Learning J

An Introduction to the J Programming Language

Roger Stokes

revised July 2013

J software and documentation are available at the J Software Home Page

This book is also available in various formats from here

Please send comments and criticisms to the J Forum

ABOUT THIS BOOK

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

My hope is that the book will be useful 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 the examples very simple. Even so, the experienced programmer will find much to appreciate in the radical simplicity and power of the J notation.

The scope of this book is the core J language defined in the J Dictionary. The coverage of the core language is meant to be relatively complete, covering (eventually) most of the Dictionary.

Hence the book does not cover topics such as graphics, plotting, GUI, and database covered in the J User Guide, nor does it cover the J Application Library . I should make clear what the aims of the book are not: neither to teach principles of programming, 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 organized as follows. Part 1 is a basic 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.

All the examples have been executed with J701.

TABLE OF CONTENTS

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

Chapter 1: Basics

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).

```
   3 % 4
0.75
```

For subtraction, we have the familiar - symbol:

```
   3 - 2
1
```

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.

```
   2 - 3
_1
```

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

```
   - 3
_3
```

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

```
   2 ^ 3
8
```

The arithmetic function to compute the square of a number has

the symbol `*:` (asterisk colon).

```
   *: 4
16
```

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 | 7)` rather than `(7 mod 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.

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
99
```

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 convey a little of the flavour 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

```
   * / 2 3 4
24
```

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, .... y, z` then:

```
F / L     means     a F b F .... 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 and "false" by the number `0`. The three propositions are rendered in J as:

```
   2 > 1
1

   2 = 1
0

   2 < 1
0
```

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). There is a function, dyadic `#`, which can take a bit-string (a sequence of choices) as left argument to select chosen items from the right argument.

`y =: 6 7 8 9 10`	`1 1 0 1 0 # y`
`6 7 8 9 10`	`6 7 9`

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

`y`	`y > 7`	`(y > 7) # y`
`6 7 8 9 10`	`0 0 1 1 1`	`8 9 10`

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 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 >.
6
```

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
```

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.

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.

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:

`2 $ 99`	`0 $ 99`	`$ 17`
`99 99`		

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`	`1` `7`	`1` `7`	`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.

`t =: 3 1 $ 5 6 7`	`$ t`	`# $ t`

`5`
`6`
`7`
	`3 1`	`2`

2.3 Terminology: Rank and Shape

The property we called "dimension-count" is in J called by the shorter name of of "rank", so a single number is a 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?
```

An empty, or zero-length, string is entered as two successive single quotes, and displays as nothing.

`''`	`# ''`
	`0`

2.5 Some Functions for Arrays

At this point it will be useful to look 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
rear          ranged          rearranged
```

The "Append" function joins lists or single items.

```
x =: 1 2 3   0 , x     x , 0       0 , 0   x , x
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                   rat                   cat
dog                   pig                   dog
                                            rat
                                            pig
```

For more information about "Append", see Chapter 05.

2.5.2 Items

The items of a list of numbers are the individual numbers, and we will say that the items of a table are its rows. The items of a 3-dimensional array are its planes. In general we will say that the items of an array are the things which appear in sequence along its first dimension. An array is the list of its items.

Recall the built-in verb `#` ("Tally") which gives the length of a list.

`x`	`# x`
`1 2 3`	`3`

In general `#` counts the number of items of an array, that is, it gives the first dimension:

`T1`	`$ T1`	`# T1`
`cat` `dog`	`2 3`	`2`

Evidently `# T1` is the first item of the list-of-dimensions `$ T1`. A scalar, with no dimensions, is regarded as a single item:

```
   # 6
1
```

Consider again the example of "Append" given above.

T1	T2	T1 , T2
cat dog	rat pig	cat dog rat pig

Now we can say that in general `(x , y)` is a list consisting of the items of `x` followed by the items of `y`.

For another example of the usefulness of "items", recall the verb `+/` where `+` is inserted between items of a list.

+/ 1 2 3	1 + 2 + 3
6	6

Now we can say that in general `+/` inserts `+` between items of an array. In the next example the items are the rows:

T =: 3 2 $ 1 2 3 4 5 6	+/ T	1 2 + 3 4 + 5 6
1 2 3 4 5 6	9 12	9 12

2.5.3 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 function `{` (left brace, called "From") .

`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.

2.5.4 Equality and Matching

Suppose we wish to determine whether two arrays are the same. There is a built-in verb `-:` (minus colon, called "Match"). It tests whether its two arguments have the same shapes and the same values for corresponding elements.

`X =: 'abc'`	`X -: X`	`Y =: 1 2 3 4`	`X -: Y`
`abc`	`1`	`1 2 3 4`	`0`

Whatever the arguments, the result of Match is always a single `0` or `1`.

Notice that an empty list of, say, characters is regarded as matching an empty list of numbers:

```
    '' -: 0 $ 0
1
```

because they have the same shapes, and furthermore it is true that all corresponding elements have the same values, (because there are no such elements).

There is another verb, `=` (called "Equal") which tests its arguments for equality. `=` compares its arguments element by element and produces an array of booleans of the same shape as the arguments.

`Y`	`Y = Y`	`Y = 2`
`1 2 3 4`	`1 1 1 1`	`0 1 0 0`

Consequently, the two arguments of `=` must have the same shapes, (or at least, as in the example of `Y=2`, compatible shapes). Otherwise an error results.

`Y`	`Y = 1 5 6 4`	`Y = 1 5 6`
`1 2 3 4`	`1 0 0 1`	`error`

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.

`n =: 42`	`s =: ": n`	`# s`	`'answer is ' , s`
`42`	`42`	`2`	`answer is 42`

For more about "Format", see Chapter 19. 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 often 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).

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
```

We might choose to do this if we prefer our own name as more memorable. We can give two different names to the same built-in function, intending to use one for the monadic case and the other for the dyadic.

```
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 be called "operators", to distinguish them from verbs.

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~ y      means    y f x
```

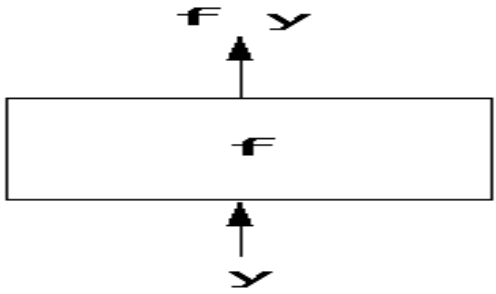
For another example, recall the residue verb `|` where `2 | 7`

means, in conventional notation, "7 mod 2". We can define a `mod` verb:

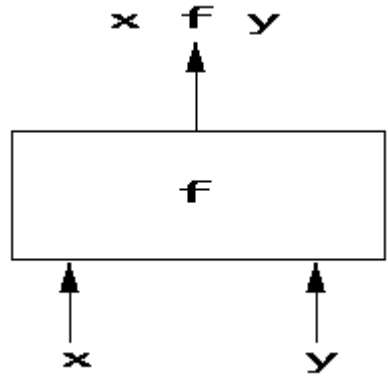
```
mod =: | ~
```

`7 mod 2`	`2	7`
`1`	`1`

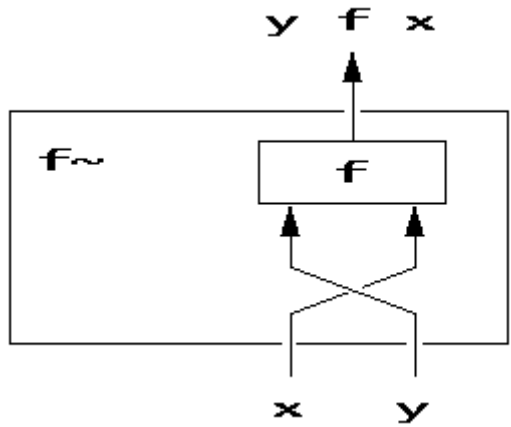
Let me draw some pictures. Firstly, here is a diagram of function `f` applied to an argument `y` to produce a result `(f y)`. In the diagram the function `f` is drawn as a rectangle and the arrows are arguments flowing into, or results flowing out of, the function. Each arrow is labelled with an expression.



Here is a similar diagram for a dyadic `f` applied to arguments `x` and `y` to produce `(x f y)`.



Here now is a diagram for the function `(f~)`, which can be pictured as containing inside itself the function `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 is said to form a bond between a function and a value for one argument. The scheme is: if `f` is a dyadic function, and `k` is a value for the right argument of `f`, then

```
(f & k) y    means    y f k
```

Instead of fixing the right argument we could fix the left, so we also have the scheme

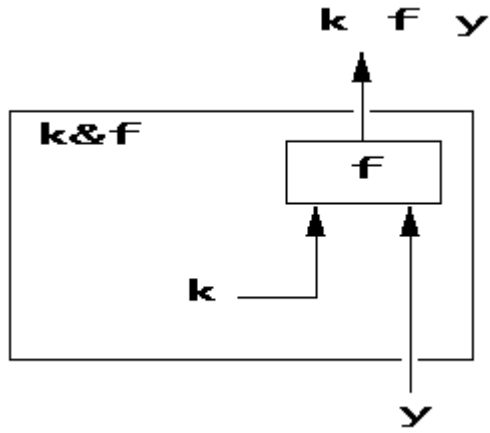
```
(k & f)  y   means    k f y
```

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`:

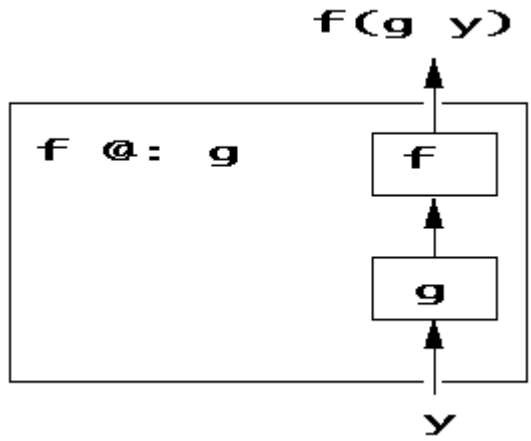
```
   sumsq =: sum @: square

   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    means  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` tomultiply 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   means   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.

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".

