre>	<pre> 2 9 4.5 2 </pre>

### 14.3.2 Gerund as Argument to Agenda Conjunction

Recall the `abs` verb defined above. Here is a reminder:

abs =: + ` - @. (< & 0)	abs 6	abs _6
+`-@.(<&0)	6	6

Here, the "Agenda" conjunction (@.) takes a verb on the right. As a variation, (@.) can also take a noun on the right. The noun consists of a list of numbers, which are indices selecting verbs from the gerund. The selected verbs form a train. This scheme gives us an abbreviation for the unpacking by indexing we saw above. The scheme is, for a gerund G and a list of indices I :

G @. I    means    (I { G) EV

For example:

G =: +`-`%	tr =: G @. 0 2	tr 4	(0 2 { G) EV 4
+--+--+  + - %  +--+--+	+ %	4.25	4.25

Next, we look at how to build trains with more structure. Consider the train T:

T =: * (- 1:)	T 3	T 4
* (- 1:)	6	12

which computes  $(T\ x) = x * (x - 1)$ . The parentheses mean that T is a hook where the second item is also a hook. Trains structured with parentheses in this way can be built with Agenda, by indexing items from a gerund, using boxed indices to indicate the parenthesisation.

foo =: (\* ` - ` 1:) @. (0 ; 1 2)

T	foo	foo 3
---	-----	-------

<code>* (- 1:)</code>	<code>* (- 1:)</code>	6
-----------------------	-----------------------	---

### 14.3.3 Gerund as Argument to Insert

We have previously encountered the insert adverb applied to a single verb: the verb is inserted between successive items of a list. More generally, when insert is applied to a gerund it inserts successive verbs from the gerund between successive items from the list. That is, if *G* is the gerund (*f`g`h`...*) and *x* is the list (*x0, x1, x2, x3, ...*) then

*G/X* means *x0 f x1 g x2 h x3 ...*

<code>ger =: + ` %</code>	<code>ger / 1 2 3</code>	<code>1 + 2 % 3</code>
<pre> +---+  + %  +---+ </pre>	1.66667	1.66667

If the gerund is too short, it is re-used cyclically to make up the needed number of verbs. This means that a one-verb gerund, when inserted, behaves the same as a single inserted verb.

### 14.3.4 Gerund as argument to POWER conjunction

Recall from Chapter 10 that the POWER conjunction (`^:`) can take, as right argument, a number which specifies the number of iterations of the verb given as left argument. As a brief reminder, 3 doublings of 1 is 8:

```
double =: +:
(double ^: 3) 1
8
```

As a variation, the number of iterations can be computed by a verb right-argument. The scheme is, for verbs *u* and *v*:

*(u ^: v) y* means *u ^: (v y) y*

For example:

```
decr =: <:
```

<code>double ^: (decr 3) 3</code>	<code>(double ^: decr) 3</code>
12	12

More generally, the right argument can be given as a gerund, and the verbs in it do some computations at the outset of the iteration process. The scheme is:

```
u ^: (v1 ` v2) y means u ^: (v1 y) (v2 y)
```

To illustrate, we define a verb to compute a Fibonacci sequence. Here each term is the sum of the preceding two terms. The verb will take an argument to specify the number of terms, so for example we want `FIB 6` to give `0 1 1 2 3 5`

The verb to be iterated, `u` say, generates the next sequence from the previous sequence by appending the sum of the last two. If we define:

```
u =: , sumlast2
sumlast2 =: +/ @ last2
last2 =: _2 & {.
```

then the iteration scheme beginning with the sequence `0 1` is shown by

<code>u 0 1</code>	<code>u u 0 1</code>	<code>u u u 0 1</code>
0 1 1	0 1 1 2	0 1 1 2 3

Now we define the two verbs of the gerund. We see that to produce a sequence with  $n$  terms the verb `u` must be applied  $(n-2)$  times, so the verb `v1`, which com



computes the number of iterations from the argument  $y$  is:

$v1 =: -\&2$

The verb  $v2$ , which computes the starting value from the argument  $y$ , we want to be the constant function which computes 0 1 whatever the value of  $y$ .

$v2 =: 3 : '0 1'$

Now we can put everything together:

$FIB =: u \wedge: (v1 \text{ ` } v2)$	$FIB \ 6$
$u \wedge: (v1 \text{ ` } v2)$	0 1 1 2 3 5

This example showed a monadic verb ( $u$ ) with the two verbs in the gerund ( $v1$  and  $v2$ ) performing some computations at the outset of the iteration. What about dyadic verbs?

Firstly, recall that with an iterated dyadic verb the left argument is bound at the outset to give a monad which is what is actually iterated, so that the scheme is:

$x \ u \wedge: k \ y$  means  $(x \&u) \wedge: k \ y$

Rather than constant  $k$ , we can perform pre-computations with three verbs  $U$   $V$  and  $W$  presented as a gerund. The scheme is:

$x \ u \wedge: (U \text{ ` } V \text{ ` } W) \ y$

means

$((x \ U \ y) \&u) \wedge: (x \ V \ y)) \ (x \ W \ y)$

Example to be supplied

The scheme above can also be written equivalently as a fork:

$u \wedge: (U \text{ ` } V \text{ ` } W)$  means  $U \ (u \wedge: V) \ W$

For example:

```
U =: [
V =: 2:
W =: ]
```

$p =: + \wedge: (U \setminus V \setminus W)$	3 p 4	$q =: U (+ \wedge: V) W$	3 q 4
$+ \wedge: (U \setminus V \setminus W)$	10	$U + \wedge: V W$	10

### 14.3.5 Gerund as Argument to Amend

Recall the "Amend" adverb from [Chapter 06 p6](#). The expression  $(new\ index\ } old)$  produces an amended version of *old*, having *new* as items at *index*. For example:

```
'o' 1 } 'baron'
boron
```

More generally, the "Amend" adverb can take an argument which is a gerund of three verbs, say  $U \setminus V \setminus W$ . The scheme is:

$x (U \setminus V \setminus W) \} y$  means  $(x\ U\ y) (x\ V\ y) \} (x\ W\ y)$

That is, the new items, the index(es) and the "old" array are all to be computed from the given *x* and *y*.

Here is an example (adapted from the Dictionary). Let us define a verb, *R* say, to amend a matrix by multiplying its *i*'th row by a constant *k*. The left argument of *R* is to be the list *i k* and the right argument is to be the original matrix. *R* is defined as the "Amend" adverb applied to a gerund of 3 verbs.

```
i =: { . @ [      NB. i = first of x
k =: { : @ [      NB. k = last of x
r =: i { ]        NB. i'th row
```

```
R =: ((k * r) ` i ` ]) }
```

For example:

```
M =: 3 2 $ 2 3 4 5 6 7
z =: 1 10      NB. row 1 times 10
```

z	M	z i M	z k M	z r M	z R M
1 10	2 3 4 5 6 7	1	10	4 5	2 3 40 50 6 7

# 14.4 Gerunds as Arguments to User-Defined Operators

Previous sections showed supplying gerunds to the built-in operators (adverbs or conjunctions). Now we look at defining our own operators taking gerunds as arguments. We begin with explicit operators and then go on to tacit operators.

The main consideration with an explicit operator is how to recover individual verbs from the gerund argument. We saw several possibilities above. Here is a simple one. Let `g Untie i` give the *i*'th verb in gerund *g*. We index *g* to get the *i*'th representation, and then apply adverb `EV` to turn the representation into a verb:

```
Untie =: 2 : '(y. { x.) EV'
```

<code>plus =: (*`-` +) Untie 2</code>	2 plus 3
<code>+</code>	5

Now for the operator. Let us define an adverb A, say, to produce a fork-like verb, so that `x (u`v`w A) y` is to mean `(u x) v (w y)`.

```
A =: 1 : 0
u =. x. Untie 0
v =. x. Untie 1
w =. x. Untie 2
((u @ [) v (w @ ])) f.
)
```

To illustrate A, here is a verb to join the first item of `x` to the last of `y`. The first and last items are yielded by the built-in verbs `{.` (left-brace dot, called "Head") and `{:` (left-brace colon, called "Tail").

<code>g =: {. ` , ` {:</code>	<code>zip =: g A</code>	<code>'abc' zip 'xyz'</code>
<pre> +---+---+  {.   ,  {:  +---+---+ </pre>	<code>{.@[ , {:@]</code>	<code>az</code>

## 14.4.1 The Abelson and Sussman Accumulator

Here is another example of a user-defined explicit operator with a gerund argument. Abelson and Sussman, (reference ...), describe how a variety of computations all conform to the following general plan, called the "accumulator":

Items from the argument (a list) are selected with a "filtering" function. For each selected item, a value is computed from it with a "mapping" function. The results of the separate mappings are combined into the overall result with a "combining" function. This plan can readily be implemented in J as an adverb, ACC say, as follows.

```
ACC =: 1 : 0
'com map fil' =. <"0 x.
((com EV /) @: (map EV) @: (#~ fil EV)) f.
)
```

ACC takes as argument a gerund of three verbs, in order, the combiner, the map and

the filter. For an example, we compute the sum of the squares of the odd numbers in a given list. Here the filter, to test for an odd number, is `(2&|)`

```
(+ ` *: ` (2&|)) ACC 1 2 3 4  
10
```

The first line of ACC splits up the gerund argument into three 1-item gerunds, `com`, `map` and `fil`. The boxing `(<"0)` is needed because the multiple assignment automatically strips off one layer of boxing. In the second line, `EV` is applied to each 1-item gerund to yield its verb.

This is the end of chapter 14.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 17 Au 2002

# Chapter 15: Tacit Operators

## 15.1 Introduction

J provides a number of built-in operators - adverbs and conjunctions. In previous chapters we looked at defining our own operators explicitly. In this chapter we look at defining adverbs tacitly.

## 15.2 Adverbs from Conjunctions

Recall from [Chapter 07 p7](#) the Rank conjunction, ( " ). For example, the verb ( < " 0 ) applies Box ( < ) to each rank-0 (scalar) item of the argument.

```
< " 0 'abc'
+---+---+
|a|b|c|
+---+---+
```

A conjunction takes two arguments. If we supply only one, the result is an adverb. For example, an adverb to apply a given verb to each scalar can be written as ( " 0 )

each =: " 0	< each	z =: < each 'abc'
"0	<"0	+---+---+  a b c  +---+---+

The scheme is, that for a conjunction C and a noun N, the expression ( C N ) denotes an adverb such that:

x ( C N ) means x C N

The argument to be supplied to the conjunction can be a noun or a verb, and on the left or on the right. Altogether there are four similar schemes:

$x (C N)$  means  $x C N$   
 $x (C V)$  means  $x C V$   
 $x (N C)$  means  $N C x$   
 $x (V C)$  means  $V C x$

The sequences  $CN$   $CV$   $NC$  and  $CV$  are called "bidents". They are a form of bonding (or currying) whereby we take a two-argument function and fix the value of one of its arguments to get a one-argument function. However, there is a difference between bonding a dyadic verb (as in  $+ \ \& \ 2$  for example) and bonding a conjunction. With the conjunction, there is no need for a bonding operator such as  $\&$ . We just write  $( \ " \ 0 )$  with no intervening operator. The reason is that in the case of  $+ \ \& \ 2$ , omitting the  $\&$  would give  $+ \ 2$  which means: apply the monadic case of  $+$  to 2, giving 2. However, conjunctions don't have monadic cases, so the bident  $( \ " \ 0 )$  is recognised as a bonding.

Recall the "Under" conjunction  $\&.$  from [Chapter 08 p8](#) whereby  $f\&.\ g$  is a verb which applies  $g$  to its argument, then  $f$  then the inverse of  $g$ . If we take  $f$  and  $g$  to be:

$f =: 'f' \ \& \ ,$   
 $g =: >$

then we see that  $f$  is applied inside each box:

z	$(f \ \&.\ g) \ z$
$+ - + - + - +$ $  a   b   c  $ $+ - + - + - +$	$+ - + - + - +$ $  f a   f b   f c  $ $+ - + - + - +$

Now, using the form  $CV$ , we can define an adverb  $EACH$  to mean "inside each box":

$EACH =: \&.\ >$	$f \ EACH$	z	$f \ EACH \ z$
------------------	------------	---	----------------

&.>	f&.>	+-+--+  a b c  +-+--+	+---+---+---+  fa fb fc  +---+---+---+
-----	------	-----------------------------	--

## 15.3 Compositions of Adverbs

If A and B are adverbs, then the bident (A B) denotes an adverb which applies A and then B. The scheme is:

$x (A B)$  means  $(x A) B$

### 15.3.1 Example: Cumulative Sums and Products

There is a built-in adverb \ (backslash). In the expression  $f \ y$  the verb  $f$  is applied to successively longer leading segments of  $y$ . For example:

```
< \ 'abc'
++-+-----+
|a|ab|abc|
++-+-----+
```

The expression  $+/\ y$  produces cumulative sums of  $y$ :

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

An adverb to produce cumulative sums, products, and so on can be written as a bident of two adverbs:

```
cum =: / \ NB. adverb adverb
```

z =: 2 3 4	+ cum z	* cum z
2 3 4	2 5 9	2 6 24



## 15.3.2 Generating Trains

Now we look at defining adverbs to generate trains of verbs, that is, hooks or forks.

First recall from [Chapter 14 p14](#) the Tie conjunction (```), which makes gerunds, and the Evoke Gerund adverb (`` : 6`) which makes trains from gerunds.

Now suppose that A and B are the adverbs:

```
A =: * `      NB. verb conjunction
B =: ` : 6     NB. conjunction noun
```

Then the compound adverb

```
H =: A B
```

is a hook-maker. Thus `< : H` generates the hook `* < :`, that is "x times x-1"

<code>&lt; : A</code>	<code>&lt; : A B</code>	<code>h =: &lt; : H</code>	<code>h 5</code>
<pre> +-+---+  * &lt;:  +-+---+ </pre>	<code>* &lt; :</code>	<code>* &lt; :</code>	20

## 15.3.3 Rewriting

It is possible to rewrite the definition of a verb to an equivalent form, by rearranging its terms. Suppose we start with a definition of the factorial function `f`. Factorial 5 is 120.

```
f =: (* ($: @: <:)) ` 1: @. (= 0:)
f 5
120
```

The idea now is to rewrite `f` to the form `$: adverb`, by a sequence of steps. Each step introduces a new adverb. The first new adverb is `A1`, which has the form `conj`

verb.

```
A1 =: @. (= 0:)
g  =: (* ($: @: <:)) ` 1: A1
g 5
120
```

Adverb A2 has the form conj verb

```
A2 =: ` 1:
h  =: (* ($: @: <:)) A2 A1
h 5
120
```

Adverb A3 has the form adv adv

```
A3 =: (* `) (`: 6)
i  =: ($: @: <:) A3 A2 A1
i 5
120
```

Adverb A4 has the form conj verb

```
A4=: @: <:
j  =: $: A4 A3 A2 A1
j 5
120
```

Combining A1 to A4:

```
A =: A4 A3 A2 A1
k =: $: A
k 5
120
```

Expanding A:

```
m =: $: (@: <:) (* `) (`: 6) (` 1:) (@. (= 0:))
m 5
120
```

We see that  $m$  and  $f$  are the same verb:

$f$	$m$
$( * \$ : @ : < : ) ` 1 : @ . ( = 0 : )$	$( * \$ : @ : < : ) ` 1 : @ . ( = 0 : )$

This is the end of Chapter 15.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 11 Aug 2002

# Chapter 16: Rearrangements

This chapter covers rearranging the items of arrays: permuting, sorting, transposing, reversing, rotating and shifting.

## 16.1 Permutations

A permutation of a vector is another vector which has all the items of the first but not necessarily in the same order. For example,  $z$  is a permutation of  $y$  where:

$y =: 'abcde'$	$z =: 4\ 2\ 3\ 1\ 0\ \{ y$
abcde	ecdba

The index vector  $4\ 2\ 3\ 1\ 0$  is itself a permutation of the indices  $0\ 1\ 2\ 3\ 4$ , that is,  $i. 5$ , and hence is said to be a permutation vector of order 5.

Notice the effect of this permutation: the first and last items are interchanged and the middle three rotate position amongst themselves. Hence this permutation can be described as a combination of cycling two items and cycling three items. After 6 ( $= 2 * 3$ ) applications of this permutation we return to the original vector.

$p =: 4\ 2\ 3\ 1\ 0\ \& \{$

$y$	$p\ y$	$p\ p\ y$	$p\ p\ p\ p\ p\ p\ y$
abcde	ecdba	adbce	abcde

The permutation  $4\ 2\ 3\ 1\ 0$  can be represented as a cycle of 2 and a cycle of 3. The verb to compute this cyclic representation is monadic  $c. .$

$c. 4\ 2\ 3\ 1\ 0$

```

+-----+-----+
| 3 1 2 | 4 0 |
+-----+-----+

```

Thus we have two representations of a permutation: ( 4 2 3 1 0 ) is called a direct representation and ( 3 1 2 ; 4 0 ) is called a cyclic representation. Monadic `c.` can accept either form and will produce the other form:

<code>c. 4 2 3 1 0</code>	<code>c. 3 1 2 ; 4 0</code>
<pre> +-----+-----+   3 1 2   4 0   +-----+-----+ </pre>	4 2 3 1 0

The dyadic verb `c.` can accept either form as its left argument, and permutes its right argument.

<code>y</code>	4 2 3 1 0 <code>c. y</code>	( 3 1 2 ; 4 0 ) <code>c. y</code>
abcde	ecdba	ecdba

## 16.1.1 Abbreviated Permutations

Dyadic `c.` can accept a left argument which is an abbreviation for a (direct) permutation vector. The effect is to move specified items to the tail, one at a time, in the order given.

<code>y</code>	2 <code>c. y</code>	2 3 <code>c. y</code>
abcde	abdec	abecd

With the abbreviated form, successive items are taken from the original vector: notice how the following two examples give different results.

y	2 3 C. y	3 C. (2 C. y)
abcde	abecd	abdce

If the left argument is boxed, then each box in turn is applied as a cycle:

y	(<3 1 2) C. y	(3 1 2 ; 4 0) C. y
abcde	acdbe	ecdba

If a is an abbreviated permutation vector (of order n) then the full-length equivalent of a is given by (a U n) where U is the utility function:

```
U =: 4 : 0
z =: y. | x.
((i. y.) -. z), z
)
```

For example, suppose the abbreviated permutation a is (1 3) then we see:

y	a =: 1 3	a C. y	f =: a U (#y)	f C. y
abcde	1 3	acebd	0 2 4 1 3	acebd

# 16.1.2 Inverse Permutation

If f is a full-length permutation vector, then the inverse permutation is given by (/ : f). (We will look at the verb / : in the next section.)

y	f	z =: f C. y	/ : f	(/ : f) C. z

abcde	0 2 4 1 3	acebd	0 3 1 4 2	abcde
-------	-----------	-------	-----------	-------

### 16.1.3 Atomic Representations of Permutations.

If  $y$  is a vector of length  $n$ , then there are altogether  $n!$  different permutations of  $y$ . A table of all permutations of order  $n$  can be generated by the expression `(tap n)` where `tap` is a utility verb defined by:

```
tap =: i. @ ! A. i.
tap 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

It can be seen that these permutations are in a well-defined order, and so any permutation of order  $n$  can be identified simply by its index in the table `(tap n)`. This index is called the atomic representation of the permutation. The monadic verb `A.` computes the atomic representation. For example, given an order-3 permutation, e.g. `2 1 0`, then `A. 2 1 0` yields the index in the table `(tap 3)`.

<code>A. 2 1 0</code>	<code>5 { tap 3</code>
<code>5</code>	<code>2 1 0</code>

Notice that `A.` gives its result as an extended integer (`5x`) rather than simply `5`. (Extended integers will be covered in [Chapter 19 p19](#).) The reason is that since the table `(tap n)` is of length  $n!$ , that is, potentially very long, indexes into it may need to be very long numbers.

The dyadic verb `A.` applies an atomic representation of a permutation.

<code>2 1 0 { 'PQR'</code>	<code>5 A. 'PQR'</code>
----------------------------	-------------------------

RQP	RQP
-----	-----

Here is an example of the use of `A..`. The process of running through all the permutations of something (say to search for anagrams of a word) might take a very long time. Hence it might be desirable to run through them say 100 at a time.

Here is a verb which finds a limited number of permutations. The argument is a boxed list: a vector to be permuted, followed by a starting permutation-number (that is, atomic index) followed by a count of the permutations to be found.