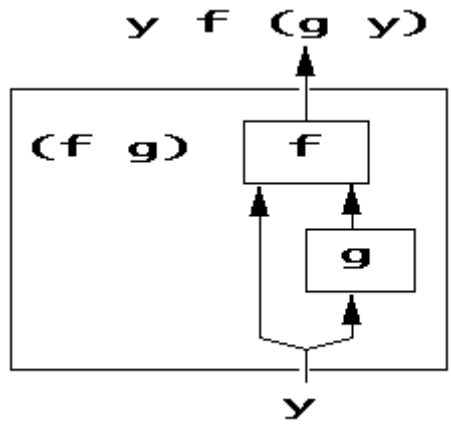
We can also form a hook just by isolating the two verbs inside parentheses:

```
   (+ 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  =:  = <.
```

y =: 3 2.7	<. y	y = <. y	wholenumber y
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 `#`.)

L =: 3 5 7 9	sum L	# L	(sum L) % (# L)
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 `#` .

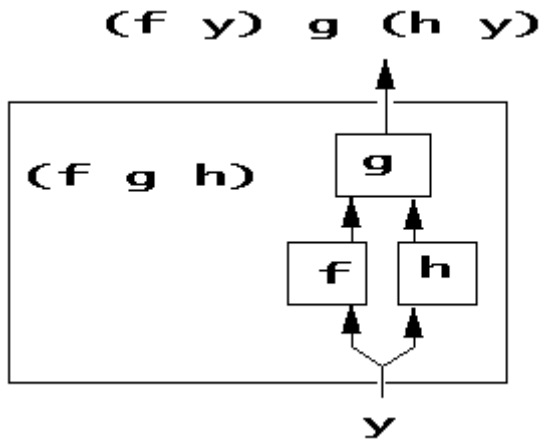
```
   mean =: sum % #

   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:



For another example of a fork, what is called the range of numbers in a list is given by the fork `smallest , largest` where the middle verb is the comma.

Recall from Chapter 01 that the largest number in a list is given by the verb `>./` and so the smallest will be given by `<./`

```
range =: <./  ,  >./
```

L	range L
3 5 7 9	3 9

Hooks and forks are sequences of verbs, also called "trains" of verbs. For more about trains, see Chapter 09.

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 7 lines, defining a verb called `display` and its supporting functions.

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

If we start with some very simple data:

```
data =: 3 1 4
```

then we see that the `display` verb shows each number as given and as a percentage (in round figures) of the total: `4` is `50%` of `8`.

```
   display data
+----+-----------+
|Data|Percentages|
+----+-----------+
|3   |38         |
|1   |13         |
|4   |50         |
+----+-----------+
```

First, we aim to divide each number by the total, to show the contribution of each as a fraction. The hook `(% +/)` is suitable: it can be read as divide-by-sum. If we call it `frac`

```
   frac =: % +/
```

then we see

data	+/data	data % (+/data)	frac data
3 1 4	8	0.375 0.125 0.5	0.375 0.125 0.5

The percentages are given by multiplying the fractions by 100.

```
   percent  =: (100 & *) @: frac
```

data	frac data	percent data
3 1 4	0.375 0.125 0.5	37.5 12.5 50

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
```

percent data	round percent data	comp data
37.5 12.5 50	38 13 50	38 13 50

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 1 4	3 1 4	38 13 50

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 1 4	+-+--+ |3|38| |1|13| |4|50| +-+--+

To add the top row of the display (the column headings), there is a useful built-in verb `,:` (comma colon, "Laminate", which will be covered in Chapter 05)

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

data	br data	tr br data
3 1 4	+-+--+ |3|38| |1|13| |4|50| +-+--+	+----+-----------+ |Data|Percentages| +----+-----------+ |3 |38 | |1 |13 | |4 |50 | +----+-----------+

and so we put everything together:

```
   display =: tr @: br

   display data
+----+-----------+
|Data|Percentages|
+----+-----------+
|3   |38         |
|1   |13         |
|4   |50         |
+----+-----------+
```

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, a hook and a fork. As with all J programs, this is only one of the many possible ways to write it.

In this chapter we have taken a first look at defining functions. There are two kinds of functions: verbs and operators. So far we have looked only at defining verbs. In the next chapter we look at another way of defining verbs, and in Chapter 13 onwards we will look at defining operators.

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 `txt` 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 given by the expression `0 !: 1` which we can call, say, `do`.

```
   do =: 0 !: 1
```

Here the expression `0 !: 1` can be understood as the verb produced by giving a left argument of `0` and a right argument of `1` to the conjunction `!:` (exclamation colon, called the "Foreign Conjunction"). `!:` offers a set of utility functions or system services which are organised into groups of verbs. For more details, see the Dictionary here .

In this example the left argument of `0` specifies the script-executing group, and the right argument of `1` picks out a particular member of that group, namely a verb to execute the script to the end regardless of errors, and displaying the execution on screen.

If we now enter `do script` we should now see the lines on the screen just as though they had been typed in from the keyboard:

```
   do script
   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`

The main features of this definition are:

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. 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` . This "argument variable" is named `y` in a monadic function. (In a dyadic function, as we shall see below, the left argument is named `x` and the right is `y`)

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 Control Structures

In the examples we have seen so far of functions defined by scripts, execution begins with the expression on the first line, proceeds to the next line, and so on to the last.

This straight-through path is not the only path possible. A choice can be made as to which expression to execute next.

For an example, here is a function to compute a volume from given length, width and height. Suppose the function is to check that its argument is given correctly as a list of 3 items (length, width and height). If so, a volume is computed. If not, the result is to be the character-string `'ERROR'`.

```
   volume =: 3 : 0
if.   3 = # y
do.   * / y
else. 'ERROR'
end.
)
```

We see:

`volume 2 3 4`	`volume 2 3`
`24`	`ERROR`

Everything from `if.` to `end.` together forms what is called a "control structure". Within it `if. do. else.` and `end.` are called

"control words". See Chapter 12 for more on control structures.

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 `volume` 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, 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. This is the default, but we can choose from several other possibilities: see Chapter 27. For now I will mention only the "linear representation", which shows a function as a sequence of characters which could be typed in again to produce the function. We can switch the session to to show functions in the linear representation by entering:

```
   (9!:3) 5
```

and we see for example:

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

In the following chapters, values of functions will often be shown in this linear representation.

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. Using a text-editor of your choice, create a file on your computer, containing 2 lines of text like the following.

```
squareroot =: %:

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

A J script file has a filename ending with `.ijs` by convention, so suppose the file is created (in Windows) with the full pathname `c:\temp\myscript.ijs` for example.

Then in the J session it will be convenient to identify the file by defining a variable `F` say to hold this filename as a string.

```
F =: 'c:\temp\myscript.ijs'
```

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

```
0!:1 < F
```

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 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.

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

5.1.1 Review

Recall from Chapter 02 what we mean by the word "items". The items of a list of numbers are the numbers. The items of a table are its rows. The items of a 3-dimensional array are its planes.

Recall also that `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:

`2 2 $ 0 1 2 3`	`2 3 $ 'ABCDEF'`
`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.

`2 3 $ 'ABCD'`	`2 2 $ 1`	`3 3 $ 1 0 0 0`

`ABC`
`DAB`
	`1 1`
	`1 1`
		`1 0 0`
		`0 1 0`
		`0 0 1`

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`

For any array `A`, its list-of-dimensions `$ A` is a 1-dimensional list (the shape). Hence `$ $ A` is a list of 1 item (the rank). Hence `$ $ $ A` is always a list containing just the number `1`.

`A`	`$ A`	`$ $ A`	`$ $ $ A`
`ABC` `DEF`	`2 3`	`2`	`1`

5.1.2 Empty Arrays

An array can be of length zero in any of its dimensions. A zero length, or empty, list can be built by writing `0` for its list of dimensions, and any value (doesn't matter what) for the value of

the item(s).

E =: 0 $ 99	$ E
	0

If `E` is empty, then it has no items, and so, after appending an item to it, the result will have one item.

E	$ E	w =: E ,98	$ w
	0	98	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

5.1.3 Building a Scalar

Suppose 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:

S =: (0$0) $ 17	$ S	$ $ S
17		0

5.1.4 Shape More Generally

We said that `(x $ y)` produces an `x`-shaped array of the items of `y`. That is, in general the shape of `(x$y)` will be not just `x`, but rather `x` followed by the shape of an item of `y`.

If `y` is a table, then an item of `y` is a row, that is, a list. In the following example, the shape of an item of `Y` is the length of a row of `Y`, which is `4` .

`X =: 2`	`Y =: 3 4 $ 'A'`	`Z =: X $ Y`	`$ Z`
`2`	`AAAA` `AAAA` `AAAA`	`AAAA` `AAAA`	`2 4`

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	uv w	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 DEF	ABC DEF XY	1 1 1 1 1 1 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 dyadic verb `,.` (comma dot) is called "Stitch". In the expression `(x ,. y)` each item of `x` has the corresponding item of `y` appended to produce an item of the result.

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". The result of `(x ,: y)` is always an array with two items, of which the first is `x` and the second is `y`

a	b	a ,: b
pqr	uvw	pqr uvw

If x and y are tables, then we can imagine the result as one table laid on top of the other to form a 3-dimensional array, of length 2 along its first dimension.

A	B	A ,: B	$ A ,: B
ABC DEF	UVW XYZ	ABC DEF UVW XYZ	2 2 3

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|2|3|
+-+-+-+
```

```
1 foo 2 foo 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
+-+-+
|1|2|
+-+-+
|3|4|
+-+-+
   ; B
1 2 3 4
   $ ; B
4
```

5.6.2 Ravelling

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

```
   B
+-+-+
|1|2|
+-+-+
|3|4|
+-+-+
   , B
+-+-+-+-+
|1|2|3|4|
+-+-+-+-+
   $ , B
4
```

5.6.3 Ravelling Items

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

```
k =: 2 2 3 $ i. 12   ,. k
0  1  2              0 1 2 3  4  5
3  4  5              6 7 8 9 10 11

6  7  8
9 10 11
```

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

```
b     ,. b
uvw   u
      v
      w
```

5.6.4 Itemizing

Monadic `,:` is called "Itemize". It makes a 1-item array out of any array, by adding a leading dimension of `1`.

```
A     ,: A    $ ,: A
ABC   ABC     1 2 3
DEF   DEF
```

5.7 Arrays Large and Small

As we have seen, an array can be built with the `$` verb.

```
   3 2 $ 1 2 3 4 5 6
1 2
3 4
5 6
```

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.

`> 1 2 ; 3 4 ; 5 6`	`1 2 , 3 4 ,: 5 6`

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

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 02 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.

`L =: 'abcdef'`	`0 { L`	`1 { L`
`abcdef`	`a`	`b`

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

6.1.1 Common Patterns of Selection.

Several items may be selected together:

`L`	`0 2 4 { L`
`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      _1 { L  _2 1 { L
abcdef f       eb
```

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

```
      T =: 3 3 $ 'abcdefghi'  (< 1 ; 2) { T
abc                           f
def
ghi
```

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	(< 1 2;0 1) { T
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.

s =: 'pasta'	# s	z =: 8 {. s	# z
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)`.

L	2 }. L
abcdef	cdef

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

L	_2 {. L	_2 }. L
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").

L	{. L	}. L
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 **(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`	`(SuAx (Sel 1), (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`	`(SuAx (Sel 0), (Sel (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, called "Ace"), 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 `(SuAx p;q;...)`. For example, to select elements at row 1, columns 0 and 2:

A	(SuAx (Sel 1),(Sel 0 2)) {A	(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	(SuAx 1;2) { A	(SuAx 1 2) { A
abc def	f	f

6.2.7 Shape of the Result

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

```
B =: 10 + 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)
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36	22 23 25 26 31 32 34 35	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:

r =: 1 $ 1	S =: (< p;r;c){B	$ S	($p),($r),($c)
1	22 23 31 32	2 1 2	2 1 2

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

r =: 1	T =: (< p;r;c){B	$ T	($p),($r),($c)	$ r
1	22 23 31 32	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 $ 13 52 51 14
```

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

M	LL =: ,M	Z =: 50 f } LL	($M) $ Z
13 52 51 14	13 52 51 14	13 50 50 14	13 50 50 14

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	50 f M
13 52 51 14	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) $ 50 (50 f M) } (, M)
13 52 51 14	13 50 50 14

Finally, provided `f` delivers linear indices, then `(})` allows the last expression to be abbreviated as:

M	50 f } M
13 52 51 14	13 50 50 14

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`

`+---+----+`
`|J S|Bach|`
`+---+----+`
	`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.

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.

```
L =: 2 3 4
2 3 4
```

```
< L
+-----+
|2 3 4|
+-----+
```

```
M
ab
c
de
f
```

```
< M
+---+
|abc|
|def|
+---+
```

However, we may choose to box each cell separately. There is a conjunction `"` (double-quote, called "Rank"), we write `(< " 0)` to box each scalar, that is, each 0-cell.

```
M
abc
def
```

```
< " 0 M
+-+-+-+
|a|b|c|
+-+-+-+
|d|e|f|
+-+-+-+
```

```
< " 1 M
+---+---+
|abc|def|
+---+---+
```

```
< " 2 M
+---+
|abc|
|def|
+---+
```

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	0 cells M	1 cells M
abc def	+-+-+-+ |a|b|c| +-+-+-+ |d|e|f| +-+-+-+	+---+---+ |abc|def| +---+---+

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,

X =: 2 2 $ 0 1 2 3	Y =: 2 3	X (* " 1 0) Y
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 a separate 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".

The intrinsic ranks of a verb are shown with the aid of a built-in adverb `b.` (lowercase b dot, called "Basic Characteristics"). For any verb `u`, the expression `u b. 0` gives the ranks in the order monadic, left, right.

`*: b. 0`	`#. b. 0`	`< b. 0`
`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) b. 0`	`(<"1 2) b. 0`	`(<"1 2 3) b. 0`
`1 1 1`	`2 1 2`	`1 2 3`

7.3 Frames

Suppose `u` is to be a verb which sums all the numbers in a table, by summing the columns and then summing the column-sums. We specify that `u` is to have monadic rank 2.

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

`w =: 4 5 $ 1`	`u w`	`u b. 0`
`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 $  1
```

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.

`u A`	`$ u A`
`20 20 20` `20 20 20`	`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'
```

`$ A`	`2 frame A`
`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 b. 0
```

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.

k	s	kfr =: k frame A	kfr, s	$ u A
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	AB C DE F	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

```
X =: 2 3 2 $ i. 12
Y =: 2     $ 0 1
```

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

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 2 3 4 5 6 7 8 9 10 11	0 1	0 0 0 0 0 0 1 1 1 1 1 1	0 1 2 3 4 5 7 8 9 10 11 12	0 1 2 3 4 5 7 8 9 10 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 stufffed 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
+-+---+ |1|2 2| | |2 2| +-+---+	1 0 0 0 2 2 2 2	1 0 0 0 2 2 2 2

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
0 1 2 3 4 5 6 7 8 9 10 11	2 3 2	+---+-----+ |0 1| 6 7| |2 3| 8 9| |4 5|10 11| +---+-----+	+---+---+-----+ |0 1|2 3|4 5 | +---+---+-----+ |6 7|8 9|10 11| +---+---+-----+

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 ... (n-1)`, and `>: i. n` gives `1 2 ... 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.

Note that 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. 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

Since `abs` is explicitly defined, we see that its monadic (first) rank is infinite:

```
   abs b. 0
_ _ _
```

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 first element. See Chapter 12 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 say that it applies separately to each scalar in its argument. 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.

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 from Chapter 03 the composition conjunction `@:` (at colon, called "At"). Given verbs `sum` and `square` we can define a composite verb, sum-of-the-squares.

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

`sumsq =: sum @: square`	`sumsq 3 4`
`sum@:square`	`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 =: <
```

`y =: 2 2 $ 1 2 3 4`	`g y`	`f g y`	`(f @: g) y`

`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 =: +/ @: *
```

`x =: 1 2`	`y =: 2 3`	`x * y`	`+/(x * y)`	`x sp y`
`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:

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

x (f @: g) y     =    f (x g y)

x (f &: g) y     =    (g x) f (g y)
```

There is a fourth scheme,

```
(f &: g) y    =    f (g y)
```

which is, evidently, the same as the first. This apparent duplication may be useful if we are interested in writing an ambivalent definition, that is, with both a monadic and a dyadic case.

Notice that from the first and second schemes it follows that if verb `g` is ambivalent then the composition `f @: g` is also ambivalent. For example, suppose `g` is 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`	`+------+`
	`|fedcba|`
	`+------+`
		`+------+`
		`|bcdefa|`
		`+------+`

From the third and fourth schemes above it follows that if verb `f` is ambivalent, then `(f &: g)` is ambivalent. For example, suppose that `f` is the verb `%` (reciprocal or divide). and `g` is `*:` (square).

`% *: 2`	`(% &: *:) 2`	`(*: 3) % (*:2)`	`3 (% &: *:) 2`
`0.25`	`0.25`	`2.25`	`2.25`

8.5 More on Composition: Monad Tracking Monad

There is a conjunction `@` (at, called "Atop"). It is a variation of the `@:` conjunction. Here is an example to show the difference between `(f @: g)` and `(f @ g)`.

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

y	f	g	(f @: g) y	(f @ g) y
0 1 2 3	<	sum"1	+---+ |1 5| +---+	+-+-+ |1|5| +-+-+

We see that with `(f @: 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`.

Recall from Chapter 07 that a verb has in general three ranks, monadic, left and right, and for a verb `f` these ranks are yielded by the expression `f b. 0`. For example

g	g b. 0
sum"1	1 1 1

Suppose that the monadic rank of `g` is `G`.

```
G =: 0 { (g b. 0)
```

Then `(f @ g)` means `(f @: g)` applied separately to each `G`-cell, that is, `(f @: g)"G`.

(f @ g) y	(f @: g)"G y
+-+-+ |1|5| +-+-+	+-+-+ |1|5| +-+-+

and so the general scheme is:

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

x=: 1 2	y=: 2 3 $ 'abcdef'	x g y
1 2	abc def	bca fde

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

f (x g y)	x (f @: g) y	x (f @ g) y
+---+ |bca| |fde| +---+	+---+ |bca| |fde| +---+	+---+---+ |bca|fde| +---+---+

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 b. 0)	x (f @:g)" G y	x (f @ g) y
0 1	+---+---+ |bca|fde| +---+---+	+---+---+ |bca|fde| +---+---+

The scheme is:

```
x (f@g) y =  x (f@:g) " G y
```

8.7 Composition: Dyad Tracking Monad

Recall that in Chapter 03 we met the conjunction `&` as a bonding operator. With one argument a noun and the other argument a dyadic verb the result is a monad. For example `+&6` is a monad

which adds `6` to its argument.

If both arguments of `&` are verbs then `&` has a different interpretation. In this case it is is a composition operator, called "Compose". Now 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 give `1 4 , 9 16`. By contrast, in `(f & g)` there are two separate applications of `f`, giving firstly `1,9` and secondly `4,16`.

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 b. 0)`	`(g x)(f" (G,G))(g y)`	`x(f&g)y`
`0`	`1  9` `4 16`	`1  9` `4 16`

8.8 Ambivalence Again

The composition `f&g` can be ambivalent. The dyadic case, `x f&g y`, we saw above. For the monadic case, `f&g y` means the same as `f@g y`.

```
f =: <
g =: *:
```

`f&g 1 2 3`	`f@g 1 2 3`

`+-+-+-+`
`|1|4|9|`
`+-+-+-+`
	`+-+-+-+`
	`|1|4|9|`
	`+-+-+-+`

8.9 Summary

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

```
@:        (f @: g) y   =  f (g y)
@:     x  (f @: g) y   =  f (x g y)

&:        (f &: g) y   =  f (g y)
&:     x  (f &: g) y   =  (g x) f (g y)

@         (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)
```

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

8.10 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. There is a built-in conjunction `^:` (caret colon, called "Power") such that the expression `f ^: _1` is the inverse of verb `f`. (This is like writing f^{-1} in conventional notation.)

For example, the inverse of square is square root:

sqrt =: *: ^: _1	sqrt 16
*:^:_1	4

^: can automatically find inverses, not only of built-in verbs, but of user-defined verbs such as compositions. For example, the inverse of "twice the square-root of" is "the square of half of"

```
foo    =: (2&*) @: %:
fooINV =: foo ^: _1
```

foo 16	fooINV 8	foo fooINV 36
8	16	36

8.11 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, the square root of a number can be found by taking the logarithm, halving and taking the antilog, that is, halving under logarithm. Recall that halve is -: and logarithm is ^.