```
LPerms =: 3 : 0
'arg start count' =. y.
(start + i. count) A. " 0 1 arg
)
```

LPerms 'abcde'; 0; 4	LPerms 'abcde'; 4; 4
abcde abced abdce abdec	abecd abedc acbde acbed

# 16.2 Sorting

There is a built-in monad, `/:` (slash colon, called "Grade Up"). For a list `L`, the expression `(/: L)` gives a set of indices into `L`, and these indices are a permutation-vector.

L =: 'barn'	/: L
barn	1 0 3 2

These indices select the items of `L` in ascending order. That is, the expression `((/:`



`L) { L)` yields the items of `L` in order.

<code>L</code>	<code>/: L</code>	<code>(/: L) { L</code>
barn	1 0 3 2	abnr

For sorting into descending order, the monad `\:` (backslash colon, called "Grade Down") can be used.

<code>L</code>	<code>(\: L) { L</code>
barn	rnba

Since `L` is a character list, its items are sorted into alphabetical order. Numeric lists or boxed lists are sorted appropriately.

<code>N =: 3 1 4 5</code>	<code>(/: N) { N</code>
3 1 4 5	1 3 4 5

<code>B =: 'pooh'; 'bah'; 10; 5</code>	<code>(/: B) { B</code>
<pre> +-----+-----+-----+  pooh bah 10 5  +-----+-----+-----+ </pre>	<pre> +-----+-----+-----+  5 10 bah pooh  +-----+-----+-----+ </pre>

Now consider sorting the rows of a table. Here is an example of a table with 3 rows:

```

T =: (". ;. _2) 0 : 0
'WA' ; 'Mozart' ; 1756
'JS' ; 'Bach' ; 1685
'CPE' ; 'Bach' ; 1714
)
```

Suppose we aim to sort the rows of the table into order of date-of-birth shown in column 2 (the third column). We say that column 2 contains the keys on which the table is to be sorted.

We extract the keys with the verb `2&{ "1 T`, generate the permutation vector with `/:` applied to the keys, and then permute the table.

T	keys =: 2&{ "1 T	(/: keys) { T
+---+-----+---+  WA  Mozart 1756  +---+-----+---+  JS  Bach  1685  +---+-----+---+  CPE Bach  1714  +---+-----+---+	+---+-----+---+  1756 1685 1714  +---+-----+---+	+---+-----+---+  JS  Bach  1685  +---+-----+---+  CPE Bach  1714  +---+-----+---+  WA  Mozart 1756  +---+-----+---+

The dyadic case of `/:` allows the expression `(/: keys { T)` to be abbreviated as `(T /: keys)`.

(/: keys) { T	T /: keys
+---+-----+---+  JS  Bach  1685  +---+-----+---+  CPE Bach  1714  +---+-----+---+  WA  Mozart 1756  +---+-----+---+	+---+-----+---+  JS  Bach  1685  +---+-----+---+  CPE Bach  1714  +---+-----+---+  WA  Mozart 1756  +---+-----+---+

Suppose now we need to sort on two columns, say by last name, and then by initials. The keys are column 1 then column 0.

keys =: 1 0 & { " 1 T	T /: keys
-----------------------	-----------

+-----+---+	+---+-----+-----+
Mozart WA	CPE Bach 1714
+-----+---+	+---+-----+-----+
Bach JS	JS Bach 1685
+-----+---+	+---+-----+-----+
Bach CPE	WA Mozart 1756
+-----+---+	+---+-----+-----+

These examples show that the keys can be a table, and the `/:` verb yields the permutation-vector which puts the rows of the table into order. In such a case, the first column of the table is the most significant, then the second column, and so on.

## 16.2.1 Predefined Collating Sequences

Characters are sorted into "alphabetical order", numbers into "numerical order" and boxes into a well-defined order. The order for sorting all possible keys of a given type is called a collating sequence (for keys of that type). We have three predefined collating sequences. The collating sequence for characters is the ASCII character set. The built-in J noun `a.` gives the value of all 256 characters in "alphabetical" order. Note that upper-case letters come before lower-case letters.

```
65 66 67 97 98 99 { a.
ABCabc
```

With numerical arguments, complex numbers are ordered by the real part then the imaginary part.

n=: 0 1 _1 2j1 1j2 1j1	n /: n
0 1 _1 2j1 1j2 1j1	_1 0 1 1j1 1j2 2j1

With boxed arrays, the ordering is by the contents of each box. The precedence is firstly by type, with numerical arrays preceding empty arrays preceding character arrays preceding boxed arrays:

k=: (< 'abc') ; 'pqr' ; 4 ; ' ' ; 3	k /: k
<pre> +-----+---+---+---+  +---+ pqr 4  3    abc          +---+        +-----+---+---+ </pre>	<pre> +-+---+---+---+  3 4  pqr +---+              abc               +---+  +-+---+---+---+ </pre>

Within arrays of the same type, low-rank precedes high-rank.

m=: 2 4 ; 3 ; (1 1 \$ 1)	m /: m
<pre> +---+---+  2 4 3 1  +---+---+ </pre>	<pre> +-+---+---+  3 2 4 1  +-+---+---+ </pre>

Within arrays of the same type and rank, precedence depends on shape and content.  
If the two arrays are the same, then the earlier takes precedence (that is, their original order is not disturbed).

```

a =: 2 3 $ 1 2 3 4 5 6
b =: 3 2 $ 1 2 5 6 3 4
c =: 1 3 $ 1 2 3
d =: 1 3 $ 1 1 3

```

w=:a;b;c;d	w /: w
<pre> +-----+---+---+---+  1 2 3 1 2 1 2 3 1 1 3   4 5 6 5 6               3 4          +-----+---+---+ </pre>	<pre> +-----+---+---+---+  1 1 3 1 2 3 1 2 1 2 3       4 5 6 5 6               3 4      +-----+---+---+ </pre>

## 16.2.2 User-Defined Collating Sequences

The keys are computed from the data. By choosing how to compute the keys, we can choose a collating sequence.

For example, suppose a list of numbers is to be sorted into ascending order of absolute value. A suitable key-computing function would then be the "Magnitude" function, `|`.

<code>x=: 2 1 _3</code>	<code>keys =:   x</code>	<code>x /: keys</code>
2 1 _3	2 1 3	1 2 _3

## 16.3 Transpositions

The monadic verb `| :` will transpose a matrix, that is, interchange the first and second axes.

<code>M =: 2 3 \$ 'abcdef'</code>	<code>  : M</code>
abc def	ad be cf

More generally, `| :` will reverse the order of the axes of a n-dimensional array.

<code>N =: 2 2 2 \$ 'abcdefgh'</code>	<code>  : N</code>
ab cd  ef gh	ae cg  bf dh

Dyadic transpose will permute the axes according to the (full or abbreviated) permutation-vector given as left argument. For a 3-dimensional array, all possible permutations are given by (tap 3)

```
'A B C D E F' =: tap 3
```

N	A   : N	B   : N	C   : N	F   : N
ab cd	ab cd	ac bd	ab ef	ae cg
ef gh	ef gh	eg fh	cd gh	bf dh

A boxed abbreviated argument can be given. Two or more boxed axis-numbers are run together to form a single axis. With two dimensions, this is equivalent to taking the diagonal.

K =: i. 3 3	(< 0 1)   : K
0 1 2 3 4 5 6 7 8	0 4 8

# 16.4 Reversing, Rotating and Shifting

## 16.4.1 Reversing

Monadic |. will reverse the order of the items of its argument.

y	. y	M	. M

abcde	edcba	abc def	def abc
-------	-------	------------	------------

Notice that "reversing the items" means reversing along the first axis. Reversal along other axes can be achieved with the rank conjunction ( " ).

N	. N	. " 1 N	. " 2 N
ab cd	ef gh	ba dc	cd ab
ef gh	ab cd	fe hg	gh ef

## 16.4.2 Rotating

Dyadic | . rotates the items of  $y$  by an amount given by the argument  $x$ . A positive value for  $x$  rotates to the left.

$y$	$1 \mid . y$	$_{-1} \mid . y$
abcde	bcdea	eabcd

Successive numbers in  $x$  rotate  $y$  along successive axes:

M	$1 \ 2 \mid . M$	N	$1 \ 2 \mid . N$
abc def	fde cab	ab cd  ef gh	ef gh  ab cd

## 16.4.3 Shifting

The items which would be brought around by cyclic rotation can instead be replaced with a fill-item. A shifting verb is written ( | . ! . f ) where f is the fill-item.

```
ash  =: | . ! . '*'    NB. alphabetic shift
nsh  =: | . ! . 0      NB. numeric shift
```

y	_2 ash y	z =: 2 3 4	_1 nsh z
abcde	**abc	2 3 4	0 2 3

This is the end of Chapter 16

---

Copyright © Roger Stokes 1999. This material may be freely reproduced, provided that this copyright notice and provision is also reproduced.

last updated 10 September 1999



# Chapter 17: Patterns of Application

In this chapter we look at applying a function to an array in various patterns made up of selected elements of the array.

## 17.1 Scanning

### 17.1.1 Prefix Scanning

In the expression  $(f \ \backslash \ y)$  the result is produced by applying verb  $f$  to successively longer leading sections ("prefixes") of  $y$ . Choosing  $f$  as the box verb  $(<)$  gives easily visible results.

$y =: 'abcde'$	$< \ \backslash \ y$
abcde	<pre> +---+---+---+---+   a   ab   abc   abcd   abcde   +---+---+---+---+ </pre>

Cumulative sums of a numeric vector can be produced:

```

+/\ 0 1 2 3
0 1 3 6

```

Various effects can be produced by scanning bit-vectors. The following example shows "cumulative OR", which turns on all bits after the first 1-bit.

```

+./\ 0 1 0 1 0
0 1 1 1 1

```

### 17.1.2 Infix Scanning

In the expression  $(x \ f \ \backslash \ y)$  the verb  $f$  is applied to successive sections ("infixes") of  $y$ , each of length  $x$ .

<code>z =: 1 4 9 16</code>	<code>2 &lt; \ z</code>
<code>1 4 9 16</code>	<pre> +---+---+---+  1 4 4 9 9 16  +---+---+---+ </pre>

If  $x$  is negative, then the sections are non-overlapping, in which case the last section may not be full-length. For example:

<code>z</code>	<code>_3 &lt; \ z</code>
<code>1 4 9 16</code>	<pre> +-----+---+  1 4 9 16  +-----+---+ </pre>

We can compute the differences between successive items, by choosing 2 for the section-length, and applying to each section a verb "second-minus-first", that is,  $(\{ : - \{ . )$

<code>f =: { : - { .</code>	<code>f 1 4</code>
<code>{ : - { .</code>	<code>3</code>

`diff =: 2 & (f\)`

<code>,. z</code>	<code>,. diff z</code>	<code>,. diff diff z</code>

1	3	2
4	5	2
9	7	
16		

### 17.1.3 Suffix Scanning

In the expression  $(f \setminus \cdot y)$  the result is produced by applying  $f$  to successively shorter trailing sections ("suffixes") of  $y$ .

$y$	$< \setminus \cdot y$
abcde	<pre> +-----+-----+-----+-----+   abcde   bcde   cde   de   e   +-----+-----+-----+-----+ </pre>

### 17.1.4 Outfix

In the expression  $(x \ f \setminus \cdot y)$  the verb  $f$  is applied to the whole of  $y$  with successive sections removed, each removed section being of length  $x$ . If  $x$  is negative, then the removed sections are non-overlapping, in which case the last removed section may not be full-length.

$y$	$2 < \setminus \cdot y$	$_{-2} < \setminus \cdot y$
abcde	<pre> +---+---+---+---+   cde   ade   abe   abc   +---+---+---+---+ </pre>	<pre> +---+---+---+   cde   abe   abcd   +---+---+---+ </pre>

## 17.2 Cutting

The conjunction  $;$  (semicolon dot) is called "Cut". If  $u$  is a verb and  $n$  a small integer, then  $(u \ ; \cdot \ n)$  is a verb which applies  $u$  in various patterns as specified by  $n$ . The possible values for  $n$  are  $_{-3} \ _{-2} \ _{-1} \ 0 \ 1 \ 2 \ 3$ . We will look some but not all

of these cases.

## 17.2.1 Reversing

In the expression  $(u \ ;. \ 0 \ y)$ , the verb  $u$  is applied to  $y$  reversed along all axes. In the following example, we choose  $u$  to be the identity-verb ( $[]$ ).

<code>M =: 3 3 \$ 'abcdefghi'</code>	<code>[] ;. 0 M</code>
abc def ghi	ihg fed cba

## 17.2.2 Blocking

Given an array, we can pick out a smaller subarray inside it, and apply a verb to just the subarray.

The subarray is specified by a two-row table. In the first row is the index of the cell which will become the first of the subarray. In the second row is the shape of the subarray.

For example, to specify a subarray starting at row 1 column 1 of the original array, and of shape 2 2, we write:

```
spec =: 1 1 ,: 2 2
```

Then we can apply, say, the identity-verb ( $[]$ ) to the specified subarray as follows:

M	spec	spec [] ;. 0 M
abc def ghi	1 1 2 2	ef hi

The general scheme is that for a verb  $u$ , the expression  $(x\ u\ ;\ 0\ y)$  applies verb  $u$  to a subarray of  $y$  as specified by  $x$ .

In the specifier  $x$ , a negative value in the shape (the second row) will cause reversal of the elements of  $M$  along the corresponding axis. For example:

```
spec =: 1 1 ,: _2 2
```

M	spec	spec [ ;. 0 M
abc def ghi	1 1 _2 2	hi ef

17.2.3 Fretting

Suppose that we are interested in dividing a line of text into separate words. Here is an example of a line of text:

```
y =: 'what can be said'
```

For the moment, suppose we regard a word as being terminated by a space. (There are other possibilities, which we will come to.) Immediately we see that in  $y$  above, the last word 'said' is not followed by a space, so the first thing to do is to add a space at the end:

```
y =: y , ' '
```

Now if  $u$  is a verb, and  $y$  ends with a space, the expression  $(u\ ;\ _2\ y)$  will apply verb  $u$  separately to each space-terminated word in  $y$ . For example we can identify the words in  $y$  by applying  $<$ , the box function:

y	< ;. _2 y
---	-----------

what can be said	<pre> +-----+-----+-----+  what can be said  +-----+-----+-----+ </pre>
------------------	---

We can count the letters in each word by applying the # verb:

y	# ; . _2 y
what can be said	4 3 2 4

The meaning of \_2 for the right argument of ; . is that the words are to be terminated by occurrences of the last character in y (the space), and furthermore that the words do not include the spaces.

More generally, we say that a list may be divided into "intervals" marked by the occurrence of "frets". The right argument (n) of ; . specifies how we choose to define intervals and frets as follows. There are four cases.

n = 1 : Each interval begins with a fret. The first item of y is taken to be a fret, as are any other items of y equal to the first. Intervals include frets.

n = \_1 : As for (n = 1) except that intervals exclude frets.

n = 2 : Each interval ends with a fret. The last item of y is taken to be a fret, as are any other items of y equal to the last. Intervals include frets.

n = \_2 : As for (n = 2), except that intervals exclude frets.

For example, the four cases are shown by:

z =: 'abdacd'

z	< ; . 1 z	< ; . _1 z	< ; . 2 z	< ; . _2 z
---	-----------	------------	-----------	------------

abdacd	+---+---+  abd acd  +---+---+	+---+---+  bd cd  +---+---+	+---+---+  abd acd  +---+---+	+---+---+  ab ac  +---+---+
--------	-------------------------------------	-----------------------------------	-------------------------------------	-----------------------------------

For another example, here is a way of entering tables of numbers. We enter a table row by row following 0 : 0

```

T =: 0 : 0
1 2 3
4 5 6
19 20 21
)
```

T is a character-string with 3 embedded line-feed characters, one at the end of each line:

\$ T	+ / T = LF
30	3

The idea now is to cut T into lines. Each line is a character-string representing a J expression (for example the characters '1 2 3'). Such character-strings can be evaluated by applying the verb ". (double-quote dot, "Do" or "Execute"). The result is, for each line, a list of 3 numbers.

TABLE =: ( " . ; . _2 ) T	\$ TABLE
1 2 3 4 5 6 19 20 21	3 3

The verb ( " . ; . \_2 ) was introduced as the utility-function ArrayMaker in Chapter 2.

## 17.2.4 Punctuation

For processing text it would be useful to regard words as terminated by spaces or by various punctuation-marks. Suppose we choose our frets as any of four characters:

```
frets =: ' ?!.'
```

Given some text we can compute a bit-vector which is true at the location of a fret:

<code>t =: 'How are you?'</code>	<code>v =: t e. frets</code>
How are you?	0 0 0 1 0 0 0 1 0 0 0 1

Here we make use of the built-in verb `e.` ("Member"). The expression `x e. y` evaluates to true if `x` is a member of the list `y`.

Now the bitvector `v` can be used to specify the frets:

<code>t</code>	<code>v</code>	<code>v &lt; ;. _2 t</code>
How are you?	0 0 0 1 0 0 0 1 0 0 0 1	+---+---+---+  How are you  +---+---+---+

For another example, consider cutting a numeric vector into intervals such that each is in ascending sequence, that is, an item less than the previous must start a new interval. Suppose our data is:

```
data =: 3 1 4 1 5 9
```

Then a bitvector can be computed by scanning infixes of length 2, applying `>/` to



each pair of items. Where we get 1, the second item of the pair is the beginning of a new interval. We make sure the first item of all is 1.

```
bv =: 1 , 2 >/ \ data
```

data	data ,: bv	bv < ;. 1 data
3 1 4 1 5 9	3 1 4 1 5 9 1 1 0 1 0 0	<pre> +--+---+-----+  3 1 4 1 5 9  +--+---+-----+ </pre>

## 17.2.5 Word Formation

There is a built-in function `;` (semicolon colon, called "Word Formation"). It analyses a string as a J expression, according to the rules of the J language, to yield a boxed list of strings, the separate constituents of the J expression.

For example:

y =: 'z =: (p+q) - 1'	;	y
z =: (p+q) - 1		<pre> +--+---+---+---+---+---+---+  z =:( p + q )- 1  +--+---+---+---+---+---+---+ </pre>

## 17.2.6 Lines in Files

Let us begin by creating a file, to serve in the examples which follow. (See [Chapter 26 p25](#) for details of file-handling functions).

```

text =: 0 : 0
What can be said
at all
can be said
clearly.
)
```

```
text (1 !: 2) < 'c:\foo.txt'
```

Now, if we are interested in cutting a file of text into lines, we can read the file into a string-variable and cut the string. On the assumption that each line ends with a line-terminating character, then the last character in the file will be our fret. Here is an example.

```
string =: (1 !: 1) < 'c:\foo.txt'  NB. read the file

lines =: (< ;. _2) string          NB. cut into lines

lines
+-----+-----+-----+-----+
|What can be said|at all|can be said|clearly.|
+-----+-----+-----+-----+
```

There are two things to be aware of when cutting files of text into lines.

Firstly, in some systems lines in a file are terminated by a single line-feed character (LF). In other systems each line may be terminated by the pair of characters carriage-return (CR) followed by line-feed (LF).

J follows the convention of the single LF regardless of the system on which J is running. However, we should be prepared for CR characters to be present. To get rid of CR characters from string, we can reduce it with the bitvector (string notequal CR), where notequal is the built-in verb ~:, thus:

```
string =: (string ~: CR) # string
```

Secondly, depending on how the file of text was produced, we may not be able to guarantee that its last line is actually terminated. Thus we should be prepared to supply the fret character (LF) ourselves if necessary, by appending LF to the string.

A small function to tidy up a string, by supplying a fret and removing CR characters, can be written as:

```
tidy =: 3 : 0
y. =. y. , (LF ~: {: y.) # LF  NB. supply LF
```

```
(y. ~: CR) # y.                                NB. remove CR
)

(< ;. _2) tidy string
+-----+-----+-----+-----+
|What can be said|at all|can be said|clearly.|
+-----+-----+-----+-----+
```

## 17.2.7 Tiling

In the expression `(x u ;. 3 y)` the verb `u` is applied separately to each of a collection of subarrays extracted from `y`. These subarrays may be called tiles. The size and arrangement of the tiles are defined by the value of `x`. Here is an example. Suppose that `y` is

```
y =: 4 4 $ 'abcdefghijklmnop'
```

and our tiles are to be of shape `2 2`, each offset by 2 along each axis from its neighbour. That is, the offset is to be `2 2`. We specify the tiling with a table: the first row is the offset, the second the shape'

```
spec =: > 2 2 ; 2 2 NB. offset, shape
```

and so we see

y	spec	spec < ;. 3 y
abcd efgh ijkl mnop	2 2 2 2	+--+--+  ab cd   ef gh  +--+--+  ij kl   mn op  +--+--+

The specified tiling may leave incomplete pieces ("shards") at the edges. Shards

can be included or excluded by giving a right argument to "Cut" of 3 or \_3 .

```
sp =: > 3 3 ; 3 3
```

y	sp	sp < ;. 3 y	sp < ;. _3 y
abcd efgh ijkl mnop	3 3 3 3	+----++  abc d   efg h   ijk l  +----++  mno p  +----++	+----+  abc   efg   ijk  +----+

This is the end of Chapter 17.

---

Copyright © Roger Stokes 1999. This material may be freely reproduced, provided that this copyright notice and provision is also reproduced.

last updated 16 Mar 00

# Chapter 18: Sets, Classes and Relations

In this chapter we look at more of the built-in functions of J. The connecting theme is, somewhat loosely, working with set, classes and relations.

Suppose that, for some list, for the purpose at hand, the order of the items is irrelevant and the presence of duplicate items is irrelevant. Then we can regard the list as (representing) a finite set. In the abstract, the set 3 1 2 1 is considered to be the same set as 1 2 3. The word "class" we will use in the sense in which, for example, each integer in a list belongs either to the odd class or to the even class.

By "relation" is meant a table of two or more columns, expressing a relationship between a value in one column and the corresponding value in another. A relation with two columns, for example, is a set of pairs.

## 18.1 Sets

### 18.1.1 Membership

There is a built-in verb `e.` (lowercase e dot, called "Member"). The expression `x e. y` tests whether `x` matches any item of `y`, that is, whether `x` is a member of the list `y`. For example:

<code>y=: 'abcde'</code>	<code>'a' e. y</code>	<code>'w' e. y</code>	<code>'ef' e. y</code>
abcde	1	0	1 0

Evidently the order of items in `y` is irrelevant and so is the presence of duplicates in `y`.

z=: 'edcbad'	'a' e. z	'w' e. z	'ef' e. z
edcbad	1	0	1 0

We can test whether a table contains a particular row:

t =: 4 2 \$ 'abcdef'	'cd' e. t
ab cd ef ab	1

## 18.1.2 Less

There is a built-in verb `-.` (minus dot, called "Less"). The expression `x -. y` produces a list of the items of `x` except those which are members of `y`.

x =: 'consonant'	y =: 'aeiou'	x -. y
consonant	aeiou	cnsnnt

Evidently the order of items in `y` is irrelevant and so is the presence of duplicates in `y`.

## 18.1.3 Nub

There is a built-in verb `~.` (tilde dot, called "Nub"). The expression `~. y` produces a list of the items of `y` without duplicates.

nub =: ~.	y =: 'hook'	nub y
-----------	-------------	-------

<code>~.</code>	<code>hook</code>	<code>hok</code>
-----------------	-------------------	------------------

We can apply `nub` to the rows of a table:

<code>t</code>	<code>nub t</code>
<code>ab</code> <code>cd</code> <code>ef</code> <code>ab</code>	<code>ab</code> <code>cd</code> <code>ef</code>

### 18.1.4 Nub Sieve

The verb "nub sieve" (`~:`) gives a boolean vector which is true only at the nub.

<code>y</code>	<code>b =: ~: y</code>	<code>b # y</code>	<code>nub y</code>
<code>hook</code>	<code>1 1 0 1</code>	<code>hok</code>	<code>hok</code>

### 18.1.5 Functions for Sets

The customary functions on sets, such as set-union, set-intersection or set-equality, are easily defined using the built-in functions available. For example two sets are equal if all members of one are members of the other, and vice versa.