SQRT =: -: &. ^.	SQRT 16
-:&.^.	4

The general scheme is that

```
(f &. g) y   means   (g ^: _1) f g y
```

This is the end of Chapter 8.

Chapter 9: Trains of Verbs

In this chapter we continue the topic of trains of verbs begun in Chapter 03. Recall that a train is an isolated sequence of functions, written one after the other, such as `(+ * -)`.

9.1 Review: Monadic Hooks and Forks

Recall from Chapter 03 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:

```
sum  =: +/
mean =: sum % #
```

`y =: 1 2 3 4`	`sum y`	`# y`	`(sum y)%(# y)`	`mean y`
`1 2 3 4`	`10`	`4`	`2.5`	`2.5`

Now we look at some further variations.

9.2 Dyadic Hooks

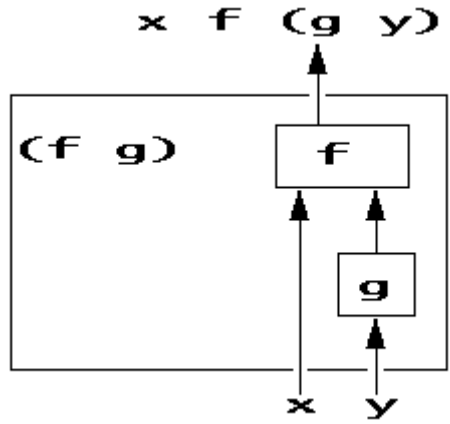
3 hours and 15 minutes is 3.25 hours. A verb `hr`, such that `(3 hr 15)` is `3.25`, can be written as a hook. We want `x hr y` to be `x + (y%60)` and so the hook is:

```
   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

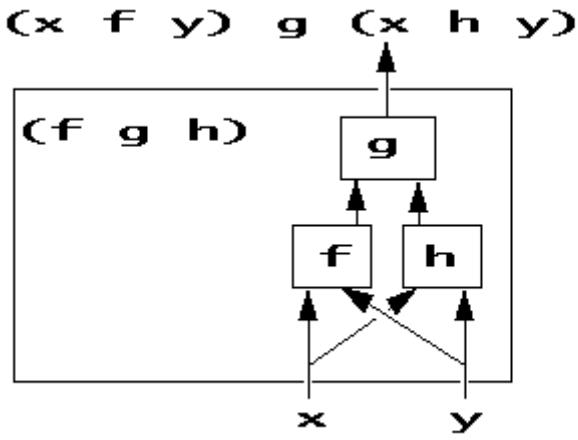
Suppose we say that the expression "10 plus or minus 2" is to mean the list `12 8`. A verb to compute `x` plus-or-minus `y` can be written as the fork `(+,-)`:

(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 schemes for trains of verbs.

```
 (f g h) y    =    (f y) g (h y)       monadic fork

x (f g h) y   =  (x f y) g (x h y)    dyadic  fork

 (f g)   y    =       y  f (g y)       monadic hook

x (f g)   y   =       x  f (g y)       dyadic  hook
```

9.5 Longer Trains

Now we begin to look at ways 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, Suppose that `y` is a list of numbers:

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

Then the "norm" of `y` is defined as `(y - mean y)`, where `mean` is defined above as `(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 `(- sum % #)` does not bring out as clearly as it might that the key idea is subtracting the mean. The formulation `( - 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`	`+-+--+----+`
	`|2|28|1999|`
	`+-+--+----+`

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 built-in verb, monadic `[` (left bracket, called "Same"). It 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`

Monadic `[` and monadic `]` are both called "Same". Dyadic `[` is called "Left". Dyadic `]` is "Right".

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

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

that is,

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

2] 3	(2 + 3) % (2] 3)	2 (+ %]) 3
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.

a =: 3 [b =: 4 [c =: 5	a,b,c
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 `~` adverb with its scheme:

```
(x f~ y) means   y f x
```

Our train can be written as the hook `(g~ 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' & ,
```

`foo =: (f @: [) , (g @: ])`	`'a' foo 'b'`
`f@:[ , g@:]`	`fagb`

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, with `f` and `g` as above, we have

`y=:'y'`	`f g y`	`(f @: g) y`	`([: f g) y`
`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 Constant Functions with the Rank Conjunction

The constant functions `_9:` to `9:` offer more choices for ways of defining trains. Neverthless they are limited to single-digit scalar constants. We look now at 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`:

k	(3 : 'k') 1	(3 : 'k') 1 2
hello	hello	hello

Since the verb `(3 : 'k')` is explicit, its rank is infinite. To apply it separately to scalars then (as we saw in Chapter 07) we need to specify a rank `R` of `0`, with the aid of the Rank conjunction `"` :

k	R =: 0	((3 : 'k') " R) 1 2
hello	0	hello hello

The expression `((3 : 'k') " R)` can be abbreviated as `(k " R)`, because `"` can take, as its left argument, a verb, as above, or a noun:

k	R	((3 : 'k') " R) 1 2	('hello' " R) 1 2
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 for constant functions with `"` is:

```
k " R   means   (3 : 'k') " R
```

9.9.1 A Special Case

Given a temperature in degrees Fahrenheit, the equivalent in Celsius is computed by subtracting `32` and multiplying by five-ninths.

```
   Celsius =: ((5%9) & *) @: (- &32)

   Celsius 32 212
0 100
```

Another way to define `Celsius` is as a fork - a train of three verbs.

```
   Celsius =: (5%9 "_ ) * (-&32)
```

```
   Celsius 32 212
0 100
```

Notice that the fork in `Celsius` above has its left verb as a constant function. Here we have a special case of a fork which can be abbreviated in the form (noun verb verb).

```
   Celsius =: (5%9) * (-&32)

   Celsius 32 212
0 100
```

The general scheme (new in J6) for this abbreviation for a fork is: if `n` is a noun, `u` and `v` are verbs, then

```
n u v  means the fork  (n"_) u v
```

We have come to the end of of Chapter 9.

Chapter 10: Conditional and Other Forms

Tacit verbs, that is, verbs defined without the use of argument variables, were introduced in Chapter 03. Continuing this theme of tacit definition, in Chapter 08 we looked at the use of composition-operators and in Chapter 09 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`.

A sequence of numbers generated by iterating the procedure is called a Collatz sequence, or sometimes a Hailstone sequence. For example:

```
17 52 26 13 40
```

To write a function for this procedure, we start with three verbs, say `halve` to halve, `mult` to multiply-and-add-one, 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:

```
COLLATZ =: 3 : 'if. odd y do. mult y else. halve y end.'
```

and equivalently as a tacit verb:

```
collatz =: halve ` mult @. odd
```

COLLATZ 17	collatz 17
52	52

In the definition of `collatz`, 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). The conjunction `@.` is called "Agenda". Its right argument is a verb, which selects another verb from the list of verbs which is the left argument. Thus in evaluating `collatz 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 ....

10.1.1 Example with 3 Cases

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 balance is unchanged.

Three verbs, one for each of the three cases, could be:

```
   pi =: * & 1.005          NB.  pay interest
   ci =: * & 1.02           NB.  charge interest
   uc =: * & 1              NB.  unchanged
```

pi 1000	ci _100	uc 50
1005	_102	50

Now we want a verb to compute, from a given balance, `0` or `1` or `2`, according to the case. We are free to choose how we number the cases. The following verb scores 1 for a balance of $0 or more plus another 1 for $100 or more.

```
   case =: (>: & 0) + (>: & 100)

   case _50 0 1 100 200
0 1 1 2 2
```

Now the processing of a balance can be represented by the verb `PB` say, being careful to write the three verbs in the correct case-number order.

```
   PB =: ci ` uc  ` pi  @. case
```

PB _50	PB 0	PB 1	PB 100	PB 200
_51	0	1	100.5	201

The balance (the argument of `PB`) is expected to fall under exactly one of the three possible cases. Suppose the argument is a list of balances. The `case` verb delivers not just one but a list of case-numbers. This is an error. The remedy is to apply the `PB` function separately to each item of its argument.

PB 99 100	(PB "0) 99 100
error	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, called "Self-Reference"), 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`

`+: +: +: 1`	`(+: ^: 3 ) 1`
`8`	`8`

The general scheme is that for a verb `f` and an integer `n`

```
(f ^: n) y  means  f f f ... f f f f  y

                   <---  n f's  --->
```

Notice that `f ^: 0 y` is just `y` and then `f ^: 1 y` is `f y`. For example, recall the `collatz` verb "halve or multiply-by-3-and-add-1 if odd".

`(collatz ^: 0) 6`	`(collatz ^: 1) 6`	`collatz 6`
`6`	`3`	`3`

With the Power conjunction we can generate a series by applying `collatz` 0 times, once, twice and so on, starting with 6 for example

```
   (collatz ^: 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 ^: _` 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`

`r =: (P ^: _) 0.5`	`P r`
`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 & |
```

`foo =: halve ^: even`	`(foo " 0) 1 2 3 4`
`halve^:even`	`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)
3
```

The scheme is that if `g` returns `0` or `1` then a function written (`f ^: g ^: _`) can be modelled by an explicit definition:

```
   model =: 3 : 0
while. (g y)
   do. y =.  f y
end.
y
)
```

```
f =: halve
g =: even
```

(f ^: g ^: _) 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 ^: w) y  means   ((x&u) ^: 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:

`k =: 99`	`p =: 3 : 'y+k'`	`q =: 13 : 'y+k'`	`p 6`	`q 6`
`99`	`3 : 'y+k'`	`99 + ]`	`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.

`k =: 0`	`p 6`	`q 6`
`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 =: %:
```

`foo =: 13 : '(f C f y) , g y'`	`f =: *:`	`foo 4`
`f@:f , g`	`*:`	`256 2`

This is the end of Chapter 10

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`:

`f =: * & 3`	`f 4`
`*&3`	`12`

Recall from Chapter 03 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 an 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 Appendix 1 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`

`scale =: * & (5 % 4)`	`scale 8`
`*&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:

`w =: *`	`g =: w & (a % b)`	`g 8`
`*`	`w&1.25`	`10`

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.

h =: ytbd & (a%b)	ytbd =: *	h 8
ytbd&1.25	*	10

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)
```

ctof	ctof 0 100
shift@scale	32 212

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.)`

`ctof`	`ctof f.`
`shift@scale`	`+&32@(*&1.8)`

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@(*&1.8)	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  =  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) 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:

tP =: lambda *] * (1: -])	tP x
lambda *] * 1: -]	0.84

Now we can vary the parameter without redefining the function:

lambda =: 3.75 " 0	tP x
3.75"0	0.9

This is the end of Chapter 11

Chapter 12: Explicit Verbs

This chapter continues from Chapter 04 the theme of the explicit definition of verbs.

12.1 The Explicit Definition Conjunction

Recall from Chapter 04 the example of an explicit dyadic verb, the "positive difference" of two numbers, defined as larger minus smaller.

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

   3 PosDiff 4
1
```

The general scheme for the explicit definition of a function is to provide two arguments to the Explicit Definition conjunction (`:` , colon) in the form

```
type : body
```

In the body, the variables `x` and `y` are the arguments.

12.1.1 Type

The type is a number: type-3 functions are monadic verbs or ambivalent verbs. Type-4 functions are strictly dyadic verbs (that is, with no monadic case). There are other types: types 1 and 2 are operators, covered in Chapter 13 . Type 13 is covered in Chapter 10 .

12.1.2 Memnonics for Types

The standard J profile predefines several variables to provide mnemonic names for the types, and other things, thus:

```
   noun        =: 0
   adverb      =: 1
   conjunction =: 2
   verb        =: 3
   monad       =: 3
   dyad        =: 4
   def         =: :
   define      =: : 0
```

Thus the `PosDiff` example above could be also written as:

```
   PosDiff =: dyad def '(x >. y) - (x <. y)'

   3 PosDiff 4
1
```

12.1.3 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, `PosDiff`, shows a single line written as a string.

A multi-line body can be introduced with a right argument of `0` for the colon operator.

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

```
   3 PosDiff 4
1
```

Another variation allows a multi-line body 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.

```
   PosDiff =: 4 : ('la =. x >. y', LF, 'sm =. x <. y', LF, 'la - sm')
```

PosDiff	3 PosDiff 4

`+-+-+----------+`
`|4|:|la =. x >. y|`
`| | |sm =. x <. y|`
`| | |la - sm |`
`+-+-+----------+`
	1

Another variation uses a boxed list of lines (again with the body parenthesized):

```
   PosDiff =: 4 : ('la =. x >. y' ; 'sm =. x <. y' ;  'la - sm')
```

PosDiff	3 PosDiff 4

`+-+-+----------+`
`|4|:|la =. x >. y|`
`| | |sm =. x <. y|`
`| | |la - sm |`
`+-+-+----------+`
	1

Notice that these are not variations of syntax, but rather alternative expressions for constructing a data-structure acceptable as the right-argument of the `:` operator.

12.1.4 Ambivalent Verbs

An ambivalent verb has both a monadic and a dyadic case. In the definition, the monadic case is presented first, then a line consisting of a solo colon, and then the dyadic case. For example:

```
   log =: 3 : 0
^. y     NB. monad - natural logarithm
:
x ^. y   NB. dyad  - base-x  logarithm
)
```

log 2.7182818	10 log 100
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
+-+-+-------+ |3|:|L =. y| | | |G =: y| | | |L | +-+-+-------+	new	old L	new

With versions of J from J6 onward, it is regarded as an error to make a global assignment (with `=:`) to a variable with the same name as an already-existing local variable.

For example, the argument variables `x` and `y` are local, so it would normally be an error in an explicit verb to make a global assignment to a variable named `y` .

```
   foo =: 3 : 0
z =. y + 1
y =: 'hello'
z
)

   foo 6
|domain error: foo
|   y    =:'hello'
```

If we really, really wanted to assign to a global named `y` from within an explicit definition, the local `y` must first be erased.

```
   foo =: 3 : 0
z =. y+1
erase <'y'
y =: 'hello'
z
)

   foo 6
7
   y
hello
```

12.2.2 Local Functions

We have seen local variables, which are nouns. We may also have local functions. 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
12
```

However, what we can't have is an explicit local function defined by an inner multiline body Recall that a multiline body is a script terminated by a solo right parenthesis, so we cannot have one such body 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 name, of a variable or function, is either global or local. If it is local, then that means it is recognised in the function in which it is defined. However it is not recognised in any inner function. For example:

```
   K =: 'hello '

   zip =: 3 : 0
K =. 'goodbye '
zap =. 3 : 'K , y'
zap y
)

   zip 'George'
hello George
```

We see that there is a global `K` and a local `K`. The inner function `zap` uses the global `K` because the `K` 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*x^2) +(b*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 Control Structures

12.3.1 Review

Recall from Chapter 04 the positive-difference function defined as:

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

   3 POSDIFF 4
1
```

Everything from `if.` to `end.` is called a "control structure". In it,

`if. do. else.` and `end.` are called "control words".

The plan for this section is to use this example for a general discussion of control structures, and then go on to look at a number of particular control structures.

12.3.2 Layout

We can freely choose a layout for the expressions and control words forming a control structure. Immediately before or immediately after any control word, any end-of-line is optional, so that we can choose to remove one or insert one. For example, by removing as many as possible from `POSDIFF` we get

```
   PD =: 4 : 'if.  x > y  do.  x - y  else.  y - x
end. '

   3 PD 4
1
```

12.3.3 Expressions versus Control Structures

We speak of evaluating an expression. We regard assignments as expressions, since they produce values, but in this case it is natural to speak of "executing" the assignment, since there is an effect as well as a value produced. We will use the words "execute" and "evaluate" more or less interchangeably.

Executing (or evaluating) a control structure produces a value, the value of one of the expressions within it. Nevertheless, a control structure is not an expression, and cannot form part of an expression. The following is a syntax error:

```
   foo =: 3 : '1 + if. y > 0 do. y else. 0 end.'

   foo 6
```

```
|syntax error: foo
|       1+
```

Observing the distinction between expressions and control structures, we can say that the body of an explicit definition is a sequence of items, where an item is either an expression or a control structure. Here is an example where the body is an expression followed by a control structure followed by an expression.

```
   PD1 =: 4 : 0
w =. x - y
if. x > y do. z =. w  else. z =. - w end.
z
)

   3 PD1 4
1
```

The value produced by a control structure is discarded if the control structure it is not the last item in the sequence. However, this value can be captured when the item is the last, so that the value becomes the result delivered by the function.

Hence the previous example can be simplified to:

```
   PD2 =: 4 : 0
w =. x - y
if. x > y do. w else. - w end.
)

   3 PD 4
1
```

12.3.4 Blocks

The examples above show 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`.

Expression `T` is regarded as evaluating to "true" if `T` evaluates to any array of which the first element is not `0`.

```
   foo =: 3 : 'if.  y do. ''yes''  else. ''no''
end.'
```

foo 1 1 1	foo 'abc'	foo 0	foo 0 1
yes	yes	no	no

More generally, `T`, `B1` and `B2` may be what are called "blocks". A block is a sequence of items, where an item is either an expression or a control structure. The result delivered by a block is the value of the last item of the block.

Here is an 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       NB. T block
    length  = 0          NB. T block
```

```
do.
    0                    NB. B1 block
else.
    first =. {. y        NB. B2 block
    rest  =. }. y        NB. B2 block
    first + sum rest     NB. B2 block
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 items of a block may be (inner) control structures. For example, here is a 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
```

12.3.5 Variants of if.

A neater version 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
```

showing the pattern:

```
         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.

If all the tests do fail, so that none of the blocks B0 ... Bn is executed, then the result will be `i. 0 0` which is a J convention for a null value.

```
   foo =: 3 : 'if. y = 1 do. 99 elseif. y = 2 do.
77 end. '

   (i. 0 0) -: foo 0
1
```

There is also the pattern:

```
          if. T do. B end.
```

Here either `B` is executed or it is not. For example, positive-difference yet again:

```
   PD =: 4 : 0
z =. x - y
if. y > x do. z =. y - x end.
z
)

   3 PD 4
1
```

12.3.6 The select. Control Structure

Consider this example of a verb to classify a name, using an `if.` control structure.

```
   class =: 3 : 0
t =. 4 !: 0 < y
if.     t = 0 do. 'noun'
elseif. t = 1 do. 'adverb'
elseif. t = 2 do. 'conjunction'
elseif. t = 3 do. 'verb'
elseif. 1     do. 'bad name'
end.
)

   class 'class'
verb
   class 'oops'
bad name
```

A neater formulation is allowed by the `select.` control structure.

```
   CLASS =: 3 : 0
```

```
select.  4 !: 0 < y
case. 0 do. 'noun'
case. 1 do. 'adverb'
case. 2 do. 'conjunction'
case. 3 do. 'verb'
case.   do. 'bad name'
end.
)

   CLASS 'CLASS'
verb
   CLASS 'oops'
bad name
```

Suppose we are interested only in a three-way classification, into nouns, verbs and operators (meaning adverbs or conjunctions). We could of course write:

```
   Class =: 3 : 0
select.  4 !: 0 < y
case. 0 do. 'noun'
case. 1 do. 'operator'
case. 2 do. 'operator'
case. 3 do. 'verb'
case.   do. 'bad name'
end.
)
```

but this can be abbreviated as:

```
   Clss =: 3 : 0
select.  4 !: 0 < y
case. 0    do. 'noun'
```

```
case. 1;2  do. 'operator'
case. 3    do. 'verb'
case.      do. 'bad name'
end.
)
```

Clss 'Clss'	o =: @:	Clss 'o'	Clss 'oops'
verb	+--+ |@:| +--+	operator	bad name

12.3.7 The while. and whilst. Control Structures

In the general pattern

```
while. T do. B end.
```

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.

12.3.8 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.` in which block `B` is executed as many times as there are items of `A`. For example, here is a verb to count the items of a list.