```
seteq =: *./ @: (e. , e.~)
```

<code>1 2 3 seteq 3 1 2 1</code>	<code>1 2 3 seteq 1 2</code>
<code>1</code>	<code>0</code>

## 18.2 The Table Adverb

Recall that the adverb `/` generates a verb; for example `+/` is a verb which sums lists. More precisely, it is the monadic case of `+/` which sums lists. The dyadic case of `+/` generates a table:

<code>x =: 0 1 2</code>	<code>y =: 3 4 5 6</code>	<code>z =: x +/ y</code>
0 1 2	3 4 5 6	3 4 5 6 4 5 6 7 5 6 7 8

The general scheme is that if we have

$$z =: x \text{ f } y$$

then `z` is a table such that the value at row `i` column `j` is given by applying `f` dyadically to the pair of arguments `i{x` and `j{y`. That is, `z` contains all possible pairings of an item of `x` with an item of `y`. Here is another example:

<code>x =: 'abc'</code>	<code>y =: 'face'</code>	<code>x =/ y</code>
abc	face	0 1 0 0 0 0 0 0 0 0 1 0

The result shows, in the first row, the value of `'a' = 'face'`, in the second row the value of `'b' = 'face'` and so on.

## 18.3 Classes

### 18.3.1 Self-Classify



Consider the problem of finding the counts of letters occurring in a string (the frequency-distribution of letters). Here is one approach.

We form a table testing each letter for equality with the nub.

y =: 'hook'	nub y	(nub y) =/ y
hook	hok	1 0 0 0 0 1 1 0 0 0 0 1

The expression  $((\text{nub } y) = / y)$  can be abbreviated as  $(= y)$ . The monadic case of the built-in verb  $=$  is called "Self-classify").

y	nub y	(nub y) =/ y	= y
hook	hok	1 0 0 0 0 1 1 0 0 0 0 1	1 0 0 0 0 1 1 0 0 0 0 1

If we sum each row of  $= y$  we obtain the counts, in the order of the letters in the nub.

y	= y	+/" 1 =y
hook	1 0 0 0 0 1 1 0 0 0 0 1	1 2 1

The counts can be paired with the letters of the nub:

y	nub y	(nub y) ;" 0 (+/" 1 =y)

hook	hok	<pre> +---+  h 1  +---+  o 2  +---+  k 1  +---+ </pre>
------	-----	--

## 18.3.2 Classification Schemes

Gardeners classify soil-types as acid, neutral or alkaline, depending on the pH value. Suppose that a pH less than 6 is classed as acid, 6 to 7 is neutral, and more than 7 as alkaline. Here now is a verb to classify a pH value, returning A for acid, N for neutral and L for alkaline (or limy).

```
classify =: ({ & 'ANL') @: ((>: & 6) + (> & 7))
```

<code>classify 6</code>	<code>classify 4.8 5.1 6 7 7.1 8</code>
N	AANNLL

The `classify` function we can regard as defining a classification scheme. The letters ANL, which are in effect names of classes, are called the keys of the scheme.

## 18.3.3 The Key Adverb

Given some data (a list, say), we can classify each item to produce a list of corresponding keys.

<code>data =: 7 5 6 4 8</code>	<code>k =: classify data</code>
7 5 6 4 8	NANAL

We can select and group together all the data in, say, class A (all the data with key A):

data	k	k = 'A'	(k = 'A') # data
7 5 6 4 8	NANAL	0 1 0 1 0	5 4

Now suppose we wish to count the items in each class. That is, we aim to apply the monadic verb # separately to each group of items all of the same key. To do this we can use the built-in adverb /. (slash dot, called "Key").

data	k =: classify data	k # /. data
7 5 6 4 8	NANAL	2 2 1

For another example, instead of counting the members we could exhibit the members, by applying the box verb <.

data	k =: classify data	k < /. data
7 5 6 4 8	NANAL	+---+---+--+   7 6   5 4   8   +---+---+--+

The verb we apply can discover for itself the class of each separate argument, by classifying the first member: Here the verb u produces a boxed list: the key and count:

```
u =: (classify @: {.) ; #
```

data	k =: classify data	k u /. data
------	--------------------	-------------

7 5 6 4 8	NANAL	+--+--+  N 2  +--+--+  A 2  +--+--+  L 1  +--+--+
-----------	-------	---

The general scheme for the "Key" adverb is as follows. In the expression `x u /. y`, we take `y` to be a list, and `x` is a list of keys of corresponding items of `y` according to some classification scheme, and `u` is the verb to be applied separately to each class. The scheme is:

`x u /. y` means `(= x) (u @ #) y`

To illustrate:

```

y =: 4 5 6 7 8
x =: classify y
u =: <

```

y	x	= x	(= x) (u @ #) y	x u /. y
4 5 6 7 8	AANNL	1 1 0 0 0 0 0 1 1 0 0 0 0 0 1	+----+----+--+  4 5 6 7 8  +----+----+--+	+----+----+--+  4 5 6 7 8  +----+----+--+

We see that each row of `=x` selects items from `y`, and `u` is applied to this selection.

### 18.3.4 Letter-Counts Revisited

Recall the example of finding the counts of letters in a string.

$y =: \text{'LETTUCE'}$	$= y$	$(\text{nub } y) ; \text{" 0 +/ "1 (= } y)$
LETTUCE	1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0	+--+--+  L 1  +--+--+  E 2  +--+--+  T 2  +--+--+  U 1  +--+--+  C 1  +--+--+

Here is a variation. We note that we have in effect a classification scheme where we have as many different classes as different letters: each letter is (the key of) its own class. Thus we can write an expression of the form  $y \text{ u } /. y$ .

The applied verb  $\text{u}$  will see, each time, a list of letters, all the same. It counts them, with  $\#$ , and takes the first, with  $\{ .$ , to be a label for the class.

$\text{u} =: \{ . ; \#$

$y$	$= y$	$y \text{ u } /. y$
LETTUCE	1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0	+--+--+  L 1  +--+--+  E 2  +--+--+  T 2  +--+--+  U 1  +--+--+  C 1  +--+--+

# 18.4 Relations

Suppose there are a number of publications, such as:

- "Pigs" by Smith, on the subject of pigs
- "Pets" by Brown, on cats and dogs
- "Dogs" by Smith and James, on dogs

and we aim to catalog such publications. A suitable data structure for such a catalog might be a table relating authors to titles and another table relating titles to subjects. For example:

author	title
Smith	"Pigs"
Brown	"Pets"
Smith	"Dogs"
James	"Dogs"

title
-------

subject

"Pigs"

pigs

"Pets"

dogs

"Pets"

cats

"Dogs"

dogs

Such tables we may call "relations". The order of the rows is not significant. Here, for the sake of simplicity, we will stick to relations with two columns.

Now we choose a representation for our relations. For a first approach, we choose tables of boxed strings. The authors-titles relation is:

```
] AT =: (". ;. _2) 0 : 0
'Smith' ; 'Pigs'
'Brown' ; 'Pets'
'Smith' ; 'Dogs'
'James' ; 'Dogs'
)
+-----+-----+
|Smith|Pigs|
+-----+-----+
|Brown|Pets|
+-----+-----+
|Smith|Dogs|
+-----+-----+
|James|Dogs|
+-----+-----+
```

and the titles-subjects relation is:

```
] TS =: (". ;. _2) 0 : 0
'Pigs' ; 'pigs'
'Pets' ; 'cats'
'Pets' ; 'dogs'
'Dogs' ; 'dogs'
)
+-----+-----+
|Pigs|pigs|
+-----+-----+
|Pets|cats|
+-----+-----+
|Pets|dogs|
+-----+-----+
|Dogs|dogs|
+-----+-----+
```

## 18.4.1 Join of Relations

From the authors-titles relation `AT` and the titles-subjects relation `TS` we can compute an authors-subjects relation showing which author has written a title on which subject. We say that `AT` and `TS` are to be joined with respect to titles, and we would expect the join to look like this:

```
+-----+-----+
|Smith|pigs|
+-----+-----+
|Brown|cats|
+-----+-----+
|Brown|dogs|
+-----+-----+
|Smith|dogs|
+-----+-----+
|James|dogs|
+-----+-----+
```

The plan for this section is to look at a function for computing joins, then at an improved version, and then at the advantage of representing relations as tables of symbols rather than boxed strings. Finally we look at some performance comparisons.

A method is as follows. We consider all possible pairs consisting of a row `x` from table `AT` and a row `y` from table `TS`. Each pair `x,y` is of the form:

```
author; title; title; subject
```

If title matches title, that is, item 1 matches item 2, then we extract author and subject, that is, items 0 and 3. Verbs for testing and extracting from `x,y` pairs can be written as:

```
test =: 1&{ = 2&{
extr =: 0 3 & {
```

and these verbs can be plugged into a suitable conjunction to do the pairing. In writing this conjunction, we aim to avoid requiring the whole set of possible pairs to be present at the same time, since this set may be large. We also aim to avoid any duplicates in the result. Here is a first attempt.

```
PAIR =: 2 : 0
```



```

:
z =. 0 0 $ ''
for_x. x. do.
  for_y. y. do.
    if. u. x,y do. z =. z, v. x,y end.
  end.
end.
~. z
)

```

The join verb can now be written as:

```
join =: test PAIR extr
```

and we see:

AT	TS	AT join TS
+-----+-----+  Smith Pigs  +-----+-----+  Brown Pets  +-----+-----+  Smith Dogs  +-----+-----+  James Dogs  +-----+-----+	+-----+-----+  Pigs pigs  +-----+-----+  Pets cats  +-----+-----+  Pets dogs  +-----+-----+  Dogs dogs  +-----+-----+	+-----+-----+  Smith pigs  +-----+-----+  Brown cats  +-----+-----+  Brown dogs  +-----+-----+  Smith dogs  +-----+-----+  James dogs  +-----+-----+

The join verb as defined above is slow, because the `test` and `extr` verbs are applied to a single `x,y` pair at a time - they are scalar computations. Performance will be better if we can give these verbs as much data as possible to work on at one time. (This is a universal rule in J). Vector or array arguments are better. Here is a revised vector-oriented version of `PAIR` and `join`, which still avoids building the entire set of pairs.

```

VPAIR =: 2 : 0
:
z =. 0 0 $ ''

```

```

for_x. x. do.
    z =. z , |: v. (#~"1 u.) |: x , "1 y.
end.
~. z
)

```

```
vjoin =: test VPAIR extr
```

giving the same result as before:

AT join TS	AT vjoin TS
+-----+-----+  Smith pigs  +-----+-----+  Brown cats  +-----+-----+  Brown dogs  +-----+-----+  Smith dogs  +-----+-----+  James dogs  +-----+-----+	+-----+-----+  Smith pigs  +-----+-----+  Brown cats  +-----+-----+  Brown dogs  +-----+-----+  Smith dogs  +-----+-----+  James dogs  +-----+-----+

Representing relations as tables of boxed strings, as above, is less than efficient. For a repeated value, the entire string is repeated. Values are compared by comparing entire strings.

Now we look at another possibility. Rather than boxed strings, a relation can be represented by a table of symbols.

## 18.4.2 What are Symbols?

Symbols are for efficient computation with string data. Symbols are a distinct data-type, in the same way that characters, boxes and numbers are distinct data-types. A symbol is a scalar which identifies, or refers to, a string.

A symbol can be created by applying the built-in verb `s:` (lowercase s colon) to a boxed string.

```
a =: s: <'hello'
```

Now the variable `a` has a value of type symbol. We inspect this value in the usual way:

```
a
`hello
```

and see that the value is displayed as the original string preceded by a left-quote. Even though `a` looks like a string when displayed, it is a scalar.

a	\$ a	# \$ a
`hello		0

The original string is stored in a data-structure, maintained automatically by the J system, called the symbol-table. Strings are not duplicated within the symbol-table. Hence if another symbol `b` is created from the same string as `a`, then `b` is equal to `a`.

a	b =: s: <'hello'	b = a
`hello	`hello	1

Notice that the comparison is simple scalar equality, with no need to compare the original strings.

Our relations above can be converted to arrays of symbols, and joined as before.

sAT =: s: AT	sTS =: s: TS	sAT vjoin sTS

`Smith `Pigs	`Pigs `pigs	`Smith `pigs
`Brown `Pets	`Pets `cats	`Brown `cats
`Smith `Dogs	`Pets `dogs	`Brown `dogs
`James `Dogs	`Dogs `dogs	`Smith `dogs
		`James `dogs

Symbols are lexicographically ordered to reflect the ordering of the original strings. Hence tables of symbols can be sorted:

sAT	/:~ sAT
`Smith `Pigs	`Brown `Pets
`Brown `Pets	`James `Dogs
`Smith `Dogs	`Smith `Dogs
`James `Dogs	`Smith `Pigs

### 18.4.3 Measurements Compared

Here is a utility verb giving time in seconds to evaluate an expression, averaged over say 4 executions.

```
time =: (8j5 & ":) @: (4 & (6!:2))
```

The examples of relations above are too small for meaningful performance measurements, so we make larger relations by replicating each say 100 times.

```
AT  =: 100 $ AT
TS  =: 100 $ TS
sAT =: 100 $ sAT
sTS =: 100 $ sTS
```

There are 4 cases to compare:

```
t1 =: time 'AT join TS'    NB. scalar method, boxed strings
t2 =: time 'sAT join sTS'  NB. scalar method, symbols
t3 =: time 'AT vjoin TS'   NB. vector method, boxed strings
t4 =: time 'sAT vjoin sTS' NB. vector method, symbols
```

and we see:

```
3 3 $ ' ' ; 'strings'; 'symbols'; 'scalar'; t1; t2; 'vector'; t3; t4
+-----+-----+-----+
|          | strings | symbols |
+-----+-----+-----+
| scalar | 10.76841 | 0.49555 |
+-----+-----+-----+
| vector | 0.17658 | 0.02611 |
+-----+-----+-----+
```

The built-in verb `7!:5` gives the size of its argument in bytes. A table of symbols is in itself smaller than the corresponding array of boxed strings.

<code>7!:5 &lt;'sat'</code>	<code>7!:5 &lt;'at'</code>
1024	13824

However, we must also take into account the size of the underlying "symbol table", which may be considerable. For more details, see the Dictionary.

This is the end of Chapter 18

---

# Chapter 19: Numbers

The topics covered in this chapter are:

- The different kinds of numbers available in J
- Special numbers (infinities and indeterminates)
- Notations for writing numbers
- How numbers are displayed

## 19.1 Six Different Kinds of Numbers

J supports computation with numbers of these kinds:

- booleans (or truth-values)
- integers
- real (or floating-point) numbers
- complex numbers
- extended integers (that is, arbitrarily large integers exactly represented)
- rationals (that is, pairs of extended integers)

Each kind of number has its own internal representation in memory. For example, an array containing only the truth-values 0 and 1 is stored in a compact internal form, called "boolean", rather than in the floating-point form. Similarly an array containing only (relatively small) whole numbers is stored in a compact form called "integer".

The choice of appropriate representation is managed entirely automatically by the J system, and is not normally something the programmer must be aware of.

However, there is a means of testing the representation of a number. Here is a utility function for the purpose.

```
types =: 'bool';'int';'float';'complex';'ext int';'rational'

type  =: > @: ({ & types) @: (1 4 8 16 64 128 & i.) @: (3 !:
```

0)

type 0=0	type 37	type 2.5	type 12345678901
bool	int	float	float

## 19.1.1 Booleans

There are built-in functions for logical computation with boolean values. Giving conventional names to these functions:

```
and      =: *.
or       =: +.
not      =: -.
notand   =: *:
notor    =: +:
```

we can show their truth-tables:

```
p =: 4 1 $ 0 0 1 1
q =: 4 1 $ 0 1 0 1
```

p	q	p and q	p or q	not p	p notand q
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

Further logical functions can be defined in the usual way. For example, logical implication, with the scheme

p implies q      means      not (p and not q)

is defined by `not` composed with the `hook` and `not`

```
imp =: not @ (and not)
```

p	q	p imp q
0	0	1
0	1	1
1	0	0
1	1	1

We regard the booleans as numbers because they can be interpreted as having arithmetic values. To illustrate, implication has the same truth-table as less-than-or-equal:

p imp q	p <: q
1	1
1	1
0	0
1	1

For another example of booleans as numbers, the sum of the positive numbers in a list is shown by:

z =: 3 _1 4	b =: z > 0	b * z	+ / b * z
3 _1 4	1 0 1	3 0 4	7

## 19.1.2 Integers



On a 32-bit machine integers range between `_2147483648` and `2147483647`.

The result of arithmetic with integers is converted to floating-point if larger than the maximum integer.

<code>maxint=:2147483647</code>	<code>type maxint</code>	<code>z =: 1+maxint</code>	<code>type z</code>
2147483647	int	2.14748e9	float

### 19.1.3 Floating-Point Numbers

A floating-point number is a number represented in the computer in such a way that: (1) there may be a fractional part as well as a whole-number part. (2) a fixed amount of computer storage is occupied by the number, whatever the value of the number. and therefore (3) the precision with which the number is represented is limited to at most about 17 significant decimal digits (on a PC).

Examples of floating-point numbers are `0.25` `2.5` `12345678901`

We will use the term "real" more or less interchangeably with "floating-point".

### 19.1.4 Scientific Notation

What is sometimes called "scientific notation" is a convenient way of writing very large or very small numbers. For example, 1500000 may be written as `1.5e6`, meaning  $1.5 * 10^6$ . The general scheme is that a number written in the form `x eY`, where Y is a (positive or negative) integer means  $(X * 10^Y)$ .

<code>3e2</code>	<code>1.5e6</code>	<code>1.5e_4</code>
300	1500000	0.00015

Note that in `3e2` the letter e is not any kind of function; it is part of the notation for writing numbers, just as a decimal point is part of the notation.

We say that the string of characters 3 followed by e followed by 2 is a numeral which denotes the number 300. The string of characters 3 followed by 0 followed by 0 is another numeral denoting the same number. Different forms of numerals provide convenient ways to express different numbers. A number expressed by a numeral is also called a "constant" (as opposed to a variable.)

We will come back to the topic of numerals: now we return to the topic of different kinds of numbers.

## 19.1.5 Comparison of Floating-Point Numbers

Two numbers are regarded as equal if their difference is relatively small. For example, we see that a and b have a non-zero difference, but even so the expression `a = b` produces "true".

<code>a =: 1.001</code>	<code>b =: a - 2^_45</code>	<code>a - b</code>	<code>a = b</code>
1.001	1.001	2.84217e_14	1

If we say that the "relative difference" of two numbers is the magnitude of the difference divided by the magnitude of the larger,

`RD =: ( | @: - ) % ( | @: > . )`

then for `a=b` to be true, the relative difference (`a RD b`) must not exceed a small value called the "comparison tolerance" which is by default `2^_44`

<code>a RD b</code>	<code>2^_44</code>	<code>a = b</code>
2.83933e_14	5.68434e_14	1

Thus to compare two numbers we need to compare relative difference with tolerance. The latter comparison is itself strict, that is, does not involve any tolerance.

Zero is not tolerantly equal to any non-zero number, no matter how small, because the relative difference must be 1, and thus greater than tolerance.

<code>tiny =: 1e_300</code>	<code>tiny = 0</code>	<code>tiny RD 0</code>
1e_300	0	1

However, `1+tiny` is tolerantly equal to 1.

<code>tiny</code>	<code>tiny = 0</code>	<code>1 = tiny + 1</code>
1e_300	0	1

The value of the comparison tolerance currently in effect is given by the built-in verb `9!:18` applied to a null argument. It is currently  $2^{44}$ .

<code>9!:18 ''</code>	$2^{44}$
5.68434e_14	5.68434e_14

Applying the built-in verb `9!:19` to an argument `y` sets the tolerance to `y` subsequently. The following example shows that when the tolerance is  $2^{44}$ , then `a = b` but when the tolerance is set to zero it is no longer the case that `a = b`.

<code>(9!:19) 2^44</code>	<code>a = b</code>	<code>(9!:19) 0</code>	<code>a = b</code>
	1		0

The tolerance queried by `9!:18` and set by `9!:19` is a global parameter, influencing the outcome of computations with `=`. A verb to apply a specified

tolerance  $\epsilon$ , regardless of the global parameter, can be written as  $\epsilon = 10^{-t}$ . For example, strict (zero-tolerance) equality can be defined by:

```
streq =: = !. 0
```

Resetting the global tolerance to the default value, we see:

<code>(9!:19) 2^_44</code>	<code>a - b</code>	<code>a = b</code>	<code>a streq b</code>
	2.84217e_14	1	0

Comparison with `=` is tolerant, and so are comparisons with `<`, `<:`, `>`, `>:`, `~:` and `-:`. For example, the difference `a-b` is positive but too small to make it true that `a>b`

<code>a - b</code>	<code>a &gt; b</code>
2.84217e_14	0

Permissible tolerances range between 0 and  $2^{-35}$ . That is, an attempt to set the tolerance larger than  $2^{-35}$  is an error:

<code>(9!:19) 2^_35</code>	<code>(9!:19) 2^_34</code>
	error

The effect of disallowing large tolerances is that no two different integers compare equal when converted to floating-point.

## 19.1.6 Complex Numbers

The square root of -1 is the imaginary number conventionally called "i". A

complex number which is conventionally written as, for example,  $3+i4$  is in J written as `3 j 4`.

In J an imaginary number is always regarded as a complex number with real part zero. Thus "i", the square root of -1, can be written `0 j 1`.

<code>i =: %: _1</code>	<code>i * i</code>	<code>0 j 1 * 0 j 1</code>
<code>0 j 1</code>	<code>_1</code>	<code>_1</code>

A complex number can be built from two separate real numbers by arithmetic in the ordinary way, or more conveniently with the built-in function `j .` (lowercase j dot, called "Complex").

<code>3 + 0 j 1 * 4</code>	<code>3 j . 4</code>
<code>3 j 4</code>	<code>3 j 4</code>

A complex number such as `3 j 4` is a single number, a scalar. To extract its real part and imaginary part separately we can use the built-in verb `+.`  (plus dot, called "Real/Imaginary"). To extract separately the magnitude and angle (in radians) we can use the built-in verb `*.`  (asterisk dot, called "Length/Angle").

<code>+. 3 j 4</code>	<code>*. 3 j 4</code>
<code>3 4</code>	<code>5 0.927295</code>

Given a magnitude and angle, we can build a complex number by taking sine and cosine, or more conveniently with the built-in function `r .` (lowercase r dot, called "Polar").