```
   f4 =: 3 : 0
count =. 0
for. y do. count =. count+1 end.
)

   f4 'hello'
5
```

12.3.9 Other Control Structures

Chapter 29 covers the control structure `try. catch. end.` . Other control words and structures are covered in the J Dictionary

This is the end of Chapter 12.

Chapter 13: Explicit Operators

This chapter covers explicit definition of operators, that is, adverbs and conjunctions defined with the colon conjunction.

The scheme for explicit definition 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 so defined 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 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:

`A =: 1 : 'u " 0'`	`f =: < A`	`f 1 2`

`1 : 'u " 0'`	`<"0`	`+-+-+`
		`|1|2|`
		`+-+-+`

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`. In the explicit definition of an adverb, the argument-variable is always `u`.

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:

`foo =: # @: (>&0 # ])`	`y`	`foo y`
`#@:(>&0 # ])`	`1 0 2 3`	`3`

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

```
u C v   is to be   u @: (v # ])
```

and the definition of `C` is straightforwardly:

`C =: 2 : 'u @: (v # ])'`	`f =: # C (>&0)`	`y`	`f y`
`2 : 'u @: (v # ])'`	`#@:(>&0 # ])`	`1 0 2 3`	`3`

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

`f =: # D (>&0)`	`y`	`f y`
`#@:select`	`1 0 2 3`	`3`

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 by their corresponding definitions. That is, "Fix" resolves a tacit verb into its primitives. For example:

p =: +	q =: *	r =: p,q	r f.
+	*	p , q	+ , *

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:

g =: # E (>&0)	y	g y
#@:(>&0 #])	1 0 2 3	3

We see that `g`, unlike `f`, has no local names.

13.2 New Definitions from Old

Suppose we aim to develope a function which, given a list of numbers, replaces each number by the mean of its two neighbours in the list, the previous and the next. For the first or last, we assume a neighbour is zero.

A suitable "data smoothing" function could be written

```
sh    =: |. !. 0    NB. shift, entering zero
prev  =: _1 & sh    NB. right shift
next  =: 1 & sh     NB. left shift
halve =: -:

smoo  =: halve @: (prev + next)
```

so for a list of numbers `N` we might see :

`N =: 6 2 8 2 4`	`prev N`	`next N`	`smoo N`
`6 2 8 2 4`	`0 6 2 8 2`	`2 8 2 4 0`	`1 7 2 6 1`

Now suppose we also want another smoothing function which rotates the data rather than shifting in zero. (The data might be, say, samples of a repeated waveform.)

The only change needed from `smoo` is that the shift verb `sh` must become a rotate verb, that is, (`|.`).

If the definition of `smoo` were large and complicated we might prefer to avoid entering it again. Instead, we could re-evaluate the definition we already have, in an environment in which the name `sh` means "rotate". This environment can be conveniently provided by a little adverb, `SMOO` say, with `|.` (rotate) for its argument:

```
SMOO =: 1 : ('sh =. u' ; 'smoo f.')
```

so the rotating variant of `smoo` is given by

```
   rv =: |. SMOO

   rv
-:@:(_1&|. + 1&|.)
```

N	smoo N	rv N
6 2 8 2 4	1 7 2 6 1	3 7 2 6 4

This example shows using an adverb to generalise an expression (`smoo`) to a function In summary, since `smoo` is defined in terms of `sh`, we have generalised it to a function taking `sh` as argument.

13.3 Operators Generating Explicit Verbs

Suppose we aim to define a conjunction `H` say, with the scheme:

```
  u H v    is to be     3 : 0
                        z =. v y
                        y u z
                        )
```

There is a messy way and a neat way to do this. Let me show you the messy way first, so that the merits of the neat way can be appreciated.

The messy way: we can write `H` in the same style as the previous examples. That is, the body of the definition computes a value which is delivered when the operator is applied to arguments. In this case the value is to be of the form `3 : string` where `string`

must be built from the arguments. For example:

```
   H =:  2 : 0
    U =. 5!:5 < 'u'
    V =. 5!:5 < 'v'
    string =. 'z =. ', V , 'y',  LF
    string =. string , 'y ', U , ' z', LF
    3 : string
)
```

and we see

```
   foo =: + H *:
   foo 5
30
```

The conjunction `H` is pretty ugly but the value of the generated function `foo` is plain to see:

```
   foo
3 : 0
z =. *:y
y + z
)
```

Now we come to the neat way to define this conjunction. So far we have seen operators where the body is executed to deliver the result. Let us say they are operators of the first kind. Now we look at operators of the second kind, where the body of the operator is not executed but instead serves as a template for the verb to be generated. For example:

```
   K =: 2 : 0
z =. v y
y u z
)
```

Clearly the definition of `K` is neater than the definition of `H` but nevertheless they are equivalent. Notice that the body of `K` contains both the argument-variables `u` and `v` for the operator, and also the argument-variable `y` of the generated verb. It is this combination of argument variables which determines that the operator is of the second kind.

```
   bar =: + K *:

   bar 5
30
```

The generated verb `bar` is equivalent to `foo` but it is displayed differently.

```
   bar
+ (2 : 0) *:
z =. v y
y u z
)
```

Now we look in more detail at examples of operators of the second kind.

13.3.1 Adverb Generating Monad

Consider the following explicit monadic verb, `e`. It selects items

greater than 0, by applying the test-verb `>&0`.

`e =: 3 : '(>&0 y) # y'`	`y`	`e y`
`3 : '(>&0 y) # y'`	`1 0 2 3`	`1 2 3`

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.

13.3.2 Adverb Generating Explicit Dyad

Suppose we want an adverb `W`, say, with the scheme: for any verb `u`

```
u W     is to be      4 : '(u x) + (u y)'
```

Recall from Chapter 12 that there is another way to write an explicit dyad. Rather than beginning with `4 :` we can begin with `3 :` and write a multi-line body in which a solo colon separates monadic and dyadic cases. Here we have no monadic case, so the scheme above can be equivalently written as:

```
u W    is to be     3 : 0

:

(u x) + (u y)

)
```

The explicit definition of adverb `W` follows straightforwardly:

```
   W =: 1 : 0
:
(u x) + (u y)
)
```

We see:

`(*: 2) + (*: 16)`	`2 (*: W) 16`
`260`	`260`

For another example, suppose we want an adverb, `T` say, to apply a given verb `u` to every combination of a scalar in vector argument `x` with a scalar in vector argument `y`. There is a built-in adverb `/` called "Table" for this, but here is a home-made version. The scheme is:

```
      u T   is to be 4 : ' x (u " 0 0) " 0 1  y'
```

that is,

```
      u T   is to be 3 : 0
                         x  (u " 0 0) " 0 1  y
                         )
```

and so `T` is defined by

```
   T =: 1 : 0
:
x  ((u " 0 0) " 0 1)  y
)
```

so we see:

```
   1 2 3 + T 4 5 6 7
5 6 7  8
6 7 8  9
7 8 9 10
```

13.3.3 Conjunction Generating Explicit Monad

A conjunction takes two arguments, called `u` and `v`.

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

13.3.4 Generating a Explicit Dyad

Suppose we want a conjunction `H` such that, schematically, for verbs `u` and `v`

```
u H v    is to be    4 : '(u x) + (v y)'
```

or equivalently, as we saw above:

```
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`

13.3.5 Alternative Names for Argument-Variables

For the sake of completeness, it should be mentioned that arguments to operators may be named `m` and `n` rather than `u` and `v`, to constrain arguments to be nouns, that is, to cause verbs to be signalled as errors.

Furthermore, for historical reasons, if the only argument variables are `x` or `y` or both, we get an operator of the first kind. That is, in the absence of `u` or `v` or `m` or `n` then `x` and `y` are equivalent to `u` and `v`.

These alternative names will not be further considered.

13.3.6 Review

So far, we have seen that for operators introduced with `1 : body` or `2 : body`, there are two kinds of definition.

- With operators of the first kind, the body is executed (that, is evaluated) to compute the value of the result. The result

can be of any type. The argument-variables occurring in the body are `u` or `v` or both.

- With operators of the second kind, the result is always an explicit function. The body of the operator is not executed, but rather becomes the body of the generated function. Here `x` and `y` are arguments to the generated function in the usual way, and `u` or `v` in this body are placeholders which receive the values of arguments to the operator.

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 BOTH (`u` or `v`) AND (`x` or `y`) then the operator is of the second kind. Otherwise it is of the first kind.

13.3.7 Executing the Body (Or Not)

We said above that, for an operator of the first kind, the body is executed (or evaluated) when arguments are supplied. This can be demonstrated.

First, here is a utility verb which displays its argument on-screen.

```
   display =: (1 !: 2) & 2
```

Now insert `display 'hello'` into an operator of the first kind:

```
     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, for an operator of the second kind, when arguments are supplied, the body is not executed, but rather becomes the body of the result function (after substituting the arguments). We can demonstrate this by inserting `display 'hello'` into the body of the operator, and observing that it becomes part of the result-function.

```
   S =: 2 : 0
display 'hello'
selected =. (v y) # y
u selected
)
```

we see that the body of `S` is NOT executed when `S` 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
```

13.4 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. We can iterate the function until there is no change (with `^: _` - see **Chapter 10**), choosing 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.

`FPF =: 1 : '(u ^: _ ) 0.5'`	`p =: f FPF`	`f p`
`1 : '(u ^: _ ) 0.5'`	`0.642857`	`0.642857`

13.5 Generating Noun or Verb

Consider two lines of J, such as

```
sum  =: +/

mean =: sum  % #
```

Sometimes a smoother presentation might be:

```
mean =: sum % #  where sum =: +/
```

provided we had available a suitable definition for **where**. How about this?

```
   where =: 2 : 'u'
```

so we can say:

```
   mean =: sum % # where sum =: +/
```

with results as expected:

mean	mean 1 2 3 4
sum % #	2.5

The right argument of **where** can be a verb or noun:

```
     (z+1) * (z-1)      where z =: 7
48
```

where is a conjunction which ignores its right argument, but evaluating the right argument makes it available to the left through the assignment. Note that the assignments to **sum** and **z** above are regular global assignments, so **where** does not localize **sum** or **z**.

13.6 Operators Generating Operators

Here is an example of an adverb generating an adverb.

First note that (as covered in Chapter 15) if we supply one

argument to a conjunction we get an adverb. The expression `(@: *:)` is an adverb which means "composed with square". To illustrate:

`CS =: @: *:`	`- CS`	`- CS 2 3`	`- *: 2 3`
`@:*:`	`-@:*:`	`_4 _9`	`_4 _9`

Now back to the main example of this section. We aim to define an explicit adverb, `K` say, which generates an adverb according to the scheme: for a verb `u`

```
u K    is to be    @: u
```

Adverb `K` can be defined as below. We see that adverb `K` delivers as a result adverb `L`:

`K =: 1 : '@: u'`	`L =: *: K`	`- L`	`- L 2 3`
`1 : '@: u'`	`@:*:`	`-@:*:`	`_4 _9`

This is the end of Chapter 13.