```
sin =: 1 & o.
cos =: 2 & o.
mag =: 5
ang =: 0.92729522 NB. radians
```

<code>mag * (cos ang) + 0j1 * sin ang</code>	<code>mag r. ang</code>
<code>3j4</code>	<code>3j4</code>

A complex constant with magnitude  $x$  and angle (in radians)  $y$  can be written in the form  $x\text{r}y$ , meaning  $x \cdot y$ . Similarly, if the angle is given in degrees, we can write  $x\text{d}y$ .

<code>5ar0.9272952</code>	<code>5ad53.1301</code>
<code>3j4</code>	<code>3j4</code>

### 19.1.7 Extended Integers

A floating-point number, having a limited storage space in the computer's memory, can represent an integer exactly only up to about 17 digits. For exact computations with longer numbers, "extended integers" are available. An "extended integer" is a number which exactly represents an integer no matter how many digits are needed. An extended integer is written with the digits followed with the letter 'x'. Compare the following:

<code>a =: *: 100000000001</code>	<code>b =: *: 100000000001x</code>
<code>1e20</code>	<code>10000000000200000000001</code>

Here  $a$  is an approximation while  $b$  is an exact result.

<code>type a</code>	<code>type b</code>
---------------------	---------------------

float	ext int
-------	---------

We can see that adding 1 to a makes no difference, while adding 1 to b does make a difference:

$(a + 1) - a$	$(b + 1) - b$
0	1

### 19.1.8 Rational Numbers

A "rational number" is a single number which represents exactly the ratio of two integers, for example, two-thirds is the ratio of 2 to 3. Two-thirds can be written as a rational number with the notation `2r3`.

The point of rationals is that they are exact representations using extended integers. Arithmetic with rationals gives exact results.

$2r3 + 1r7$	$2r3 * 4r7$	$2r3 \% 5r7$
<code>17r21</code>	<code>8r21</code>	<code>14r15</code>

Rationals can be constructed by dividing extended integers. Compare the following:

$2 \% 3$	$2x \% 3x$
<code>0.666667</code>	<code>2r3</code>

A rational can be constructed from a given floating-point number with the verb `x`:

<code>x: 0.3</code>	<code>x: 1 % 3</code>
<code>3r10</code>	<code>1r3</code>

A rational number can be converted to a floating-point approximation with the inverse of `x:`, that is, verb `x: ^: _1`

<code>float =: x: ^: _1</code>	<code>float 2r3</code>
<pre> +---+---+---+  x: ^: _1  +---+---+---+ </pre>	0.666667

Given a rational number, its numerator and denominator can be recovered with the verb `2 & x:`, which gives a list of length 2.

<code>nd =: 2 &amp; x:</code>	<code>nd 2r3</code>
<pre> +---+---+  2 &amp; x:  +---+---+ </pre>	2 3

## 19.1.9 Type Conversion

We have numbers of six different types: boolean, integer, extended integer, rational, floating-point and complex.

Arithmetic can be done with a mixture of types. For example an integer plus an extended gives an extended, and a rational times a float gives a float.

<code>1 + 10^19x</code>	<code>1r3 * 0.75</code>
-------------------------	-------------------------





## 19.2.2 Indeterminate Numbers

Infinity is equal to infinity. However, infinity minus infinity is not zero but rather a special number called "indeterminate", written as  $\_.$  (underscore dot)

$\_ = \_$	$\_ - \_$
1	$\_.$

Indeterminate is equal to indeterminate. However indeterminate minus indeterminate is not zero but indeterminate.

$\_.\_ = \_.\_$	$\_.\_ - \_.\_$
1	$\_.$

Computations with indeterminate may produce surprising results. Thus indeterminate is not to be regarded as extending the meaning of "number" according to some extended axiomatization of arithmetic. Rather, the view is recommended that the only purpose of indeterminate is as a signal of numerical error when it occurs in the result of a computation.

Infinity is equal to itself and nothing else, and the same is true of indeterminate. Thus we have a reliable (that is, determinate) test for occurrences.

$z =: 1 \_ \_.$	$z = 1$	$z = \_$	$z = \_.$
1 $\_ \_.$	1 0 0	0 1 0	0 0 1

## 19.3 Notations for Numerals

We have seen above numerals formed with the letters e, r and j, for example:  $1e3$ ,

2r3, and 3j4. Here we look at more letters for forming numerals.

A numeral written with letter p, of the form  $x_p y$  means  $x * \pi^y$  where  $\pi$  is the familiar value 3.14159265....

$\pi =: 1p1$	$2\pi =: 2p1$	$2p\_1$
3.14159	6.28319	0.63662

Similarly, a numeral written with letter x, of the form  $x_x y$  means  $x * e^y$  where  $e$  is the familiar value 2.718281828....

$e =: 1x1$	$2x\_1$	$2 * e^{\_1}$
2.71828	0.735759	0.735759

These p and x forms of numeral provide a convenient way of writing constants accurately without writing out many digits.

Finally, we can write numerals with a base other than 10. For example the binary or base-2 number with binary digits 101 has the value 5 and can be written as 2b101.

2b101  
5

The general scheme is that  $N_b DDD.DDD$  is a numeral in number-base  $N$  with digits  $DDD.DDD$ . With bases larger than 10, we will need digits larger than 9, so we take letter 'a' as a digit with value 10, 'b' with value 11, and so on up to 'z' with value 35.

For example, letter 'f' has digit-value 15, so in hexadecimal (base 16) the numeral written 16bff has the value 255. The number-base  $N$  is given in decimal.

16bff	$(16 * 15) + 15$
255	255

One more example. 10b0.9 is evidently a base-10 number meaning "nine-tenths" and so, in base 20, 20b0.f means "fifteen twentieths"

10b0.9 20b0.f  
0.9 0.75

### 19.3.1 Combining the Notations

The notation-letters e, r, j and p x and b may be used in combination. For example we can write 1r2p1 to mean "pi over two". Here are some further examples of possible combinations.

A numeral in the form x<sub>r</sub>Y denotes the number x%Y. A numeral in the form x<sub>e</sub>Y<sub>r</sub>Z denotes the number (x<sub>e</sub>Y) % Z because e is considered before r.

1.2e2	$(1.2e2) \% 4$	1.2e2r4
120	30	30

A numeral in the form x<sub>j</sub>Y denotes the complex number (x<sub>j</sub>. Y) (that is, (x + (%: \_1) \* Y)). A numeral in the form x<sub>r</sub>Y<sub>j</sub>Z denotes the number (x<sub>r</sub>Y) <sub>j</sub>. Z because r is considered before j

3r4	$(3r4) \text{ j. } 5$	3r4j5
3r4	0.75j5	0.75j5

A numeral in the form x<sub>p</sub>Y denotes the number x\*pi^Y. A numeral in the form x<sub>j</sub>Y<sub>p</sub>Z denotes (x<sub>j</sub>Y) \*pi^Z because j is considered before p.

<code>3j4p5</code>	<code>(3j4) * pi ^ 5</code>
<code>918.059j1224.08</code>	<code>918.059j1224.08</code>

A numeral in the form `xbY` denotes the number *Y-in-base-x*. A numeral in the form `xpYbZ` denotes the number *Z-in-base-(XpY)* because `p` is considered before `b`.

<code>(3*pi)+5</code>	<code>1p1b35</code>
<code>14.4248</code>	<code>14.4248</code>

## 19.4 How Numbers are Displayed

A number is displayed by J with, by default, up to 6 or 7 significant digits. This means that, commonly, small integers are shown exactly, while large numbers, or numbers with many significant digits, are shown approximately.

<code>10 ^ 3</code>	<code>2.7182818285</code>	<code>2.718281828 * 10 ^ 7</code>
<code>1000</code>	<code>2.71828</code>	<code>2.71828e7</code>

The number of significant digits used for display is determined by a global variable called the "print-precision". If we define the two functions:

```
ppq =: 9 !: 10    NB. print-precision query
pps =: 9 !: 11    NB. print-precision set
```

then the expression `ppq` gives the value of print-precision currently in effect, while `pps n` will set the print-precision to `n`.

ppq ''	e =: 2.718281828	pps 8	e
6	2.71828		2.7182818

## 19.4.1 The "Format" Verb

There is a built-in verb `" :` (doublequote colon, called "Format"). Monadic Format converts a number into a string representing the number with the print-precision currently in effect. In the following example, note that `a` is a scalar, while the formatted representation of `a` is a list of characters.

a =: 1 % 3	" : a	\$ " : a
0.33333333	0.33333333	10

The argument can be a list of numbers and the result is a single string.

b =: 1 % 3 4	" : b	\$ b	\$ " : b
0.33333333 0.25	0.33333333 0.25	2	15

Dyadic Format allows more control over the representation. The left argument is complex: a value of say, `8j4` will format the numbers in a width of 8 characters and with 4 decimal places.

c =: % 1 + i. 2 2	w =: 8j4 " : c	\$ w
1 0.5 0.33333333 0.25	1.0000 0.5000 0.3333 0.2500	2 16

If the width is specified as zero (as in say `0j3`) then sufficient width is allowed. If

the number of decimal places is negative (as in 10j\_3) then numbers are shown in "scientific notation"

c	0j3 ": c	10j_3 ": c
1 0.5 0.33333333 0.25	1.000 0.500 0.333 0.250	1.000e0 5.000e_1 3.333e_1 2.500e_1

This brings us to the end of Chapter 19.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 15 Mar 2002

# Chapter 20: Scalar Numerical Functions

In this chapter we look at built-in scalar functions for computing numbers from numbers. This chapter is a straight catalog of functions, with links to the sections as follows:

Ceiling Conjugate  $\cos$   $\cos^{-1}$  cosh  $\cosh^{-1}$

Decrement divide Double Exponential Factorial Floor

GCD Halve Increment LCM Logarithm Log, Natural

Magnitude Minus multiply Negate OutOf PiTimes

Plus power Pythagorean Reciprocal Residue Root

Signum  $\sin$   $\sin^{-1}$  sinh  $\sinh^{-1}$  Square

SquareRoot  $\tan$   $\tan^{-1}$  tanh  $\tanh^{-1}$

## 20.1 Numbers from Numbers

### 20.1.1 Plus and Conjugate

Dyadic + is arithmetic addition.

$2 + 2$	$3j4 + 5j4$	$2r3 + 1r6$
4	$8j8$	$5r6$



Monadic + is "Conjugate". For a real number  $y$ , the conjugate is  $y$ . For a complex number  $x+jy$  (that is,  $x + 0jy$ ), the conjugate is  $x - 0jy$ .

+ 2	+ 3j4
2	3j_4

## 20.1.2 Minus and Negate

Dyadic - is arithmetic subtraction.

2 - 2	3 - 0j4	2r3 - 1r6
0	3j_4	1r2

Monadic - is "Negate".

- 2	- 3j4
_2	_3j_4

## 20.1.3 Increment and Decrement

Monadic >: is called "Increment". It adds 1 to its argument.

>: 2	>: 2.5	>: 2r3	>: 2j3
3	3.5	5r3	3j3

Monadic <: is called "Decrement". It subtracts 1 from its argument.

$\angle: 3$	$\angle: 2.5$	$\angle: 2r3$	$\angle: 2j3$
2	1.5	$\_1r3$	$1j3$

## 20.1.4 Times and Signum

Dyadic  $*$  is multiplication.

$2 * 3$	$3j1 * 2j2$
6	$4j8$

Monadic  $*$  is called "Signum". For a real number  $y$ , the value of  $( * y)$  is  $\_1$  or 0 or 1 as  $y$  is negative, zero or positive.

$* \_2$	$* 0$	$* 2$
$\_1$	0	1

More generally,  $y$  may be real or complex, and the signum is equivalent to  $y \% | y$ . Hence the signum of a complex number has magnitude 1 and the same angle as the argument.

$y =: 3j4$	$  y$	$y \%   y$	$* y$	$  * y$
$3j4$	5	$0.6j0.8$	$0.6j0.8$	1

## 20.1.5 Division and Reciprocal

Dyadic  $\%$  is division.

$2 \% 3$	$3j4 \% 2j1$	$12x \% 5x$
0.666667	$2j1$	$12r5$

$1 \% 0$  is "infinity" but  $0 \% 0$  is 0

$1 \% 0$	$0 \% 0$
—	0

Monadic  $\%$  is the "reciprocal" function.

$\% 2$	$\% 0j1$
0.5	$0j\_1$

## 20.1.6 Double and Halve

Monadic  $+:$  is the "double" verb.

$+: 2.5$	$+: 3j4$	$+: 3x$
5	$6j8$	6

Monadic  $-:$  is the "halve" verb:

$-: 6$	$-: 6.5$	$-: 3j4$	$-: 3x$
3	3.25	$1.5j2$	$3r2$

## 20.1.7 Floor and Ceiling

Monadic  $<.$  (left-angle-bracket dot) is called "Floor". For real  $y$  the floor of  $y$  is  $y$  rounded downwards to an integer, that is, the largest integer not exceeding  $y$ .

$<. 2$	$<. 3.2$	$<. \_3.2$
2	3	$\_4$

For complex  $y$ , the floor lies within a unit circle center  $y$ , that is, the magnitude of  $(y - <. y)$  is less than 1.

$y =: 3.4j3.4$	$z =: <. y$	$y - z$	$  y-z$
$3.4j3.4$	$3j3$	$0.4j0.4$	0.565685

This condition (magnitude less than 1) means that the floor of say  $3.8j3.8$  is not  $3j3$  but  $4j3$  because  $3j3$  does not satisfy the condition.

$y =: 3.8j3.8$	$z =: <. y$	$  y-z$	$  y - 3j3$
$3.8j3.8$	$4j3$	0.824621	1.13137

Monadic  $>.$  is called "Ceiling". For real  $y$  the ceiling of  $y$  is  $y$  rounded upwards to an integer, that is, the smallest integer greater than or equal to  $y$ . For example:

$>. 3.0$	$>. 3.1$	$>. \_2.5$
3	4	$\_2$

Ceiling applies to complex  $y$

<code>&gt;. 3.4j3.4</code>	<code>&gt;. 3.8j3.8</code>
<code>3j4</code>	<code>4j4</code>

## 20.1.8 Power and Exponentiation

Dyadic  $\wedge$  is the "power" verb:  $(x^y)$  is  $x$  raised-to-the-power  $y$

<code>10 ^ 2</code>	<code>10 ^ _2</code>	<code>100 ^ 1%2</code>
<code>100</code>	<code>0.01</code>	<code>10</code>

Monadic  $\wedge$  is exponentiation (or antilogarithm):  $\wedge y$  means  $(e^y)$  where  $e$  is Euler's constant, 2.71828...

<code>^ 1</code>	<code>^ 0j1</code>
<code>2.71828</code>	<code>0.540302j0.841471</code>

Euler's equation, supposedly engraved on his tombstone is:  $e^{i\pi} + 1 = 0$   
`(^ 0j1p1) + 1`  
`0`

## 20.1.9 Square

Monadic  $*:$  is "Square".

<code>*: 4</code>	<code>*: 2j1</code>
-------------------	---------------------

16	3j4
----	-----

### 20.1.10 Square Root

Monadic  $\%:$  is "Square Root".

$\%: 9$	$\%: 3j4$	$2j1 * 2j1$
3	2j1	3j4

### 20.1.11 Root

If  $x$  is integral, then  $x \%: y$  is the "x'th root" of  $y$ :

$3 \%: 8$	$\_3 \%: 8$
2	0.5

More generally,  $(x \%: y)$  is an abbreviation for  $(y \wedge \% x)$

$x =: 3 \ 3.1$	$x \%: 8$	$8 \wedge \% x$
3 3.1	2 1.95578	2 1.95578

### 20.1.12 Logarithm and Natural Logarithm

Dyadic  $\wedge.$  is the base-x logarithm function, that is,  $(x \wedge. y)$  is the logarithm of  $y$  to base  $x$ :

$10 \wedge. 1000$	$2 \wedge. 8$
-------------------	---------------

3	3
---	---

Monadic  $\wedge.$  is the "natural logarithm" function.

<code>e =: ^ 1</code>	<code>^ . e</code>
2.71828	1

### 20.1.13 Factorial and OutOf

The factorial function is monadic  $!$ .

<code>! 0 1 2 3 4</code>	<code>! 5x 6x 7x 8x</code>
1 1 2 6 24	120 720 5040 40320

The number of combinations of  $x$  objects selected out of  $y$  objects is given by the expression  $x ! y$

<code>1 ! 4</code>	<code>2 ! 4</code>	<code>3 ! 4</code>
4	6	4

### 20.1.14 Magnitude and Residue

Monadic  $|$  is called "Magnitude". For a real number  $y$  the magnitude of  $y$  is the absolute value:

<code>  2</code>	<code>  _2</code>
------------------	-------------------

2	2
---	---

More generally,  $y$  may be real or complex, and the magnitude is equivalent to  $(\%: y^* + y)$ .

$y =: 3j4$	$y^* + y$	$\%: y^* + y$	$  y$
3j4	25	5	5

The dyadic verb  $|$  is called "residue". the remainder when  $y$  is divided by  $x$  is given by  $(x | y)$ .

$10   12$	$3   \_2 \_1 0 1 2 3 4 5$	$1.5   3.7$
2	1 2 0 1 2 0 1 2	0.7

If  $x | y$  is zero, then  $x$  is a divisor of  $y$ :

$4   12$	$12 \% 4$
0	3

The residue function applies to complex numbers:

$a =: 1j2$	$b =: 2j3$	$a   b$	$a   (a*b)$	$(b-1j1) \% a$
1j2	2j3	1j1	0	1

## 20.1.15 GCD and LCM



The greatest common divisor (GCD) of  $x$  and  $y$  is given by  $(x +. y)$ . Reals and rationals in the domain of  $+..$

$6 +. 15$	$_{-}6 +. _{-}15$	$2.5 +. 3.5$	$6r7 +. 15r7$
3	3	0.5	$3r7$

Complex numbers are also in the domain of  $+..$

$a=: 1j2$	$b=: 2j3$	$c=: 3j5$	$(a*b) +. (b*c)$
$1j2$	$2j3$	$3j5$	$2j3$

If  $x$  and  $y$  are complex, then  $x +. y$  may differ from  $y +. x$ .

$1 +. 0j1$	$0j1 +. 1$
1	$0j1$

We can see that the same result is produced by Euclid's algorithm for  $(x \text{ GCD } y)$ , which is: if  $y=0$  then  $x$ , otherwise  $(x|y) \text{ GCD } x$ . Here is a verb E, to model the algorithm.

$E =: (| E []) \text{ `` } [ @. (]=0:)$

$6 \text{ E } 15$	$1 +. 0j1$	$1 \text{ E } 0j1$	$0j1 +. 1$	$0j1 \text{ E } 1$
3	1	1	$0j1$	$0j1$

The Least Common Multiple of  $x$  and  $y$  is given by  $(x * . y)$ .

<code>(2 * 3) *. (3 * 5)</code>	<code>2*3*5</code>
30	30

## 20.2 Circle Functions

### 20.2.1 Pi Times

There is a built-in verb `o.` (lower-case o dot). Monadic `o.` is called "Pi Times"; it multiplies its argument by 3.14159...

<code>o. 1</code>	<code>o. 2</code>	<code>o. 1r6</code>
3.14159	6.28319	0.523599

### 20.2.2 Trigonometric and Other Functions

If  $y$  is an angle in radians, then the sine of  $y$  is given by the expression `1 o. y`. The sine of (pi over 6) is 0.5

<code>y =: o. 1r6</code>	<code>1 o. y</code>
0.523599	0.5

The general scheme for dyadic `o.` is that  $(k \ o. \ y)$  means: apply to  $y$  a function selected by  $k$ . Giving conventional names to the available functions, we have:

```
sin    =: 1 & o.  NB.  sine
cos    =: 2 & o.  NB.  cosine
tan    =: 3 & o.  NB.  tangent
```

```

sinh  =: 5 & o. NB. hyperbolic sine
cosh  =: 6 & o. NB. hyperbolic cosine
tanh  =: 7 & o. NB. hyperbolic tangent

asin  =: _1 & o. NB. inverse sine
acos  =: _2 & o. NB. inverse cosine
atan  =: _3 & o. NB. inverse tangent

asinh =: _5 & o. NB. inverse hyperbolic sine
acosh =: _6 & o. NB. inverse hyperbolic cosine
atanh =: _7 & o. NB. inverse hyperbolic tangent

```

y	sin y	asin sin y
0.523599	0.5	0.523599

## 20.2.3 Pythagorean Functions

There are also the "pythagorean" functions:

```

0 o. y means %: 1 - y^2
4 o. y means %: 1 + y^2
8 o. y means %: - 1 + y^2
_4 o. y means %: _1 + y^2
_8 o. y means - %: - 1 + y^2

```

y =: 0.6	0 o. y	%: 1 - y^2
0.6	0.8	0.8

This is the end of chapter 20

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 16 Mar 2002

# Chapter 21: Factors and Polynomials

In this chapter we look at the built-in functions  $p$ ·,  $q$ · and  $p$ ·.

## 21.1 Primes and Factors

The built-in function monadic  $q$ · computes the prime factors of a given number.

$q$ · 6	$q$ · 8	$q$ · 17 * 31	$q$ · 1 + 2^30
2 3	2 2 2	17 31	5 5 13 41 61 1321

The number 0 is not in the domain of  $q$ ·. The number 1 is in the domain of  $q$ ·, but is regarded as having no factors, that is, its list of factors is empty.

$q$ · 0	$q$ · 1	# $q$ · 1
error		0

For large numbers, the value can be given as an extended integer to avoid a domain error.

$q$ · 1 + 2^31	$q$ · 1 + 2^31x
error	3 715827883

A prime number is the one and only member of its list of factors. Hence a test for primality can readily be written as the hook: member-of-its-factors

pr =: e. q:	pr 8	pr 17	pr 1
e. q:	0	1	0

Any positive integer can be written as the product of powers of successive primes. Some of the powers will be zero. For example we have:

$$9 = (2^0) * (3^2) * (5^0) * (7^0)$$

1

The list of powers, here 0 2 0 0 ... can be generated with dyadic q: . The left argument x specifies how many powers we choose to generate.

4 q: 9	3 q: 9	2 q: 9	1 q: 9	6 q: 9
0 2 0 0	0 2 0	0 2	0	0 2 0 0 0 0

Giving a left argument of "infinity" ( \_ ) means that the number of powers generated is just enough, in which case the last will be non-zero.

_ q: 9	_ q: 17 * 31
0 2	0 0 0 0 0 0 1 0 0 0 1

There is a built-in function, monadic p: (lowercase p colon) which generates prime numbers. For example (p: 17) is the 18th prime.

p: 0 1 2 3 4 5 6	p: 17
------------------	-------

2 3 5 7 11 13 17	61
------------------	----

On my computer the largest prime which can be so generated is between  $p: 2^{26}$  and  $p: 2^{27}$ .

$p: 2^{26}$	$p: 2^{27}$	$p: 2^{27}x$
1339484207	error	error

# 21.2 Polynomials

## 21.2.1 Coefficients

If  $x$  is a variable, then an expression in conventional notation such as

$$a + bx + cx^2 + dx^3 + \dots$$

is said to be a polynomial in  $x$ . If we write  $C$  for the list of coefficients  $a, b, c, d, \dots$  and assign a value to  $x$ , then the polynomial expression can be written in J in the form  $+/ C * x ^ i . \# C$

$C =: \_1 \ 0 \ 1$	$x=:2$
$\_1 \ 0 \ 1$	2

C	#C	i.#C	x	$x^{i.\#C}$	$C*x^{i.\#C}$	$+/C*x^{i.\#C}$
$\_1 \ 0 \ 1$	3	0 1 2	2	1 2 4	$\_1 \ 0 \ 4$	3

The dyadic verb  $p.$  allows us to abbreviate this expression to  $C \ p. \ x$ ,

<code>+ / C * x ^ i . # C</code>	<code>C p . x</code>
<code>3</code>	<code>3</code>

The scheme is that, for a list of coefficients `C`:

`C p . x` means `+ / C * x ^ i . # C`

A polynomial function is conveniently written in the form `C&p .`

<code>p =: _1 0 1 &amp; p .</code>	<code>p 0 1 2</code>
<code>_1 0 1&amp;p .</code>	<code>_1 0 3</code>

This form has a number of advantages: compact to write, efficient to evaluate and (as we will see) easy to differentiate.

## 21.2.2 Roots

Given a list of coefficients `C`, we can compute the roots of the polynomial function `C&p .` by applying monadic `p .` to `C`.

<code>C</code>	<code>p =: C &amp; p .</code>	<code>Z =: p . C</code>
<code>_1 0 1</code>	<code>_1 0 1&amp;p .</code>	<code>++-+-----+   1   1 _1   ++-+-----+</code>

We see that the result `z` is a boxed structure, of the form `M;R`, that is, multiplier `M` followed by list-of-roots `R`. We expect to see that `p` applied to each root in `R` gives zero.



'M R' =: Z	R	p R
<pre> +-+-----+  1 1 _1  +-+-----+ </pre>	1 _1	0 0

The significance of the multiplier M is as follows. If we write  $r, s, t \dots$  for the list of roots R,

$$'r \ s' =: R$$

then M is such that the polynomial  $C \ p. \ x$  can be written equivalently as

$$M * (x-r) * (x-s)$$

3

or more compactly as

$$M * */x-R$$

3

We saw that monadic  $p.$ , given coefficients C computes multiplier-and-roots  $M;R$ . Furthermore, given  $M;R$  then monadic  $p.$  computes coefficients C

C	MR =: p. C	p. MR
_1 0 1	<pre> +-+-----+  1 1 _1  +-+-----+ </pre>	_1 0 1

### 21.2.3 Dyadic p. Revisited

We saw above that the left argument of  $p.$  can be a list of coefficients, with the scheme

$$C \text{ p. } x \text{ is } +/ C * x ^ i. \#C$$

The left argument of p. can also be of the form multiplier;list-of-roots. In this way we can generate a polynomial function with specified roots. Suppose the roots are to be 2 3

p =: (1; 2 3) & p.	p 2 3
(1;2 3)&p.	0 0

The scheme is that

$$(M;R) \text{ p. } x \text{ means } M * */ x - R$$

When M;R is p. C then we expect (M;R) p. x to be the same as C p. x

C	MR=: p.C	MR p. x	C p. x
_1 0 1	<pre> ++-+-----+   1   1  _1   ++-+-----+ </pre>	3	3

### 21.2.4 Multinomials

Where there are many zero coefficients in a polynomial, it may be more convenient to write functions in the "multinomial" form, that is, omitting terms with zero coefficients and instead specifying a list of coefficient-exponent pairs. Here is an example. With the polynomial \_1 0 1 & p., the nonzero coefficients are the first and third, \_1 1, and the corresponding exponents are 0 2. We form the pairs thus:

<code>coeffs =: _1 1</code>	<code>exps=: 0 2</code>	<code>pairs =: coeffs ,. exps</code>
<code>_1 1</code>	<code>0 2</code>	<code>_1 0</code> <code>1 2</code>

Now the pairs can be supplied as boxed left argument to `p`. We expect the results to be the same as for the original polynomial.

<code>x</code>	<code>pairs</code>	<code>(&lt; pairs) p. x</code>	<code>_1 0 1 p. x</code>
<code>2</code>	<code>_1 0</code> <code>1 2</code>	<code>3</code>	<code>3</code>

With the multinomial form, exponents are not limited to non-negative integers. For example, with exponents and coefficients given by:

`E =: 0.5 _1 2j3`  
`C =: 1 1 1`

then the multinomial form of the function is:

`f =: (< C ,.E) & p.`

and for comparison, an equivalent function:

`g =: 3 : '+/ C * y. ^ E'`

We see

<code>x=: 2</code>	<code>f x</code>	<code>g x</code>

2	<code>_0.0337641j3.49362</code>	<code>_0.0337641j3.49362</code>
---	---------------------------------	---------------------------------

This is the end of Chapter 21.

---

Copyright © Roger Stokes 2001. This material may be freely reproduced, provided that this copyright notice, including this provision, is also reproduced.

last updated 09 Jan 2001

# Chapter 22: Vectors and Matrices

In this chapter we look at built-in functions which support computation with vectors and matrices.

## 22.1 The Dot Product Conjunction

Recall the composition of verbs, from [Chapter 08 p8](#). A sum-of-products verb can be composed from `sum` and `product` with the `@:` conjunction.

<code>P =: 2 3 4</code>	<code>Q =: 1 0 2</code>	<code>P * Q</code>	<code>+/ P * Q</code>	<code>P (+/ @: *) Q</code>
2 3 4	1 0 2	2 0 8	10	10

There is a conjunction `.` (dot, called "Dot Product"). It can be used instead of `@:` to compute the sum-of-products of two lists.

<code>P</code>	<code>Q</code>	<code>P (+/ @: *) Q</code>	<code>P (+/ . *) Q</code>
2 3 4	1 0 2	10	10

Evidently, the `.` conjunction is a form of composition, a variation of `@:` or `@`. We will see below that it is more convenient for working with vectors and matrices.

## 22.2 Scalar Product of Vectors

Recall that `P` is a list of 3 numbers. If we interpret these numbers as coordinates of a point in 3-dimensional space, then `P` can be regarded as defining a vector, a line-

segment with length and direction, from the origin at  $0\ 0\ 0$  to the point  $P$ . We can refer to the vector  $P$ .

With  $P$  and  $Q$  interpreted as vectors, then the expression  $P\ (+/\ .\ *)\ Q$  gives what is called the "scalar product" of  $P$  and  $Q$ . Other names for the same thing are "dot product", or "inner product", or "matrix product", depending on context. In this chapter let us stick to the neutral term "dot product", for which we define a function `dot`:

<code>dot =: +/ . *</code>	$P$	$Q$	$P\ dot\ Q$
<code>+/ . *</code>	$2\ 3\ 4$	$1\ 0\ 2$	$10$

A textbook definition of scalar product of vectors  $P$  and  $Q$  may appear in the form:  
 $(\text{magnitude } P) * (\text{magnitude } Q) * (\cos \alpha)$

where the magnitude (or length) of a vector is the square root of sum of squares of components, and  $\alpha$  is the smallest non-negative angle between  $P$  and  $Q$ . To show the equivalence of this form with  $P\ dot\ Q$ , we can define utility-verbs `ma` for magnitude-of-a-vector and `ca` for cos-of-angle-between-vectors.

```
ma =: %: @: (+/ @: *: )
ca =: 4 : '(-/ *: b, (ma x.-y.), c) % (2*(b=.ma x.)*(c=. ma y.))'
```

We expect the magnitude of vector  $3\ 4$  to be 5, and expect the angle between  $P$  and itself to be zero, and thus cosine to be 1.

<code>ma 3 4</code>	<code>P ca P</code>
$5$	$1$

then we see that the `dot` verb is equivalent to the textbook form above

P	Q	P dot Q	(ma P)*(ma Q)*(P ca Q)
2 3 4	1 0 2	10	10

# 22.3 Matrix Product

The verb we called dot is "matrix product" for vectors and matrices.

M =: 3 4 ,: 2 3	V =: 3 5	V dot M	M dot V	M dot M
3 4 2 3	3 5	19 27	29 21	17 24 12 17

To compute `Z =: A dot B` the last dimension of A must equal the first dimension of B.

A =: 2 5 \$ 1  
B =: 5 4 \$ 2

\$ A	\$ B	Z =: A dot B	\$ Z
2 5	5 4	10 10 10 10 10 10 10 10	2 4

The example shows that the last-and-first dimensions disappear from the result. If these two dimensions are not equal then an error is signalled.

\$ B	\$ A	B dot A
5 4	2 5	error

## 22.4 Generalisations

### 22.4.1 Various Verbs

The "Dot Product" conjunction forms the dot-product verb with  $(+ / \cdot *)$ . Other verbs can be formed on the pattern  $(u.v)$ .

For example, consider a relationship between people: person  $i$  is a child of person  $j$ , represented by a square boolean matrix true at row  $i$  column  $j$ . Using verbs  $+$  (logical-or) and  $*$  (logical-and). We can compute a grandchild relationship with the verb  $(+./ \cdot *.)$ .

$$g =: +. / \cdot *.$$

Taking the "child" relationship to be the matrix  $c$ :

$$c =: 4 \ 4 \ \$ \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0$$

Then the grandchild relationship is, so to speak, the child relationship squared.

C	G =: C g C
0 0 0 0	0 0 0 0
1 0 0 0	0 0 0 0
1 0 0 0	0 0 0 0
0 1 0 0	1 0 0 0

We can see from  $c$  that person 3 is a child of person 1, and person 1 is a child of person 0. Hence, as we see in  $g$  person 3 is a grandchild of person 0.

### 22.4.2 Symbolic Arithmetic

As arguments to the "Dot Product" conjunction we could supply verbs to perform symbolic arithmetic. Thus we might symbolically add the strings 'a' and 'b' to get the string 'a+b'. Here is a small collection of utility functions to do some limited symbolic arithmetic on strings.



```

pa      =: ('('&,) @: (,&'))
cp      =: [ ` pa @. (+./ @: ('+-*' & e.))
symbol  =: (1 : (':';'< (cp > x.), u., (cp > y.)))( " 0 0)

splus   =: '+' symbol
sminus  =: '-' symbol
sprod   =: '*' symbol

a =: <'a' b =: <'b' c =: <'c'

```

a	b	c	a splus b	a sprod b splus c
<pre> +--+  a  +--+ </pre>	<pre> +--+  b  +--+ </pre>	<pre> +--+  c  +--+ </pre>	<pre> +----+  a+b  +----+ </pre>	<pre> +-----+  a*(b+c)  +-----+ </pre>

As a variant of the symbolic product, we could elide the multiplication symbol to give an effect more like conventional notation:

```
sprodc =: '' symbol
```

a sprod b	a sprodc b
<pre> +----+  a*b  +----+ </pre>	<pre> +---+  ab  +---+ </pre>

Now for the `dot` verb, which we recall is `(+ / . *)`, a symbolic version is:

```
sdot =: splus / . sprodc
```

To illustrate:

```

S =: 3 2 $ < "0 'abcdef'
T =: 2 3 $ < "0 'pqrstu'

```

S	T	S sdot T
+--+--+  a b  +--+--+  c d  +--+--+  e f  +--+--+	+--+--+  p q r  +--+--+  s t u  +--+--+	+-----+-----+-----+  ap+bs aq+bt ar+bu  +-----+-----+-----+  cp+ds cq+dt cr+du  +-----+-----+-----+  ep+fs eq+ft er+fu  +-----+-----+-----+

### 22.4.3 Matrix Product in More than 2 Dimensions

An example in 3 dimensions will be sufficiently general. Symbolically:

A =: 1 2 3 \$ <"0 'abcdef'  
 B =: 3 2 2 \$ <"0 'mnopqrstuvwx'

A	B	Z =: A sdot B	\$A	\$B	\$Z
+--+--+  a b c  +--+--+  d e f  +--+--+	+--+--+  m n  +--+--+  o p  +--+--+  +--+--+  q r  +--+--+  s t  +--+--+  +--+--+  u v  +--+--+  w x  +--+--+	+-----+-----+-----+  am+(bq+cu) an+(br+cv)  +-----+-----+-----+  ao+(bs+cw) ap+(bt+cx)  +-----+-----+-----+  +-----+-----+-----+  dm+(eq+fu) dn+(er+fv)  +-----+-----+-----+  do+(es+fw) dp+(et+fx)  +-----+-----+-----+	1 2 3	3 2 2	1 2 2 2

The last dimension of A must equal the first dimension of B. The shape of the result z is the leading dimensions of A followed by the trailing dimensions of B. The last-and-first dimension of A and B have disappeared, because each dimensionless scalar in z combines a "row" of A with a "column" of B. We see in the result z that each row of A is combined separately with the whole of B.

## 22.4.4 Dot Compared With @:

Recall from [Chapter 07 p7](#) that a dyadic verb  $v$  has a left and right rank. Here are some utility functions to extract the ranks from a given verb.

```
RANKS    =: 1 : 'x. b. 0'
LRANK    =: 1 : '1 { (x. RANKS)'    NB. left rank only
```

* RANKS	* LRANK
0 0 0	0

The general scheme defining dyadic verbs of the form  $(u.v)$  is:

$u.v$  means  $u @ (v " (1+L, \_))$  where  $L = (v LRANK)$

or equivalently,

$u.v$  means  $(u @: v) " (1+L, \_)$   
and hence  
 $+/.*$  means  $(+/@: *) " 1 \_$

and so we see the difference between  $.$  and  $@:$ . For simple vector arguments they are the same, in which case the dimensions of the arguments must be the same, but this is not the condition we require for matrix multiplication in general, where (in the example above) each row of A is combined with the whole of B.

## 22.5 Determinant

The monadic verb (`- / . *`) computes the determinant of a matrix.  
`det =: - / . *`

M	det M	(3*3)-(2*4)
3 4 2 3	1	1

Symbolically:  
`sdet =: sminus / . sprodc`

S	sdet S
+-++  a b  +-++  c d  +-++  e f  +-++	+-----+   (a(d-f)) - ((c(b-f)) - (e(b-d)))   +-----+

## 22.5.1 Singular Matrices

A matrix is said to be singular if the rows (or columns) are not linearly independent, that is, if one row (or column) can be obtained from another by multiplying by a constant. A singular matrix has a zero determinant. In the following example A is a (symbolic) singular matrix, with `m` the constant multiplier.

A =: 2 2 \$ 'a';'b';'ma';'mb'	sdet A

<pre> +---+---+   a   b   +---+---+   ma   mb   +---+---+ </pre>	<pre> +-----+   amb-mab   +-----+ </pre>
--	--

We see that the resulting term ( amb-mab ) must be zero for all a, b and m.

## 22.6 Matrix Divide

### 22.6.1 Simultaneous Equations

The built-in verb `%.` (percent dot) is called "Matrix Divide". It can be used to find solutions to systems of simultaneous linear equations. For example, consider the equations written conventionally as:

$$\begin{aligned} 3x + 4y &= 11 \\ 2x + 3y &= 8 \end{aligned}$$

Rewriting as a matrix equation, we have, informally,

$$M \text{ dot } U = R$$

where  $M$  is the matrix of coefficients  $U$  is the vector of unknowns  $x, y$  and  $R$  is the vector of right-hand-side values:

<code>M =: 3 4 ,: 2 3</code>	<code>R =: 11 8</code>
<pre> 3 4 2 3 </pre>	<pre> 11 8 </pre>

The vector of unknowns  $U$  (that is,  $x, y$ ) can be found by dividing  $R$  by matrix  $M$ .

M	R	$U =: R \% . M$	$M \text{ dot } U$
$\begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 11 & 8 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \end{bmatrix}$	$\begin{bmatrix} 11 & 8 \end{bmatrix}$

We see that  $M \text{ dot } U$  equals  $R$ , that is,  $U$  solves the equations.

## 22.6.2 Complex, Rational and Vector Variables

The equations to be solved may be in complex variables. For example:

M	$R =: 15j22 \ 11j16$	$U =: R \% . M$	$M \text{ dot } U$
$\begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 15j22 & 11j16 \end{bmatrix}$	$\begin{bmatrix} 1j2 & 3j4 \end{bmatrix}$	$\begin{bmatrix} 15j22 & 11j16 \end{bmatrix}$

or in rationals. In this case both  $M$  and  $R$  must be rationals to give a rational result.