Chapter 14: Gerunds

What is a gerund, and what is it good for? Briefly, a gerund represents a list of verbs. It is useful mainly 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 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:

`abs =: + ` ` - @. (< & 0)`	`abs _3`
`+`-@.(<&0)`	`3`

The expression `(+`-)` 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
+-----+---+----+ |count|sum|mean| +-----+---+----+	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
+-----+-----+          2 9
|count|sum  |        4.5 2
+-----+-----+
|mean |count|
+-----+-----+
```

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. This noun can be a single number or a list of numbers. A single number is taken as an index selecting a verb from the gerund. For example.

```
   G =: + ` - ` %    f =: G @. 0    1 f 1
+-+-+-+              +              2
|+|-|%|
+-+-+-+
```

A list of numbers is taken as a list of indices selecting verbs from the gerund to form a train. In the following example the selected two verbs form a hook.

G	h =: G @. 0 2	h 4
+-+-+-+ |+|-|%| +-+-+-+	+ %	4.25

The scheme is, for a gerund `G` and indices `I` :

```
G @. I   means   (I { G) EV
```

For example:

G	(G @. 0 2) 4	((0 2 { G)) EV 4
+-+-+-+ |+|-|%| +-+-+-+	4.25	4.25

This scheme gives us an abbreviation for the unpacking by indexing we saw above. 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
* (- 1:)	* (- 1:)	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 and `x` is the list `(x0, x1, x2, x3, ...)` then

```
G/X    means   x0 f x1 g x2 h x3 ...
```

ger =: + ` %	ger / 1 2 3	1 + 2 % 3
+-+-+ |+|%| +-+-+	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 =: <:
```

`double ^: (decr 3) 3`	`(double ^: decr) 3`
`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

`u 0 1`	`u u 0 1`	`u u u 0 1`
`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 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 ^: (v1 `v2)`	`FIB 6`
`(, sumlast2)^:(v1`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 ^: k  y    means    (x&u) ^: 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 ^: (U`V`W) y  means  (((x U y)&u) ^: (x V y))  (x W y)
```

or equivalently as a fork:

```
u ^: (U`V`W)   means   U (u ^: V) W
```

For example, suppose we define::

```
U =: [
V =: 2:
W =: ]
```

Then we see that `p` and `q` below are equivalent. 3 added twice to 4 gives 10.

p =: + ^: (U`V`W)	3 p 4	q =: U (+ ^: V) W	3 q 4
+^:(U`V`W)	10	U +^:V W	10

14.3.5 Gerund as Argument to Amend

Recall the "Amend" adverb from Chapter 06 . 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`V`W`. The scheme is:

```
          x (U`V`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. x i y  is  first of x
   k =: {: @ [    NB. x k y  is  last of x
   r =: i { ]     NB. x r y  is  (x i y)'th  row of y

   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.

The main consideration with an operator is how to recover individual verbs from the gerund argument. Useful here is the agenda conjunction `@.` which we looked at above. Recall that it can select one or more verbs from a gerund.

G	G @. 0	G @. 0 2
+-+-+-+ |+|-|%| +-+-+-+	+	+ %

Now for the operator. Let us define an adverb `A`, say, to produce a fork-like verb, so that

```
  x (f `g ` h  A)  y   is to mean   (f x) g (h y)


   A =: 1 : 0
f =. u @. 0
g =. u @. 1
h =. u @. 2
((f @ []) g (h @ ]))  f.
)
```

To demonstrate `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").

`H =: {. ` , ` {:`	`zip =: H A`	`'abc' zip 'xyz'`

`+--+-+--+`
`|{.|,|{:|`
`+--+-+--+`
	`{.@[ ,`
	`{:@]`
		`az`

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 ("Structure and Interpretation of Computer Programs", MIT Press 1985) 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 =. u @. 0
map =. u @. 1
fil =. u @. 2
((com /)  @:  map  @:  (#~ fil))  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
```

This is the end of chapter 14.

Chapter 15: Tacit Operators

15.1 Introduction

J provides a number of built-in operators - adverbs and conjunctions. In Chapter 13 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 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 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 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| +-+-+-+	+--+--+--+ |fa|fb|fc| +--+--+--+

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, called Prefix). 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 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"

<: A	<: A B	h =: <: H	h 5
+-+--+ |*|<:| +-+--+	* <:	* <:	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.

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	ecdb a	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:

c. 4 2 3 1 0	c. 3 1 2 ; 4 0
+-----+---+ |3 1 2|4 0| +-----+---+	4 2 3 1 0

The dyadic verb `c.` can accept either form as its left argument, and permutes its right argument.

y	4 2 3 1 0 C. y	(3 1 2 ; 4 0) C. y
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.

y	2 C. y	2 3 C. y
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)`.

`A. 2 1 0`	`5 { tap 3`
`5`	`2 1 0`

The dyadic verb `A.` applies an atomic representation of a permutation.

`2 1 0 { 'PQR'`	`5 A. 'PQR'`
`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 permutions 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.

L	/: L	(/: L) { L
barn	1 0 3 2	abnr

For sorting into descending order, the monad `\:` (backslash colon, called "Grade Down") can be used.

L	(\: L) { L
barn	rnba

Since `L` is a character list, its items are sorted into alphabetical order. Numeric lists or boxed lists are sorted appropriately.

N =: 3 1 4 5	(/: N) { N
3 1 4 5	1 3 4 5

B =: 'pooh';'bah';10;5	(/: B) { B
+----+---+--+-+ |pooh|bah|10|5| +----+---+--+-+	+-+--+---+----+ |5|10|bah|pooh| +-+--+---+----+

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`, 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 expression `(/: keys { T)` can be abbreviated as `(T /: keys)`, using the dyadic case of `/:`, (called "Sort")

```
(/: keys) { T
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
```

```
T /: keys
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
```

The dyadic case of `\:` is similar: it is also called "Sort".

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
+------+---+
|Mozart|WA |
+------+---+
|Bach  |JS |
+------+---+
|Bach  |CPE|
+------+---+
```

```
T /: keys
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|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`

`+-----+---+-++-+`
`|+---+|pqr|4||3|`
`||abc||   | || |`
`|+---+|   | || |`
`+-----+---+-++-+`
	`+-+-++---+-----+`
	`|3|4||pqr|+---+|`
	`| | ||   ||abc||`
	`| | ||   |+---+|`
	`+-+-++---+-----+`

Within arrays of the same type, low-rank precedes high-rank.

```
m=: 2 4 ; 3 ; (1 1 $ 1)
+---+-+-+
|2 4|3|1|
+---+-+-+
```

```
m /: m
+-+---+-+
|3|2 4|1|
+-+---+-+
```

Within arrays of the same type and rank, in effect the arrays are ravelled, and then compared element by element. In this case, `1 2` takes precedence over `1 3` (because `2 < 3`), and `3 3` takes precedence over `3 3 3` (because `3 3` is shorter than `3 3 3`). 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
```

```
u=:a;b;c
+-----+---+-----+
|1 2 3|1 2|1 2 3|
|4 5 6|5 6|     |
|     |3 4|     |
+-----+---+-----+
```

```
u /: u
+---+-----+-----+
|1 2|1 2 3|1 2 3|
|5 6|     |4 5 6|
|3 4|     |     |
+---+-----+-----+
```

```
w=:a;b;c;d
```

```
+-----+---+-----+-----+
|1 2 3|1 2|1 2 3|1 1 3|
|4 5 6|5 6|      |     |
|     |3 4|      |     |
+-----+---+-----+-----+
```

```
w /: w
```

```
+-----+---+-----+-----+
|1 1 3|1 2|1 2 3|1 2 3|
|     |5 6|     |4 5 6|
|     |3 4|     |     |
+-----+---+-----+-----+
```

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, `|`.

```
x=: 2 1 _3
```

```
2 1 _3
```

```
keys =: | x
```

```
2 1 3
```

```
x /: keys
```

```
1 2 _3
```

16.3 Transpositions

The monadic verb `|:` will transpose a matrix, that is, interchange the first and second axes.

`M =: 2 3 $ 'abcdef'`	`|: M`

`abc`
`def`
	`ad`
	`be`
	`cf`

More generally, `|:` will reverse the order of the axes of a n-dimensional array.

`N =: 2 2 2 $ 'abcdefgh'`	`|: N`

`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, there are 6 possible permutations, with the first being the identity-permutation

`N`	`0 1 2 |: N`	`0 2 1 |: N`	`1 0 2 |: N`

`ab`
`cd`
 	
`ef`
`gh`
	`ab`
	`cd`
	
	`ef`
	`gh`
		`ac`
		`bd`
		
		`eg`
		`fh`
			`ab`
			`ef`
			
			`cd`
			`gh`

1 2 0 |: N	2 0 1 |: N	2 1 0 |: N
ae bf cg dh	ac eg bd fh	ae cg 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	edcb a	ab c de f	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	ef gh ab cd	ba dc fe hg	cd ab 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 |. y	_1 |. y
abcde	bcdea	eabcd

Successive numbers in `x` rotate y along successive axes:

M	1 2 |. M	N	1 2 |. 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

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 \ 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'`	`< \ y`

`abcde`	`+-+--+---+----+-----+`
	`|a|ab|abc|abcd|abcde|`
	`+-+--+---+----+-----+`

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 \ y)` the verb `f` is applied to successive sections ("infixes") of `y`, each of length `x`.

`z =: 1 4 9 16`	`2 < \ z`

`1 4 9 16`	`+---+---+----+`
	`|1 4|4 9|9 16|`
	`+---+---+----+`

If `x` is negative, then the sections are non-overlapping, in which case the last section may not be full-length. For example:

`z`	`_3 < \ z`

`1 4 9 16`	`+-----+--+`
	`|1 4 9|16|`
	`+-----+--+`

We can compute the differences between succesive items, by choosing 2 for the section-length, and applying to each section a verb "second-minus-first", that is, `({: - {.)`

`smf =: {: - {.`	`smf 1 4`
`{: - {.`	`3`

```
diff =: 2 & (smf\)
```

,. z	,. diff z	,. diff diff z
1 4 9 16	3 5 7	2 2

17.1.3 Suffix Scanning

In the expression `(f \. y )` the result is produced by applying `f` to successively shorter trailing sections ("suffixes") of y .

y	< \. y
abcde	+-----+----+---+--+-+ |abcde|bcde|cde|de|e| +-----+----+---+--+-+

17.1.4 Outfix

In the expression `(x f \. 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 < \. y	_2 < \. y
abcde	+---+---+---+---+ |cde|ade|abe|abc| +---+---+---+---+	+---+---+----+ |cde|abe|abcd| +---+---+----+

17.2 Cutting

The conjunction `;.` (semicolon dot) is called "Cut". If `u` is a verb and `n` a small integer, then `(u ;. 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 (`[`).