```
M =: 2 2 $ 3x 4x 2x 3x
R =: 15r22 11r16
```

M	R	$U =: R \% . M$	$M \text{ dot } U$
$\begin{bmatrix} 3 & 4 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 15r22 & 11r16 \end{bmatrix}$	$\begin{bmatrix} \_31r44 & 123r176 \end{bmatrix}$	$\begin{bmatrix} 15r22 & 11r16 \end{bmatrix}$

In the previous examples the unknowns in  $U$  were scalars. Now suppose the unknowns are vectors and our equations for solving are:

$$\begin{aligned} 3x + 4y &= \begin{bmatrix} 15 & 22 \end{bmatrix} \\ 2x + 3y &= \begin{bmatrix} 11 & 16 \end{bmatrix} \end{aligned}$$

so we write:

```
M =: 2 2 $ 3 4 2 3
R =: 2 2 $ 15 22 11 16
```

M	R	U =: R %. M	M dot U
3 4 2 3	15 22 11 16	1 2 3 4	15 22 11 16

The unknowns  $x$  and  $y$  are the rows of  $U$ , that is, vectors.

## 22.6.3 Curve Fitting

Suppose we aim to plot the best straight line fitting a set of data points. If the data points are  $x, y$  pairs given as:

```
x =: 0 1 2
y =: 3.1 4.9 7
```

we aim to find  $a$  and  $b$  for the equation:

$$y = a + bx$$

The 3 data points give us 3 equations in the 2 unknowns  $a$  and  $b$ . Conventionally:

$$\begin{array}{rcl} 1 \cdot a + 0 \cdot b & = & 3.1 \\ 1 \cdot a + 1 \cdot b & = & 4.9 \\ 1 \cdot a + 2 \cdot b & = & 7 \end{array}$$

so we take the matrix of coefficients  $M$  to be

```
M =: 3 2 $ 1 0 1 1 1 2
```

and divide  $y$  by matrix  $M$  to get the vector of unknowns  $U$ , (that is,  $a, b$ )

M	y	U =: y %. M	M dot U
1 0 1 1 1 2	3.1 4.9 7	3.05 1.95	3.05 5 6.95

Here we have more equations than unknowns, (more rows than columns in M) and so the solutions U are the best fit to all the equations together. We see that M dot U is close to, but not exactly equal to, y.

"Best fit" means that the sum of the squares of the errors is minimized, where the errors are given by  $y - M \text{ dot } U$ . If the sum of squares is minimized, then we expect that by perturbing U slightly, the sum of squares is increased.

+ / , *: (y - M dot U)	+ / , *: y - M dot U + 0.01
0.015	0.0164

The method extends straightforwardly to fitting a polynomial to a set of data points. Suppose we aim to fit

$$y = a + bx + cx^2$$

to the data points:

$$\begin{array}{l} x =: 0 \quad 1 \quad 2 \quad 3 \\ y =: 1 \quad 6 \quad 17 \quad 34.1 \end{array}$$

The four equations to be solved are:

$$1 \cdot a + bx_0 + cx_0^2 = y_0$$

$$1 \cdot a + bx_1 + cx_1^2 = y_1$$

$$1 \cdot a + bx_2 + cx_2^2 = y_2$$

$$1 \cdot a + bx_3 + cx_3^2 = y_3$$



and so the columns of matrix  $M$  are  $1, x, x^2$ , conveniently given by  $x \wedge / 0 1 2$

<code>M =: x ^/ 0 1 2</code>	<code>y</code>	<code>U =: y %. M</code>	<code>M dot U</code>
<pre>1 0 0 1 1 1 1 2 4 1 3 9</pre>	<pre>1 6 17 34.1</pre>	<pre>1.005 1.955 3.025</pre>	<pre>1.005 5.985 17.015 34.095</pre>

There may be more equations than unknowns, as this example shows, but evidently there cannot be fewer. That is, in `R %. M` matrix  $M$  must have no more columns than rows.

## 22.6.4 Dividing by Higher-Rank Arrays

Here is an example of `U =: R %. M`, in which the divisor  $M$  is of rank 3.

```
M =: 3 2 2 $ 3 4 2 3 0 3 1 2 3 1 2 3
R =: 21 42
```

<code>M</code>	<code>R</code>	<code>U =: R %. M</code>	<code>M dot U</code>	<code>M dot"2 1 U</code>
<pre>3 4 2 3  0 3 1 2  3 1 2 3</pre>	<pre>21 42</pre>	<pre>_105 84  28  7    3 12</pre>	<pre>error</pre>	<pre>21 42 21 42 21 42</pre>

Because the dyadic rank of `%.` is `_ 2`,

```
%. b. 0
2 _ 2
```

in this example the whole of  $R$  is combined separately with each of the 3 matrices in  $M$ . That is, we have 3 separate sets of equations, each with the same right-hand-side  $R$ . Thus we have 3 separate solutions (the rows of  $U$ ).

The condition  $R=M \cdot U$  evidently does not hold (because the last dimension of  $M$  is not equal to the first of  $U$ ), but it does hold separately for each matrix in  $M$  with corresponding row of  $U$ .

## 22.7 Identity Matrix

A (non-singular) square matrix  $M$  divided by itself yields an "identity matrix",  $I$  say, such that  $(M \cdot I) = M$ .

$M =:$  3 3 \$ 3 4 7 0 0 4 6 0 3

M	I =: M % . M	M dot I
3 4 7 0 0 4 6 0 3	1 0 0 0 1 0 0 0 1	3 4 7 0 0 4 6 0 3

## 22.8 Matrix Inverse

The monadic verb `% .` computes the inverse of a matrix. That is, `% . M` is equivalent to `I % . M` for a suitable identity matrix  $I$ :

M	I =: M % . M	I % . M	% . M

3 4	1 0	0 _0.125	0 _0.125
7	0	0.166667	0.166667
0 0	0 1	0.25 _0.34375	0.25 _0.34375
4	0	_0.125	_0.125
6 0	0 0	0	0
3	1	0.25 0	0.25 0

For a vector v, the inverse w has the reciprocal magnitude and the same direction. Thus the product of the magnitudes is 1 and the cosine of the angle between is 1.

V	W =: %. V	(ma V) * (ma W)	V ca W
3 5	0.0882353 0.147059	1	1

This is the end of Chapter 22.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 17 Aug 2002

# Chapter 24: Names and Locales

In this chapter we look at locales. The interest of locales is twofold: as a way of organizing large programs, and (as we will come to in the next chapter) as the basis of object-oriented programming in J.

## 24.1 Background

It is generally agreed that a large program is best developed in several parts which are, as much as possible, independent of each other. For example, an independent part of a larger program might be a collection of statistical functions, with its own script-file.

For the things defined in an independent script, we expect to choose names for those things more or less freely, without regard for what names may be defined in other scripts. Clearly there may be a problem in combining independent scripts: what if the same name accidentally receives different definitions in different scripts? The J facility of the "locale" gives a way to deal with this problem.

## 24.2 What are Locales?

After entering an assignment of the form `(name =: something)` we say we have a definition of `name`. Every definition is stored in some region of the memory of the J system called a "locale". Roughly speaking, locales are to definitions as directories are to files. The important features of locales are:

- There can be several different locales existing at the same time.
- Each locale stores a collection of definitions.
- The same name can occur at the same time in different locales with different definitions.

Hence a name of the form "name  $N$  as defined in locale  $L$ " is uniquely defined,

without conflict. Such a name can be written as  $\underline{N\_L\_}$  ( $\underline{N}$  underbar  $\underline{L}$  underbar) and is called a "locative name". Finally

- At any one time, only one locale is current. The current locale is the one whose definitions are available for immediate use.

Hence a plain name  $N$  commonly means " $N$  as defined in the current locale".

Locales are neither nouns, verbs, adverbs nor conjunctions: that is, locales are not values which can be assigned to variables or be passed as arguments to functions. They do have names, but whenever we need to refer to a locale by name we do so either with special syntactic forms, (locative names such as  $\underline{N\_L\_}$ ) or by quoting the name to form a string.

## 24.3 Example

Suppose we are interested in the correlation between the price of whisky and the general level of employee salaries. Suppose also that we have available two scripts, of independent origin, one with economic data and the other with statistical functions. These script-files might have been created like this:

```
(0 : 0) (1 !: 2) < 'c:\economic.ijs'
y  =: 1932  1934  1957  1969  1972    NB. years
s  =: 1000  1000  3000  9000 11000    NB. salaries
p  =: 1.59  1.68  2.00  4.50  5.59    NB. prices
)

(0 : 0) (1 !: 2) < 'c:\statfns.ijs'
m =: +/ % #          NB. Mean
n =: - m             NB. Norm
v =: m @: *: @:
n  NB. Variance
s =: %: @: v         NB. Standard Deviation
c =: m @: (*&n)       NB. Covariance
r =: c % (*&s)        NB. Correlation Coefficient
)
```

We aim to load these two scripts, and then hope to compute the coefficient of correlation between prices  $p$  and salaries  $s$  as the value of the expression  $(p\ r\ s)$ .

Unfortunately we can see that the name  $s$  means different things in the different

scripts. If we were to load both scripts into the same locale, one definition of `s` would overwrite the other. The remedy is to load the two scripts into different locales.

There is always a locale named `base`, and by default this is usually current. We load `economic.ijs` into the current locale (`base`) with the built-in verb `(0 !: 0)`.

```
(0 !: 0) < 'c:\economic.ijs'
```

Next we load `statfns.ijs` into another locale which we choose to call, say, `stat`. To do this with the minimum of new apparatus we can use the built-in verb `(18 !: 4)`.

```
(18 !: 4) < 'stat'
(0 !: 0) < 'C:\statfns.ijs'
(18 !: 4) < 'base'
```

The first line creates the `stat` locale and makes it current. The second line loads `statfns.ijs` into the now-current locale `stat`. The third line makes the `base` locale current again, to restore the status quo.

At this point our data variables `s` and `p` are available because they are in `base` which is current. The correlation-coefficient function `r` is not yet available, because it is in `stat` which is not current.

The next step is to define the correlation-coefficient function to be `r`-as-defined-in-locale- `stat`, using the locative form of name `r_stat_`

```
corr =: r_stat_
```

`corr` is available because it has just been defined in `base` (because `base` is current). Everything we need is now available. We can compute the correlation between salaries and prices.

s corr p	p corr s	p corr p
0.993816	0.993816	1

## 24.3.1 Review

What we have seen is the use of locative names to resolve name-conflicts between independent scripts. What it took was a relatively small amount of ad-hoc further definition.

In this tiny example the conflict was easily identified and could be easily fixed by editing one of the scripts. However, the point is that we aim to avoid tampering with independent scripts to get them to work together.

## 24.4 The Current Locale

Several locales may coexist, but at any one time only one is current. There is a built-in verb (18 !: 5) which tells us the name of the current locale.

```
(18 !: 5) '' NB. show name of current locale
+-----+
|base|
+-----+
```

The significance of the current locale is that it is in the current locale that simple names are evaluated:

```
s
1000 1000 3000 9000 11000
```

Notice that we get the value of `s` as defined in script `economic.ijs` which we loaded into `base`, and not the value of `s` in `statfns.ijs` which was loaded into locale `stat`.

It is the current locale in which new definitions are stored. To see what names are defined in the current locale we can use the built-in verb (4 !: 1) with an argument of 0 1 2 3.

```
(4 !: 1) 0 1 2 3 NB. show all names in current locale
+-----+-----+
|corr|p|s|y|
+-----+-----+

foo =: +/
```

```

      (4 !: 1) 0 1 2 3
+-----+-----+-----+
|corr|foo|p|s|y|
+-----+-----+-----+

```

As we saw above, we can change the current locale with the built-in verb (18 !: 4). We can make the `stat` locale current with:

```
(18 !: 4) < 'stat'
```

and now we can see what names are defined in this locale with:

```

      (4 !: 1) 0 1 2 3
+-----+-----+-----+
|c|m|n|r|s|v|
+-----+-----+-----+

```

and return to `base` again

```
(18 !: 4) < 'base'
```

## 24.5 The `z` Locale Is Special

The locale named `z` is special in the following sense. When a name is evaluated, and a definition for that name is not present in the current locale, then the `z` locale is automatically searched for that name. Here is an example. We put into locale `z` a definition of a variable `q`, say.

```

(18 !: 4) < 'z'
q =: 99
(18 !: 4) < 'base'

```

Now we see that `q` is not present in the current locale (`base`) but nevertheless we can evaluate the simple name `q` to get its value as defined in locale `z`.

```

      (4 !: 1) 0 1 2 3
+-----+-----+-----+
|corr|foo|p|s|y|
+-----+-----+-----+

```



Since we can find in `z` things which are not in `base`, locale `z` is the natural home for functions of general utility. On starting a J session, locale `z` is automatically populated with a collection of useful predefined "library" functions.

The names of these functions in the `z` locale are all available for immediate use, and yet the names, of which there are more than 100, do not clutter the `base` locale.

We saw above the use of built-in verbs such as `(18 !: 4)` and `(4 !: 1)`. However the library verbs of locale `z` are often more convenient. Here is a small selection:

<code>coname ''</code>	show name of current locale
<code>conl 0</code>	show names of all locales
<code>names ''</code>	show all names in current locale
<code>nl ''</code>	show all names in current locale (as a boxed list)
<code>cocurrent 'foo'</code>	locale <code>foo</code> becomes current
<code>clear 'foo'</code>	remove all defs from locale <code>foo</code>
<code>coerase 'A';'B';'C'</code>	erase locales <code>A B</code> and <code>C</code>

We have seen that when a name is not found in the current locale, the search proceeds automatically to the `z` locale. In other words, there is what is called a "path" from every locale to the `z` locale. We will come back to the subject of paths below.

# 24.6 Locative Names and the Evaluation of Expressions

## 24.6.1 Assignments

An assignment of the form `n_L_ =: something` assigns the value of `something` to the name `n` in locale `L`. Locale `L` is created if it does not already exist. For example:

```
n_L_ =: 7
```

creates the name `n` in locale `L` with value 7. At this point it will be helpful to introduce a utility-function to view all the definitions in a locale. We put this `view` function into locale `z`:

```
VIEW_z_ =: 3 : '(> ,. (' =: '&,)@:(5!:5)"0) nl ''''  
view_z_ =: 3 : 'VIEW__lo '''' [ lo =. < y.'
```

If we make a few more definitions:

```
k_L_ =: 0  
n_M_ =: 2
```

we can survey what we have in locales `L` and `M`:

view 'L'	view 'M'
k =: 0 n =: 7	n =: 2

Returning now to the theme of assignments, the scheme is: if the current locale is `L`, then `(foo_M_ =: something)` means:

1. evaluate `something` in locale `L` to get value `v` say.

2. cocurrent 'M'
3. foo =: v
4. cocurrent 'L'

For example:

cocurrent 'L'

view 'L'	view 'M'	k_M_ =: n-1	view 'M'
k =: 0 n =: 7	n =: 2	6	k =: 6 n =: 2

## 24.6.2 Evaluating Names

Now we look at locative names occurring in expressions. The scheme (call this scheme 2) is: if the current locale is L then (n\_M\_) means

1. cocurrent 'M'
2. evaluate the name n to get a value v
3. cocurrent 'L'
4. v

For example:

cocurrent 'L'

view 'L'	view 'M'	n_M_
k =: 0 n =: 7	k =: 6 n =: 2	2

## 24.6.3 Applying Verbs

Consider the expression  $(f\_M\_ n)$ . This means: function  $f$  (as defined in locale  $M$ ) applied to an argument  $n$  (as defined in the current locale) In this case, the application of  $f$  to  $n$  takes place in locale  $M$ . Here is an example:

```
u_M_ =: 3 : 'y.+k'
```

```
cocurrent 'L'
```

view 'L'	view 'M'	u_M_ n
k =: 0 n =: 7	k =: 6 n =: 2 u =: 3 : 'y.+k'	13

We see that the argument  $n$  comes from the current locale  $L$ , but the constant  $k$  comes from the locale ( $M$ ) from which we took verb  $u$ . The scheme (call it scheme 3) is: if the current locale is  $L$ , then  $(f\_M\_ something)$  means:

1. evaluate something in  $L$  to get a value  $V$  say
2. cocurrent 'M'
3. in locale  $M$ , evaluate the expression  $f\_V$  to get a value  $R$  say
4. cocurrent 'L'
5.  $R$

Here is another example. The verb  $g$  is taken from the same locale in which  $f$  is found:

```
g_L_ =: +&1
```

```
g_M_ =: +&2
```

```
f_M_ =: g
```

```
cocurrent 'L'
```

view 'L'	view 'M'	f_M_ k
g =: +&1 k =: 0 n =: 7	f =: g g =: +&2 k =: 6 n =: 2 u =: 3 : 'y.+k'	2

## 24.6.4 Applying Adverbs

To begin, note that when an adverb is applied, names of verbs do not get evaluated.

w =: +	z =: *	ADV =: @: z	w ADV
+	*	@: z	w@: z

The result is an expression for a verb in terms of w, the argument, and z which occurs in the definition of ADV

Here now is an example of an adverb referred to by a locative name. We enter definitions in fresh locales P and Q.

```
u_P_ =: *&2
v_P_ =: *&3
u_Q_ =: *&7
v_Q_ =: *&5
A_Q_ =: @: v
```

make P the current locale, and apply adverb A\_Q\_ to argument u to get verb D:  
cocurrent 'P'

view 'P'	view 'Q'	D =: u A_Q_	D 1
u =: *&2 v =: *&3	A =: @: v u =: *&7 v =: *&5	u@: v	6

Evidently the result 6 is obtained by taking  $u$  and  $v$  from the current locale which is  $P$ .

The scheme is that if the current locale is  $P$ , and  $A$  is an adverb, the expression  $u A_Q_$  means:

1. evaluate  $u$  in locale  $P$  to get a value  $U$  say. (and if  $u$  is the name of a verb, then the result  $U$  is just  $u$ .)
2. cocurrent  $Q$
3. evaluate  $U A$  in locale  $Q$ . The result is a verb,  $D$  say, defined in terms of named verbs ( $u$  and  $v$  in this example.)
4. cocurrent  $P$
5.  $D$

We can demonstrate that the evaluation of  $u A_Q_$  takes place in locale  $Q$  by forcing all named verbs to be evaluated. We apply the "fix" adverb to disclose the definitions of the named verbs.

$B_Q_ =: (@: v) f.$

view 'P'	view 'Q'	D =: u B_Q_	D 1
D =: u@: v u =: *&2 v =: *&3	A =: @: v B =: (@: v) f. u =: *&7 v =: *&5	*&7@: ( *&5 )	35

Evidently the argument  $u$  and the auxiliary verb  $v$  are both taken from locale  $Q$ .

## 24.7 Paths

Recall that the `z` locale contains useful "library" functions, and that we said there is a path from any locale to `z`.

We can inspect the path from a locale with the library verb `copath`; we expect the path from locale `base` to be just `z`.

```
copath 'base'    NB. show path
+--+
| z |
+--+
```

A path is represented as a (list of) boxed string(s). We can build our own structure of search-paths between locales. We will give `base` a path to `stat` and then to `z`, using dyadic `copath`.

```
('stat';'z') copath 'base'
```

and check the result is as expected:

```
copath 'base'
+-----+--+
| stat | z |
+-----+--+
```

With this path in place, we can, while `base` is current, find names in `base`, `stat` and `z`.

```
cocurrent 'base'

s      NB. in base
1000 1000 3000 9000 11000

r      NB. in stat
+--+-----+
| c | % | +--+--+ | | | | |
| | | | * | & | s | |
| | | +--+--+ |
+--+-----+
```

q NB. in z  
99

Suppose we set up a path from L to M. Notice that we want every path to terminate at locale z, (otherwise we may lose access to the useful stuff in z) so we make the path go from L to M to z.

```
('M';'z') copath 'L'
```

If we access a name along a path, there is no change of current locale. Compare the effects of referring to verb u via a locative name and searching for u along a path.

```
cocurrent 'L'
```

view 'L'	view 'M'	u_M_ 0	u 0
ADV =: @:z g =: +&1 k =: 0 n =: 7 w =: + z =: *	f =: g g =: +&2 k =: 6 n =: 2 u =: 3 : 'y.+k'	6	0

We see that in evaluating (u\_M\_ 0) there is a change of locale to M, so that the variable k is picked up with its value in M of 6. In evaluating (u 0), where u is found along the path, the variable k is picked up from the current locale, with its value in L of 0.

When a name is found along a path, it is as though the definition were temporarily copied into the current locale. Here is another example.

view 'L'	view 'M'	f_M_ 0	f 0



ADV =: @:z	f =: g	2	1
g =: +&1	g =: +&2		
k =: 0	k =: 6		
n =: 7	n =: 2		
w =: +	u =: 3 : 'y.+k'		
z =: *			

## 24.8 Combining Locatives and Paths

We sometimes want to populate a locale with definitions from a script-file. We saw above one way to do this: in three steps:

- (1) use `cocurrent` (or `18! : 4`) to move to the specified locale.
- (2) execute the script-file with `0! : 0`
- (3) use `cocurrent` (or `18! : 4`) to return to the original locale.

A neater way is as follows. We first define:

```
POP_z_ =: 0 !: 0
```

and then to populate locale `Q` say, from file `economic.ijs`, we can write:

```
POP_Q_ < 'c:\economic.ijs'
```

which works like this:

1. The `POP` verb is defined in locale `z`.
2. Encountering `POP_Q_ < ...` the system makes locale `Q` temporarily current, creating `Q` if it does not already exist.
3. The system looks for a definition of `POP`. It does not find it in `Q`, because `POP` is of course defined in locale `z`.
4. The system then looks along the path from `Q` to `z` and finds `POP`. Note that the current locale is still (temporarily) `Q`.
5. The `POP` verb is applied to its argument, in temporarily-current locale `Q`.

6. The current locale is automatically restored to whatever it was beforehand.

Back to base. (If we are nipping about between locales it is advisable to keep track of where we are.)

```
cocurrent <'base'
```

## 24.9 Indirect Locatives

A variable can hold the name of a locale as a boxed string. For example, given that `m` is a locale,

```
loc =: < 'M'
```

Then a locative name such as `k_M_` can be written equivalently in the form `k__loc` (u underscore underscore loc)

```
k_M_
```

6

```
k__loc
```

6

The point of this indirect form is that it makes it convenient to supply locale-names as arguments to functions.

```
NAMES =: 3 : 0
locname =. < y.
names__locname ''
)
```

```
NAMES 'L'
ADV g k n w z
```

This is the end of Chapter 24

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 15 Mar 2002

# Chapter 25: Object-Oriented Programming

## 25.1 Background and Terminology

In this chapter "OOP" will stand for "object-oriented programming". Here is the barest thumbnail sketch of OOP.

On occasion, a program needs to build, maintain and use a collection of related data, where it is natural to consider the collection to be, in some sense, a whole. For example, a "stack" is a sequence of data items, such that the most-recently added item is the first to be removed. If we intend to make much use of stacks, then it might be a worthwhile investment to write some functions dedicated to building and using stacks.

The combination of some data and some dedicated functions is called an object. Every object belongs to some specific class of similar objects. We will say that a stack is an object of the `Stack` class.

The dedicated functions for objects of a given class are called the "methods" of the class. For example, for objects of the `Stack` class we will need a method for adding a new item, and a method for retrieving the last-added item.

An object needs one or more variables to represent its data. Such variables are called fields. Thus for a stack we may choose to have a single field, a list of items.

In summary, OOP consists of identifying a useful class of objects, and then defining the class by defining methods and fields, and then using the methods. By organizing a program into the definitions of different classes, OOP can be viewed as a way of managing complexity. The simple examples which follow are meant to illustrate the machinery of the OOP approach, but not to provide much by way of motivation for OOP.

We will be using a number of library functions. A brief summary of them is given at the end of this chapter.

## 25.2 Defining a Class

### 25.2.1 Introducing the Class

For a simple example, we look at defining a class of `Stack` objects. A new class is introduced with the library function `coclass`.

```
coclass 'Stack'
+-----++
|Stack|z|
+-----++
```

`coclass` is used for its effect, not its result. The effect of `coclass` is to establish and make current a new locale called `Stack`. To verify this, we can inspect the name of the current locale:

```
coname ''
+-----+
|Stack|
+-----+
```

### 25.2.2 Defining the Methods

A new object comes into being in two steps. The first step uses library verb `conew` to create a rudimentary object, devoid of fields, a mere placeholder. The second step gives a new object its structure and initial content by creating and assigning values to the field-variables.

We will deal with the first step below. The second step we look at now. It is done by a method conventionally called `create` (meaning "create fields", not "create object"). This is the first of the methods we must define.

For example, we decide that a `Stack` object is to have a single field called `items`, initially an empty list.

```
create =: 3 : 'items =: 0 $ 0'
```

The connection between this method and the `Stack` class is that `create` has just been defined in the current locale, which is `Stack`.

This `create` method is a verb. In this example, it ignores its argument, and its result is of no interest: it is executed purely for its effect. Its effect will be that the (implicitly specified) object will be set up to have a single field called `items` as an empty list.

Our second method is for pushing a new value on to the front of the `items` in a stack.

```
push =: 3 : '# items =: (< y.) , items'
```

The `push` method is a verb. Its argument `y.` is the new value to be pushed. We made a design-decision here that `y.` is to be boxed and then pushed. The result is of no interest, but there must be some result, so we chose to return `(# items)` rather than just `items`.

Next, methods for, respectively, returning and removing the "top" (most-recently added) item on the stack.

```
top =: 3 : '> {. items'
pop =: 3 : '# items =: }. items'
```

Finally, a method to "destroy" a `Stack` object, that is, eliminate it when we are finished with it. For this purpose there is a library function `code destroy`.

```
destroy =: code destroy
```

This completes the definition of the `Stack` class. Since we are still within the scope of the `coclass 'Stack'` statement above, the current locale is `Stack`. To use this class definition we return to our regular working environment, the `base` locale.

```
cocurrent 'base'
```

## 25.3 Making New Objects

Now we are in a position to create and use `Stack` objects. A new `Stack` is created in two steps. The first step uses the library verb `conew`.

```
S =: conew 'Stack'
```

The result of `conew` which we assigned to `s` is not the newly-created object itself. Rather, the value of `s` is in effect a unique reference-number which identifies the newly-created `Stack` object. For brevity we will say "Stack `s`" to mean the object referred to by `s`.

Stack `s` now exists but its state is so far undefined. Therefore the second step in making the object is to use the `create` method to change the state of `s` to be an empty stack. Since `create` ignores its argument, we supply an argument of 0

```
create__S 0
```

Now we can push values onto the stack `s` and retrieve them in last-in-first-out order. In the following, the expression `(push__S 'hello'` means: the method `push` with argument `'hello'` applied to object `s`.

```
push__S 'hello'
1
push__S 'how are you?'
2
push__S 'goodbye'
3
pop__S 0
2
top__S 0
how are you?
```

## 25.3.1 Dyadic Conew

The two steps involved in creating a new object, `conew` followed by `create`, can be collapsed into one using dyadic `conew`. The scheme is that:

```
C =: conew 'Class'
create__C arg
```

can be abbreviated as:

```
C =: arg conew 'Class'
```

That is, any left argument of `conew` is passed to `create`, which is automatically invoked. In this simple `Stack` class, `create` ignores its argument, but even so one step is neater than two. For example:

```
T =: 0 conew 'Stack'
push__T 77
1
push__T 88
2
top__T 0
88
```

## 25.4 Classes and Objects are Locales

Recall from [Chapter 24 p23](#) that the expression `conl 0` produces a list of existing locales.

```
conl 0
+-----+-----+----+
|Stack|base|j|z|
+-----+-----+----+
```

We see that `Stack` is amongst this list, and so a class-definition is a locale. The methods of the `Stack` class are defined in the locale named `Stack`. We can view this locale (using the `view` utility function from the previous chapter.)

```
view 'Stack'
COCLASSPATH =: 'Stack';'z'
create      =: 3 : 'items =: 0 $ 0'
destroy     =: codestroy
pop         =: 3 : '# items =: }. items'
push        =: 3 : '# items =: (< y.) , items'
top         =: 3 : '> {. items'
```

### 25.4.1 What objects?

The library verb `costate` produces a report showing what objects exist, and their



classes. Currently we have variables `s` and `T` each referring to a `Stack` object.

```

costate ''
+-----+-----+-----+-----+
|refs|id|creator|path  |
+-----+-----+-----+-----+
|S   |0 |base   |Stack z|
+-----+-----+-----+-----+
|T   |1 |base   |Stack z|
+-----+-----+-----+-----+

```

Now look at the value of `s`. This value is a boxed character string consisting of numeric digits:

```

S
+-+
|0|
+-+

```

This string is the name of a locale (a name consisting only of numeric digits) and this locale contains the fields of object `s`. Thus objects are locales. We can view them:

S	view >S	view >T
<pre> +-+  0  +-+ </pre>	<pre> COCREATOR =: &lt;'base' items      =: &lt;['_1 '] how are you? hello' </pre>	<pre> COCREATOR =: &lt;'base' items      =: 88;77 </pre>

The numeric names of object locales are shown in the `costate` report above under the heading `id`.

Let us look at the `costate` report again.

```

costate ''
+-----+-----+-----+-----+
|refs|id|creator|path  |
+-----+-----+-----+-----+
|S   |0 |base   |Stack z|
+-----+-----+-----+-----+
|T   |1 |base   |Stack z|
+-----+-----+-----+-----+

```

```
+-----+---+-----+-----+
```

What makes `S` a `Stack` object is that there is a path from the `S` locale to the `Stack` locale. We can inspect this path (also shown in the `costate` report):

copath S	costate ''
+-----+---+  Stack z  +-----+---+	+-----+---+-----+-----+  refs id creator path  +-----+---+-----+-----+  S 0 base Stack z  +-----+---+-----+-----+  T 1 base Stack z  +-----+---+-----+-----+

Recall from [Chapter 24 p23](#) that, since `S = <'0'` then the expression `push__S 99` means:

1. change the current locale to `'0'`. Now the fields of object `S`, (that is, the `items` variable of locale `'0'`) are available.
- apply the `push` verb to argument `99`. Since `push` is not in locale `'0'`, a search is made along the path from locale `'0'` which takes us to locale `Stack` whence `push` is retrieved before it is applied.
  - Restore the current locale to the status quo.

## 25.5 Inheritance

Here we look at how a new class can build on an existing class. The main idea is that, given some class, we can develop a new class as a specialized version of the old class.

For example, suppose there is a class called `Collection` where the objects are

collections of values. We could define a new class where, say, the objects are collections without duplicates, and this class could be called `Set`. Then a `Set` object is a special kind of a `Collection` object.

In such a case we say that the `Set` class is a child of the parent class `Collection`. The child will inherit the methods of the parent, perhaps modifying some and perhaps adding new methods, to realize the special properties of child objects.

For a simple example we begin with a parent-class called `Collection`,

```
coclass 'Collection'
+-----+
|Collection|z|
+-----+
  create  =: 3 : 'items =: 0 $ 0'
  add      =: 3 : '# items =: (< y.) , items'
  remove   =: 3 : '# items =: items -. < y.'
  inspect  =: 3 : 'items'
  destroy  =: code:destroy
```

Here the `inspect` method yields a boxed list of all the members of the collection.

A quick demonstration:

```
cocurrent 'base'
C1 =: 0 conew 'Collection'
add__C1 'foo'
1
add__C1 37
2
remove__C1 'foo'
1
inspect__C1 0
+--+
|37|
+--+
```

Now we define the `Set` class, specifying that `Set` is to be a child of `Collection` with the library verb `coextend`.

```
coclass 'Set'
+-----+
|Set|z|
```

```

+---+--+
      coextend 'Collection'
+---+-----+--+
|Set|Collection|z|
+---+-----+--+

```

To express the property that a `Set` has no duplicates, we need to modify only the `add` method. Here is something that will work:

```

add =: 3 : '# items =: ~. (< y.) , items'

```

All the other methods needed for `Set` are already available, inherited from the parent class `Collection`. We have finished the definition of `Set` and are ready to use it.

```

cocurrent 'base'
s1 =: 0 conew 'Set' NB. make new Set object.
add__s1 'a'
1
add__s1 'b'
2
add__s1 'a'
2
remove__s1 'b'
1
inspect__s1 0 NB. should have just one 'a'
+--+
|a|
+--+

```

## 25.5.1 A Matter of Principle

Recall the definition of the `add` method of class `Set`.

```

add_Set_
+---+-----+
|3|:|# items =: ~. (< y.) , items|
+---+-----+

```

It has an objectionable feature: in writing it we used our knowledge of the internals of a `Collection` object, namely that there is a field called `items` which is a boxed list.

Now the methods of `Collection` are supposed to be adequate for all handling of `Collection` objects. As a matter of principle, if we stick to the methods and avoid rummaging around in the internals, we hope to shield ourselves, to some degree, from possible future changes to the internals of `Collection`. Such changes might be, for example, for improved performance.

Let's try redefining `add` again, this time sticking to the methods of the parent as much as possible. We use our knowledge that the parent `inspect` method yields a boxed list of the membership. If the argument `y.` is not among the membership, then we add it with the parent `add` method.

```
    add_Set_ =: 3 : 0
if. (< y.) e. inspect 0
do. 0
else. add_Collection_ f. y.    NB. see below !
end.
)
```

Not so nice, but that's the price we pay for having principles. Trying it out on the set `s1`:

```
    inspect__s1 0
+--+
|a|
+--+
    add__s1      'a'
0
    add__s1      'z'
2
    inspect__s1 0
+--+--+
|z|a|
+--+--+
```

## 25.6 Using Inherited Methods

Let us review the definition of the `add` method of class `Set`.

```
add_Set_  
+--+-----+  
|3|:|if. (< y.) e. inspect 0|  
| | |do. 0|  
| | |else. add_Collection_ f. y. NB. see below !|  
| | |end.|  
+--+-----+
```

There are some questions to be answered.

## 25.6.1 First Question

How are methods inherited? In other words, why is the `inspect` method of the parent `Collection` class available as a `Set` method? In short, the method is found along the path, that is,

- a `Set` object such as `s1` is a locale. It contains the field-variable(s) of the object.
- when a method of a class is executed, the current locale is (temporarily) the locale of an object of that class. This follows from the way we invoke the method, with an expression of the form `method__object argument`.
- the path from an object-locale goes to the class locale and thence to any parent locale(s). Hence the method is found along the path.

. We see that a `Set` object such as `s1` has a path to `Set` and then to `Collection`.

```
copath > s1  
+--+-----+  
|Set|Collection|z|  
+--+-----+
```

## 25.6.2 Second Question

In the definition of `add_Set_`

```
add_Set_  
+--+-----+  
|3|:|if. (< y.) e. inspect 0|
```

```

| | |do. 0
| | |else. add_Collection_ f. y. NB. see below !
| | |end.
+--+-----+

```

Given that the parent method `inspect` is referred to as simply `inspect`, why is the parent method `add` referred to as `add_Collection_`? Because we are defining a method to be called `add` and inside it a reference to `add` would be a fatal circularity.

### 25.6.3 Third Question

why is the parent `add` method specified as `add_Collection_ f. ?`

Because `add_Collection_` is a locative name, and evaluating expressions with locative names will involve a change of locale. Recall from [Chapter 24 p23](#) that `add_Collection_ 0` would be evaluated in locale `Collection`, which would be incorrect: we need to be in the object locale when applying the method.

Since `f.` is built-in, by the time we have finished evaluating `(add_Collection_ f.)` we are back in the right locale with a fully-evaluated value for the function which we can apply without change of locale.

Would not some other adverb, say the identity- adverb `] :` do instead of `f. ?` No, because `] :` does not evaluate its argument - its result is still a locative.

<code>add_Collection_ f.</code>	<code>add_Collection_ ] :</code>
<pre> +--+-----+  3 : # items =: (&lt; y.) , items  +--+-----+ </pre>	<pre> +-----+  add_Collection_  +-----+ </pre>

## 25.7 Library Verbs

Here is a brief summary of selected library verbs.

<code>coclass 'foo'</code>	introduce new class <code>foo</code>
<code>coextend 'foo'</code>	this class to be a child of <code>foo</code>
<code>conew 'foo'</code>	introduce a new object of class <code>foo</code>
<code>conl 0</code>	list locale names
<code>conl 1</code>	list ids of object locales
<code>costate ''</code>	list objects and classes
<code>names_foo_ ''</code>	list the methods of class <code>foo</code>
<code>copath &lt;'foo'</code>	show path of class <code>foo</code>
<code>codestroy ''</code>	method to destroy this object
<code>copathnlx__o</code>	show field-names in object <code>o</code>
<code>coname ''</code>	show name of current locale

This brings us to the end of Chapter 25

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 15 Mar 2002



# Chapter 26: Script Files

A file containing text in the form of lines of J is called a script-file, or just a script. By convention a script has a filename terminating with `.ijs`. The process of executing the lines of J in a script-file is called "loading" a script.

We write our own scripts for our particular programming projects. In addition, the J system comes supplied with a library of predefined scripts of general utility.

We look first at built-in functions for loading scripts, and then at a few utilities.

## 26.1 Creating Scripts

Here is an example of a tiny script. It is supposed to show a number of definitions and then a computation. The lines of J look like this:

```
plus =: +
k     =: 2 plus 3
k plus 1
```

Scripts are usually created using a text editor, but here it is convenient to use J to create small examples of scripts as we need them. We can create this script with a filename of say `example.ijs` using the built-in verb `1!:2`, like this:

```
(0 : 0) (1!:2) < 'c:\example.ijs'
plus =: +
k     =: 2 plus 3
k plus 1
)
```

## 26.2 Loading Scripts

There is a built-in verb `0!:1` to load a script. The argument is the file-name as a boxed string.

```
0!:1 < 'c:\example.ijs'
```

```

plus =: +
k     =: 2 plus 3
k plus 1
6

```

We see on the screen a transcript, or execution log, showing the lines of the script as they were executed, together with the result-values of any computations. The definitions of `plus` and `k` are now available:

plus	k
+	5

Now we look at some variations on this basic theme.

## 26.3 With or Without a Transcript

To suppress the transcript we can load with `0!:0`

```
0!:0 <'c:\example.ijs'
```

We see nothing on the screen. The value computed in the script for `k plus 1` is discarded. As well as `0!:0` and `0!:1` there are more variations of `0!:n` - see the Dictionary.

## 26.4 Local Assignments in Scripts

Here is an example of a script.

```

(0 : 0) (1!:2) < 'c:\exa.ijs'
w      =: 1 + 1
foo    =: + & w
)

```

Suppose that variable `w` has the sole purpose of helping to define verb `foo` and otherwise `w` is of no interest. It would be better to make `w` a local variable.

`w` can, in effect, be made local to the execution of the script on two conditions. The first is that we assign to `w` with `=.` in the same way that we assign to local variables in explicit functions. Our revised script becomes:

```
(0 : 0) (1! : 2) < 'c:\exb.ijs'
w      =. 1 + 1
foo =: + & w
)
```

The second condition is that we load the script inside an explicit function, so there is something for `w` to be local to. (Outside any explicit definition that is, "at the top level", `=.` is the same as `=:`)

All that is needed is the merest wrapper of explicit definition around `0! : 0` or `0! : 1`. A suitable "localizing loader" verb might be:

```
LL =: 3 : '0! : 0 y.'
```

If we now load this script

```
LL < 'c:\exb.ijs'
```

and then look at the results:

foo	w
+&2	error

we see that `foo` is as expected, but there is no value for `w`. Therefore `w` was local to the execution of the script, or strictly speaking, local to the execution of `LL`.

## 26.5 Local Verbs in Scripts

In the previous example, the local variable `w` was a noun. With a local verb, there is

a problem. Here is an example of a script which tries to use a local verb (`sum`) to assist the definition of a global verb (`mean`).

```
(0 : 0) (1!:2) < 'c:\exc.ijs'
sum =. +/
mean =: sum % #
)

LL < 'c:\exc.ijs'
```

We see that this will not work, because `mean` needs `sum` and `sum`, being local, is no longer available.

```
mean
sum % #
```

The remedy is to "fix" the definition of `mean`, with the adverb `f.` (as we did in [Chapter 12 p12](#)). Our revised script becomes

```
(0 : 0) (1!:2) < 'c:\exd.ijs'
sum =. +/
mean =: (sum % #) f.
)
```

Now `mean` is independent of `sum`

```
LL < 'c:\exd.ijs'
mean
+/ % #
```

## 26.6 Loading Into Locales

We looked at locales in [Chapter 24 p23](#). When we load a script with `0! : 0` or `LL` the locale that becomes populated with definitions from the script is the current locale.

By default, the current locale is `base`. In general, we may wish to load a script into a specified locale, say locale `f00`. Evidently we can do this by switching to locale `f00` loading and switching back to the base locale.

```
18! : 4 'f00'
```

```
0!:0 <'c:\example.ijs'
18!:4 'base'
```

A neater way is as follows. We define a loading verb, `LLL` say, which is just like `LL` above but this time we install it in locale `z`.

```
LLL_z_ =: 3 : '0!:0 y.'
```

Now we can load a script into a specified locale, `foo` say, with:

```
LLL_foo_ <'c:\example.ijs'
```

## 26.7 Library Scripts

The J system comes supplied with a useful library of script files containing predefined utility functions. Library script files are organized in a set of directories. The topmost of these library directories is identified by the expression `(1!:42 '')` which yields a pathname as a string:

```
1!:42 ''
c:\j\
```

Within this topmost library directory, a typical library script-file might be, for example, `system\main\dates.ijs`. This contains functions for handling calendar dates, and can of course loaded with `0!:0` (although we will see below there is a better way.)

```
0!:0 < (1!:42 ''), 'system\main\dates.ijs'
```

We can inspect one of the utility functions just loaded:

```
weekday
7: | 3: + todayno
```

## 26.8 The Profile

A J session begins with the automatic behind-the-scenes loading of a script file called the "profile". The contents of the profile can be whatever we choose - whatever function definitions or other things we find convenient to have on hand as our regular setup at the beginning of a session. Commonly a profile itself loads a further selection of library scripts and our own scripts.

The profile to be used is specified in the operating-system command-line initiating the J session. If no profile is specified in the command-line, then a standard system-supplied profile is used. In this case a session begins with the automatic execution of:

```
0!:0 < (1!:42 ''), 'system\extras\config\profile.ijs'
```

Loading this standard profile will load a further standard selection of library scripts, to give a set of commonly useful predefined verbs. The user can customize the standard profile to load further scripts.

### 26.8.1 load and loadadd

Among these useful predefined verbs is `load`. Its effect is the same as `0!:0`, that is, to load a script without displaying a transcript.

```
load 'c:\example.ijs'
```

Notice that the argument filename above can be a plain string, not boxed. A companion verb is `loadadd`, which, like `0!:1` loads a script displaying a transcript.

`load` and `loadadd` have several advantages over `0!:0` and `0!:1`. The first advantage of `load` is that for a library script the full pathname is not needed: a simple name is enough. Instead of

```
load (1!:42 ''), 'system\main\dates.ijs'
```

it is enough to say

```
load 'dates'
```

To achieve this effect, `load` and `loadadd` use a predefined catalog of all the library scripts, set up in the course of executing the standard profile. The catalog is a noun - a boxed table - named `PUBLIC_j_`. There are many scripts; the first 7 are shown by:

```

7 { . PUBLIC_j_
+-----+-----+-----+-----+-----+-----+
| colib   | c:\j\system\main\colib.ijs          | z |
+-----+-----+-----+-----+-----+-----+
| color16 | c:\j\system\packages\color\color16.ijs | z |
+-----+-----+-----+-----+-----+-----+
| colortab| c:\j\system\packages\color\colortab.ijs| z |
+-----+-----+-----+-----+-----+-----+
| compare | c:\j\system\main\compare.ijs        | z |
+-----+-----+-----+-----+-----+-----+
| convert | c:\j\system\main\convert.ijs        | z |
+-----+-----+-----+-----+-----+-----+
| dates   | c:\j\system\main\dates.ijs          | z |
+-----+-----+-----+-----+-----+-----+
| dd      | c:\j\system\main\dd.ijs             | z |
+-----+-----+-----+-----+-----+-----+

```

Each row has a simple name, the associated full path-name for the script, and finally a locale, usually `z` or `j`, into which the script will be loaded. `PUBLIC_j_` is itself set up from a file `system\extras\config\scripts.ijs`. This file can be edited by the user. Thus the second advantage of `load` is that a script can be automatically steered into an appropriate locale on loading.

The third advantage of `load` is this. Suppose one script depends on (definitions in) a second script. If the first includes a line such as `load 'second'` then the second is automatically loaded when the first is loaded.

If we load the first script again (say, after correcting an error) then the second will be loaded again. This may be unnecessary or undesirable. To avoid repeated loading of the second script we can `require` it rather than `load` it, that is load it only if not already loaded.

Here is a demonstration. Suppose we have these two lines for the first script:

```
(0 : 0) (1!:2) < 'c:\first.ijs'
```

```

    require 'c:\second.ijs'
    a =: a + 1
)

```

Here the variable `a` is a counter: every time `first.ijs` is loaded, `a` will be incremented. Similarly for a second script:

```

    (0 : 0) (1!!2) < 'c:\second.ijs'
    b =: b + 1
)

```

We set the counters `a` and `b` to zero, load the first script and inspect the counters:

<code>(a =: 0), (b =: 0)</code>	<code>load 'c:\first.ijs'</code>	<code>a,b</code>
0 0		1 1

Evidently each script has executed once. If we now load the first again, we see that it has executed again, but the second has not:

<code>load 'c:\first.ijs'</code>	<code>a,b</code>
	2 1

The effect is achieved by automatically tracking what has been loaded with `load` or `loadadd` in a table called `LOADED_j_`.

```

LOADED_j_
+-----+
|c:\book\tools\mnemonic.ijs| |
+-----+
|c:\j\system\main\files.ijs|z|
+-----+
|c:\example.ijs           | |
+-----+
|c:\j\system\main\dates.ijs|z|
+-----+

```



```

|c:\first.ijs          | |
+-----+-----+
|c:\second.ijs         | |
+-----+-----+

```

The fourth advantage of `load` is that it respects local assignments (because at the heart of `load` is `0! : 0`).

The fifth advantage of `load` is that an optional left argument can specify a locale into which to load the script. For example:

```

'mylocale' load 'c:\example.ijs'

plus_mylocale_
+

```

## 26.8.2 More on Load Status

We saw above that the J system maintains a record of which scripts have been loaded with the `load` verb. There is another separate system which keeps track of ALL scripts loaded, whether with `load` or with `0! : 0`. The built-in verb `4 !: 3` with a null argument gives a report as a boxed list of filenames.

```

, . 4 !: 3 ''
+-----+
|c:\book\tools\pr1406.ijs      |
+-----+
|c:\j\system\extras\util\boot.ijs |
+-----+
|c:\j\system\main\stdlib.ijs   |
+-----+
|c:\j\system\main\winlib.ijs   |
+-----+
|c:\j\system\main\colib.ijs    |
+-----+
|c:\j\system\main\loadlib.ijs  |
+-----+
|c:\j\system\main\jadelib.ijs  |
+-----+
|c:\j\system\extras\config\scripts.ijs|
+-----+
|c:\j\system\extras\config\winpos.ijs |
+-----+

```

```

|c:\j\system\extras\util\configur.ijs |
+-----+
|c:\j\system\extras\config\config.ijs |
+-----+
|c:\book\tools\pr2406.ijs             |
+-----+
|c:\book\tools\mnemonic.ijs           |
+-----+
|c:\j\system\main\files.ijs           |
+-----+
|c:\book\work\current.ijs             |
+-----+
|c:\example.ijs                       |
+-----+
|c:\exb.ijs                           |
+-----+
|c:\exc.ijs                           |
+-----+
|c:\exd.ijs                           |
+-----+
|c:\j\system\main\dates.ijs           |
+-----+
|c:\first.ijs                         |
+-----+
|c:\second.ijs                        |
+-----+

```

We see some scripts loaded by the standard profile, and others particular to this session. Recall that we defined `plus` in the script `example.ijs` which we loaded above. The built-in verb `4!:4` keeps track of which name was loaded from which script. The argument is a name (`plus` for example) and the result is an index into the list of scripts generated by `4!:3`. We see that `plus` was indeed defined by loading the script `example.ijs`

<code>i =: 4!:4 &lt; 'plus'</code>	<code>i { 4!:3 ''</code>
15	<pre> +-----+  c:\example.ijs  +-----+ </pre>

## 26.8.3 Summary

The recommendation is :

- Use the standard profile, or the standard profile with additional customizing.  
This ensures that a session begins having loaded the standard library scripts.
- Use `load`, `loadadd` or `require` for loading scripts.

This is the end of Chapter 26.

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 15 Mar 2002

# Chapter 31: Evaluating Expressions

## 31.1 Introduction

In this chapter we look at the process of evaluating a J expression. Evaluating a complete expression proceeds by a sequence of basic steps, such as obtaining the value assigned to a name, or applying a function to its argument(s). For example, given

```
x =: 3
```

then the expression

```
4+5*x
```

```
19
```

is (in outline) evaluated by the steps:

1. obtain the value assigned to `x` giving 3
2. compute `5 * 3` giving 15
3. compute `4 + 15` giving 19

The sequence in which the steps take place is governed by the grammatical (or "parsing") rules of the J language. The parsing rules have various consequences, or effects, which can be stated informally, for example:

- verbs have long right scope (this is the "rightmost-first" rule we saw above)
- verbs have short left scope
- adverbs and conjunctions get applied before verbs
- adverbs and conjunctions have long left scope and short right scope
- names denoting nouns are evaluated as soon as encountered
- names denoting functions are not evaluated until the function is applied
- names with no assigned values are assumed to denote verbs
- long trains of verbs are resolved into trains of length 2 or 3

and we will look at how the parsing rules give rise to these effects. To illustrate the process, we can use a function which models, or simulates, the evaluation process step by step, showing it at work in slow motion.

This function, an adverb called `EVM`, is based on the description of the parsing algorithm given in the J Dictionary, section IIE. It is defined in a downloadable J script.

## 31.2 First Example

Evaluation of an expression such as `2+3` can be modelled by offering the argument `'2+3'` (a string, notice) to the modelling adverb `EVM`.

2+3	'2+3' EVM
5	5

We see that `'2+3' EVM` computes the same value as `2+3`, but `EVM` also produces a trace, or history, of the evaluation process. The history of `2+3` looks like this:

```
show ''
```

		queue		stack					rule
0		§ 2 + 3							
1		§ 2 +		3					
2		§ 2		+	3				
3		§		2	+	3			
4				§	2	+	3		dyad



```

+-----+-----+-----+-----+
|       | | | | mark | 5 | | |       |
+-----+-----+-----+-----+

```

However, a more readable display is produced by the `show` function which computes, from `Qh Sh` and `Rh`, a fragment of HTML. This HTML is not for viewing in the execution window but rather for pasting into a web page such as this one. Corresponding to `Qh Sh` and `Rh` as above we would see:

```
show ' '
```

		queue		stack					rule
0		§ 2 + 3							
1		§ 2 +		3					
2		§ 2		+	3				
3		§		2	+	3			
4				§	2	+	3		dyad
5				§	5				

## 31.3 Parsing Rules

In this section an example is shown of each of the 9 parsing rules. Each rule looks for a pattern of items at the front of the stack, such as something verb noun verb. Each item of the stack is classified as one of the following: verb, noun, adjective, conjunction, name, left-parenthesis, right-parenthesis, assignment-symbol (= . or =:) or beginning-mark.

To aid in a compact statement of the rules, larger classes of items can be formed. For example, an item is classified as an "EDGE" if it is a beginning-mark, an assignment-symbol or a left-parenthesis.

The rules are always tried in the same order, the order in which they are presented

below, beginning with the 'monad rule' and ending with the 'parenthesis rule'.

### 31.3.1 Monad Rule

If the first 3 items of the stack are an "EDGE" followed by a verb followed by a noun, then the verb is applied (monadically) to the noun to give a result-value symbolized by z say, and the value z replaces the verb and noun in the stack. The scheme for transforming the items of the stack is:

monad rule: EDGE VERB NOUN etc => EDGE Z etc

where z is the result computed by applying VERB to NOUN. For example:

*: 4	' *: 4 ' EVM
16	16

show ' '

		queue		stack				rule
0		§ *: 4						
1		§ *:		4				
2		§		*:	4			
3				§	*:	4		monad
4				§	16			

### 31.3.2 Second Monad Rule

An item in the stack is classified as "EAVN" if it is an EDGE or an adverb or verb



or noun. The scheme is:

```
monad2 rule: EAVN VERB1 VERB2 NOUN etc => EAVN VERB1 Z etc
```

where z is VERB2 monadically applied to NOUN. For example:

- *: 4	'- *: 4' EVM
_16	_16

```
show ''
```

	queue		stack					rule
0	§ - *: 4							
1	§ - *:		4					
2	§ -		*:	4				
3	§		-	*:	4			
4			§	-	*:	4		monad2
5			§	-	16			monad
6			§	_16				

### 31.3.3 Dyad Rule

The scheme is

```
dyad rule: EAVN NOUN1 VERB NOUN2 etc => EAVN Z etc
```

where z is VERB applied dyadically to NOUN1 and NOUN2. For example.

3 * 4	'3 * 4' EVM
12	12

show ' '

		queue		stack					rule
0		§ 3 * 4							
1		§ 3 *		4					
2		§ 3		*	4				
3		§		3	*	4			
4				§	3	*	4		dyad
5				§	12				

31.3.4 Adverb Rule

An item which is a verb or a noun is classified as a "VN" The scheme is:  
adverb rule: EAVN VN ADVERB etc => EAVN Z etc

where z is the result of applying ADVERB to VN. For example:

+ / 1 2 3	'+ / 1 2 3' EVM
-----------	-----------------

6	6
---	---

show ''

	queue		stack					rule
0	§ + / 1 2 3							
1	§ + /		1 2 3					
2	§ +		/	1 2 3				
3	§		+	/	1 2 3			
4			§	+	/	1 2 3		adv
5			§	+/	1 2 3			monad
6			§	6				

### 31.3.5 Conjunction Rule

The scheme is:

conjunction EAVN VN1 CONJ VN1 etc => EAVN Z etc

where z is the result of applying conjunction CONJ to arguments VN1 and VN2. For example:

1 & + 2	'1 & + 2' EVM
3	3

show ''

		queue		stack						rule
0		§ 1 & + 2								
1		§ 1 & +		2						
2		§ 1 &		+	2					
3		§ 1		&	+	2				
4		§		1	&	+	2			
5				§	1	&	+	2		conj
6				§	1&+	2				monad
7				§	3					

31.3.6 Trident Rule

The scheme is:  
trident rule: EAVN VERB1 VERB2 VERB3 etc => EAVN Z etc

where z is a single verb defined as the fork (VERB1 VERB2 VERB3). For example:  
f=: +/  
g=: %  
h=: #

(f g h) 1 2	'(f g h) 1 2' EVM
1.5	1.5

show ''

		queue		stack							rule
0		§ ( f g h ) 1 2									
1		§ ( f g h )		1 2							
2		§ ( f g h		)	1 2						
3		§ ( f g		h	)	1 2					
4		§ ( f		g	h	)	1 2				
5		§ (		f	g	h	)	1 2			
6		§		(	f	g	h	)	1 2		trident
7		§		(	f g h	)	1 2				paren
8		§		f g h	1 2						
9				§	f g h	1 2					monad
10				§	1.5						

### 31.3.7 Bident Rule

The scheme is:

bident rule: EDGE CAVN1 CAVN2 etc => EDGE Z etc

and there are altogether these 6 cases for the bident rule:

CAVN1	CAVN2	Z
verb	verb	verb (a hook)
adverb	adverb	adverb
conjunction	verb	adverb
conjunction	noun	adverb
noun	conjunction	adverb
verb	conjunction	adverb

The first case (the hook) is described in [Chapter 03 p3](#) and the remaining cases in the schemes for bidents in [Chapter 15 p15](#).

In the following example the expression ( 1 & ) is a bident of the form noun conjunction. Therefore it is an adverb.

+ ( 1 & ) 2	' + ( 1 & ) 2 ' EVM
3	3

show ' '

		queue		stack							rule
0		§ + ( 1 & ) 2									
1		§ + ( 1 & )		2							
2		§ + ( 1 &		)	2						
3		§ + ( 1		&	)	2					

4		§ + (		1	&	)	2			
5		§ +		(	1	&	)	2		bident
6		§ +		(	1&	)	2			paren
7		§ +		1&	2					
8		§		+	1&	2				
9				§	+	1&	2			adv
10				§	1&+	2				monad
11				§	3					

31.3.8 Assignment Rule

We write `NN` to denote a noun or a name. and `Asgn` for the assignment symbol `=:` or `=..`. The scheme is:

assign rule: `NN Asgn CAVN etc => Z etc`

where `z` is the value of `CAVN`.

1 + x =: 6	'1 + x =: 6' EVM
7	7

show ''

	queue		stack		rule
0	§ 1 + x =: 6				





1		§ ( 1 + 2 ) *		3								
2		§ ( 1 + 2 )		*	3							
3		§ ( 1 + 2		)	*	3						
4		§ ( 1 +		2	)	*	3					
5		§ ( 1		+	2	)	*	3				
6		§ (		1	+	2	)	*	3			
7		§		(	1	+	2	)	*	3		dyad
8		§		(	3	)	*	3				paren
9		§		3	*	3						
10				§	3	*	3					dyad
11				§	9							

### 31.3.10 Examples of Transfer

The following example shows that when a name is transferred from queue to stack, if the name denotes a value which is a noun, then the value, not the name, moves to the queue.

a =: 6	(a=:7) , a
6	7 6

a=: 6	'(a =: 7) , a' EVM
6	7 6

show ''

	queue		stack							rule
0	§ ( a =: 7 ) , a									
1	§ ( a =: 7 ) ,		6							
2	§ ( a =: 7 )		,	6						
3	§ ( a =: 7		)	,	6					
4	§ ( a =:		7	)	,	6				
5	§ ( a		=:	7	)	,	6			
6	§ (		a	=:	7	)	,	6		assign
7	§ (		7	)	,	6				
8	§		(	7	)	,	6			paren
9	§		7	,	6					
10			§	7	,	6				dyad
11			§	7 6						

By contrast, if the name is that of a verb, then the name is transferred into the stack without evaluating it. Hence a subsequent assignment changes the verb applied.

f=: +	((f=: -) , f) 4
+	_4 _4

f =: +	'((f =: -),f) 4' EVM
+	_4 _4

show ''

		queue		stack									rule
0		§ ( (f =: -) , f ) 4											
1		§ ( (f =: -) , f )		4									
2		§ ( (f =: -) , f		)	4								
3		§ ( (f =: -) ,		f	)	4							
4		§ ( (f =: -)		,	f	)	4						
5		§ ( (f =: -		)	,	f	)	4					

6		§ ( (		-	)	,	f	)	4				
7		§ ( (		=:	-	)	,	f	)	4			
8		§ ( (		f	=:	-	)	,	f	)	4		assign
9		§ ( (		-	)	,	f	)	4				
10		§ (		(	-	)	,	f	)	4			paren
11		§ (		-	,	f	)	4					
12		§		(	-	,	f	)	4				trident
13		§		(	<sup>-</sup> f	,	)	4					paren
14		§		<sup>-</sup> f	,	4							
15				§	<sup>-</sup> f	,	4						monad
16				§	<sup>-4</sup> <sub>-4</sub>								

### 31.3.11 Review of Parsing Rules

rule	stack before				stack after				where Z is ...
monad	EDGE	Verb	Noun	etc	EDGE	Z	etc		Verb applied to Noun
monad2	EAVN	Verb1	Verb2	Noun	EAVN	Verb1	Z		Verb2 applied to Noun

dyad		EAVN	Noun1	Verb	Noun2		EAVN	Z	etc	Verb applied to Noun1 and Noun2
adverb		EAVN	VN	Adv	etc		EAVN	Z	etc	Adv applied to VN
conj		EAVN	VN1	Conj	VN2		EAVN	Z	etc	Conj applied to VN1 and VN2
trident		EAVN	Verb1	Verb2	Verb3		EAVN	Z	etc	fork (Verb1 Verb2 Verb3)
bident		EDGE	CAVN1	CAVN2	etc		EDGE	Z	etc	bident (CAVN1 CAVN2)
assign		NN	Asgn	CAVN	etc		Z	etc	etc	CAVN
paren		(	CAVN	)	etc		Z	etc	etc	CAVN

## 31.4 Effects of Parsing Rules

Now we look at some of the effects, of the parsing rules. In what follows, notice how the parsing rules in effect give rise to implicit parentheses.

### 31.4.1 Dyad Has Long Right Scope

Consider the expression  $4+3-2$ , which means  $4+(3-2)$ .

4 + 3 - 2	4 + (3-2)	'4+3-2' EVM
5	5	5

show ' '

		queue		stack					rule
0		§ 4 + 3 - 2							
1		§ 4 + 3 -		2					
2		§ 4 + 3		-	2				
3		§ 4 +		3	-	2			
4		§ 4		+	3	-	2		dyad
5		§ 4		+	1				
6		§		4	+	1			
7				§	4	+	1		dyad
8				§	5				

Here we have an example of a general rule: a dyadic verb takes as its right argument as much as possible, so in this example + is applied to 3-2, not just 3.

Further, a dyadic verb takes as left argument as little as possible. In this example the left argument of - is just 3, not 4+3. Hence a dyadic verb is said to have a "long right scope" and a "short left scope".

### 31.4.2 Operators Before Verbs

Adverbs and conjunctions get applied first, and then the resulting verbs:

* & 1 % 2	* & ( 1 % 2 )	( *&1 ) % 2
0.5	*&(0.5)	0.5

```
'* & 1 % 2' EVM
0.5
show ''
```

	queue		stack							rule
0	§ * & 1 % 2									
1	§ * & 1 %		2							
2	§ * & 1		%	2						
3	§ * &		1	%	2					
4	§ *		&	1	%	2				
5	§		*	&	1	%	2			
6			§	*	&	1	%	2		conj
7			§	*&1	%	2				monad2
8			§	*&1	0.5					monad
9			§	0.5						

### 31.4.3 Operators Have Long Left Scope

An adverb or a conjunction takes as its left argument as much as possible. In the following, look at the structure of the resulting verbs: evidently the / adverb and the @ conjunction take everything to their left:

f @ g /	f & g @ h	'f&g@h' EVM
(f@g)/	(f&g)@h	(f&g)@h

show ' '

	queue		stack								rule
0	§ f & g @ h										
1	§ f & g @		h								
2	§ f & g		@	h							
3	§ f &		g	@	h						
4	§ f		&	g	@	h					
5	§		f	&	g	@	h				
6			§	f	&	g	@	h			conj
7			§	f&g	@	h					conj
8			§	(f&g)@h							

Thus operators are said to have a "long left scope". In the example of  $f \& g @ h$  we see that the right argument of  $\&$  is just  $g$ , not  $g @ h$ . Thus conjunctions have "short right scope".

### 31.4.4 Train on the Left

The long left scope of an adverb does not extend through a train: parentheses may



be needed to get the desired effect. Suppose  $f \ g \ h$  is intended as a train, then compare the following:

$f \ g \ h \ /$	$(f \ g \ h) \ /$
$f \ g \ (h/)$	$(f \ g \ h)/$

```
'f g h / ' EVM
f g h/
show ''
```

	queue		stack					rule
0	§ f g h /							
1	§ f g h		/					
2	§ f g		h	/				
3	§ f		g	h	/			adv
4	§ f		g	h/				
5	§		f	g	h/			
6			§	f	g	h/		trident
7			§	f g (h/)				

Similarly for a conjunction (with a right argument)

f g h @ +	'f g h @ +' EVM
f g (h@+)	f g (h@+)

show ''

	queue		stack					rule
0	§ f g h @ +							
1	§ f g h @		+					
2	§ f g h		@	+				
3	§ f g		h	@	+			
4	§ f		g	h	@	+		conj
5	§ f		g	h@+				
6	§		f	g	h@+			
7			§	f	g	h@+		trident
8			§	f g (h@+)				

However, for a conjunction with no right argument, the left scope does extend through a train:

f g h @	'f g h @' EVM
(f g h)@	(f g h)@

show ''

		queue		stack						rule
0		§ f g h @								
1		§ f g h		@						
2		§ f g		h	@					
3		§ f		g	h	@				
4		§		f	g	h	@			
5				§	f	g	h	@		trident
6				§	f g h	@				bident
7				§	(f g h)@					

By contrast, in the case of of  $f @ g /$ , notice how the "conj" rule is applied before there is a chance to apply the "adverb" rule"

f @ g /	'f @ g / ' EVM
(f@g)/	(f@g)/

show ''

		queue		stack						rule
0		§ f @ g /								
1		§ f @ g		/						

2		§ f @		g /					
3		§ f		@ g	/				
4		§		f @	g /				
5				§ f	@ g /				conj
6				§ f@g	/				adv
7				§ (f@g)/					

31.4.5 Presumption of Verb

A name with no value assigned is presumed to be a verb. For example, in the following the three names make a fork:

Maple Leaf Rag	'Maple Leaf Rag' EVM
Maple Leaf Rag	Maple Leaf Rag

show ' '

		queue		stack					rule
0		§ Maple Leaf Rag							
1		§ Maple Leaf		Rag					
2		§ Maple		Leaf Rag					
3		§		Maple Leaf Rag					

4			§	Maple	Leaf	Rag		trident
5			§	Maple Leaf Rag				

This is the end of Chapter 31

---

Copyright © Roger Stokes 2002. This material may be freely reproduced, provided that this copyright notice is also reproduced.

last updated 4 Aug 2002