M =: 3 3 $ 'abcdefghi'	[;. 0 M
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`	`+----+---+--+----+`
	`|what|can|be|said|`
	`+----+---+--+----+`

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:

t =: 'How are you?'	b =: t e. frets
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 `b` can be used to specify the frets for cutting text `t` into words:

t	b	b < ;. _2 t
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	+-+---+-----+ |3|1 4|1 5 9| +-+---+-----+

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`	`+-+--+-+-+-+-+-+-+-+`
	`|z|=:|(|p|+|q|)|-|1|`
	`+-+--+-+-+-+-+-+-+-+`

17.2.6 Lines in Files

Let us begin by creating a file, to serve in the examples which follow. (See Chapter 26 for details of file-handling functions).

```
   text =: 0 : 0
What can be said
at all
can be said
clearly.
)

   text (1 !: 2) < '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) < '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 in input data. 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.

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 expresssion `x e. y` tests whether `x` matches any item of `y`, that is, whether x is a member of the list `y`. For example:

y=: 'abcde'	'a' e. y	'w' e. y	'ef' e. y
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
~.	hook	hok

We can apply `nub` to the rows of a table:

t	nub t
ab cd ef ab	ab cd ef

18.1.4 Nub Sieve

The verb "nub sieve" `(~:)` gives a boolean vector which is true only at the nub.

y	b =: ~: y	b # y	nub y
hook	1 1 0 1	hok	hok

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.~)
```

1 2 3 seteq 3 1 2 1	1 2 3 seteq 1 2
1	0

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:

x =: 0 1 2	y =: 3 4 5 6	z =: x +/ y
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 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:

`x =: 'abc'`	`y =: 'face'`	`x =/ y`

`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 `((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	+-+-+ |h|1| +-+-+ |o|2| +-+-+ |k|1| +-+-+

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))
```

classify 6	classify 4.8 5.1 6 7 7.1 8
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.

data =: 7 5 6 4 8	k =: classify data
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 =: 'LETTUCE'	= y	(nub y) ; " 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 u /. y`.

The applied verb `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.

```
u =: {. ; #
```

y	= y	y 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 `at` from table `AT` and a row `ts` from table `TS`. Each pair `at,ts` 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 `at,ts` 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_at. x do.
   for_ts.  y do.
     if. u at,ts do. z =. z, v at,ts  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_at. x do.
      z =. z , |: v (#~"1  u) |: at , "1 y
end.
~. z
)

   vjoin  =: test VPAIR extr
```

giving the same result as before:

```
AT join TS                AT vjoin TS
+-----+----+              +-----+----+
|Smith|pigs|              |Smith|pigs|
+-----+----+              +-----+----+
|Brown|cats|              |Brown|cats|
+-----+----+              +-----+----+
|Brown|dogs|              |Brown|dogs|
+-----+----+              +-----+----+
|Smith|dogs|              |Smith|dogs|
+-----+----+              +-----+----+
|James|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 `Brown `Pets `Smith `Dogs `James `Dogs	`Pigs `pigs `Pets `cats `Pets `dogs `Dogs `dogs	`Smith `pigs `Brown `cats `Brown `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 `Smith `Dogs `James `Dogs	`Brown `Pets `James `Dogs `Smith `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| 1.78752| 0.04552|
+------+--------+--------+
|vector| 0.02507| 0.00199|
+------+--------+--------+
```

In Chapter 31 we will return to the topic of performance in computing join of relations.

18.4.4 Saving and Restoring the Symbol Table

Suppose that `data` is an array of symbols.

```
   ] data =: s: 2 2 $ 'hello';
'blah';'blah';'goodbye'
`hello `blah
`blah  `goodbye
```

For a symbol in `data` its original string (`'hello'` for example) is stored only in the symbol table, not in `data` itself. The original string is needed to display the value of the symbol.

Suppose that we write `data` to a file, aiming to read it back in a new session. At the beginning of a new session, the symbol table is empty. Thus we must save the symbol table from the earlier session, and reinstate it at the beginning of the new session.

First, here are two utility functions to save a value to a file and retrieve it. (See Chapter 27 and Chapter 28 for more about data in files.)

```
   save =: 4 : '(3!:1 x ) 1!:2 < y '
   retr =: 3 : '3!:2 (1!:1 < y )'
```

Save the data to a file named, say, `data.xyz`

```
   data save 'data.xyz'
```

The symbol table is not itself a variable, but the expression `0 s: 10` gives a value for it. We save this value to a file named, say, `symtab.xyz`

```
   (0 s: 10) save 'symtab.xyz'
```

Start a new J session. The symbol table is initially empty, so begin by reinstating it from the file saved in the earlier session:

```
   10 s: (retr 'symtab.xyz')
1
```

Now, with the correct symbol table in place, we can retrieve the array of symbols `data` from its file:

```
   DATA =: retr 'data.xyz'
```

and we see that the symbols are correctly interpreted:

```
   DATA
`hello `blah
`blah  `goodbye
```

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 and formatted
- Number bases
- Random numbers

19.1 Numbers of Six Different Kinds

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`

```
implies =: not @ (and not)
```

p	q	p implies q
0	0	1
0	1	1
1	0	0
1	1	1

Notice that in the truth-table above the rows are given in an order such that p,q = successively 00 01 10 11 in binary or 0 1 2 3. Call this the standard order.

With the rows of a truth-table in standard order, the result-column can be read as a 4-bit number, `1 1 0 1` in this example. This

means that there are altogether only 16 possible logical functions of two arguments, and that any of them can be specified by giving its 4-bit result.

There is a built-in adverb `b.` (lowercase b dot, called "Boolean"), which can take an integer in the range 0-15, expressing a 4-bit result, and produces the corresponding logical function.

For example, we saw above that for logical implication its 4-bit specification is `1 1 0 1` or `13`, giving us another way to define implication as `13 b.` We see:

p	q	p implies q	p (13 b.) q
0	0	1	1
0	1	1	1
1	0	0	0
1	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 implies 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.

maxint=:2147483647	type maxint	z =: 1+maxint	type z
2147483647	int	2.14748e9	float

19.1.3 Bitwise Logical Functions on Integers

J provides all the expected functions on integers, so not much need be said here. However, this might be a good place to mention that bitwise logical functions on integers are available through the built-in adverb `b.` we met above. To begin, here is a utility function to show the last 8 bits of an integer.

```
bits =: ('... ' & ,) @: ({&'01')@:((8#2) & #:)
```

bits 0	bits 1	bits 5
... 00000000	... 00000001	... 00000101

Recall that logical function `k` is given by `k b.` where `k` is in the range `0-15`. The function `(k+16) b.` is logically the same, but applies, not to booleans, but to integers bitwise, that is, on machine words.

For example `(1 b.)` is logical-and on booleans, while `(17 b.)` is logical-and on integers.

```
   BLAND =: 17 b.    NB. bit-wise logical and
```

bits 45	bits 7	bits 45 BLAND 7
... 00101101	... 00000111	... 00000101

The verb `(32 b.)` rotates the bits of `y` leftward by `x` places, or rightward for negative x. Similarly `(33 b.)` shifts and `(34 b.)` performs a "signed shift" that is, propagating the sign bit on a rightward move. For example:

```
   ] a =: 1
1

   ] b =: _1 (32 b.) a   NB. rotating rightwards
_2147483648

   ] c =: _1 (34 b.) b   NB. shifting right,
propagating sign-bit
_1073741824

   ] d =: 2  (34 b.) c   NB. shifting left,
removing sign-bits
0
```

For one more example, recall the Collatz function from **Chapter 10** : halve if even, otherwise triple and add one. Here is a bitwise version.

```
   odd    =: (17 b.) & 1
   halve  =: _1 & (33 b.)      NB. OK for an even number !
   triple =: + (1 & (33 b.))

   collatz =:   halve ` (1 + triple) @. odd

   collatz ^: (i. 10) 5
5 16 8 4 2 1 4 2 1 4
```

19.1.4 Floating-Point Numbers

A floating-point number is a number represented in the computer in such a way that: (1) there may be a 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".

19.1.5 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 `XeY`, where `Y` is a (positive or negative) integer means `(X * 10^Y)`.

3e2	1.5e6	1.5e_4
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.

19.1.6 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".

a =: 1.001	b =: a - 2^_45	a - b	a = b
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 larger of the

magnitudes:

```
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`

a RD b	2^_44	a = b
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.

tiny =: 1e_300	tiny = 0	tiny RD 0
1e_300	0	1

However, `1+tiny` is tolerantly equal to `1`.

tiny	tiny = 0	1 = tiny + 1
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`.

9!:18 ''	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`.

(9!:19) 2^_44	a = b	(9!:19) 0	a = b
	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 `t`, regardless of the global parameter, can be written as `= !. t`. For example, strict (zero-tolerance) equality can be defined by:

```
streq =: = !.  0
```

Resetting the global tolerance to the default value, we see:

(9!:19) 2^_44	a - b	a = b	a streq b
	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`

a - b	a > b
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:

(9!:19) 2^_35	(9!:19) 2^_34
	error

The effect of disallowing large tolerances is that no two different integers compare equal when converted to floating-point.

19.1.7 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 `3j4`.

In J an imaginary number is represented as a complex number with real part zero. Thus "i", the square root of -1, can be written `0j1`.

i =: %: _1	i * i	0j1 * 0j1
0j1	_1	_1

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").

`3 + (%: _1) * 4`	`3 j. 4`
`3j4`	`3j4`

Some more examples of arithmetic with complex numbers:

`2j3 * 5j7`	`10j21 % 5j7`	`2j3 % 2`
`_11j29`	`2.66216j0.47297 3`	`1j1.5`

A complex number such as `3j4` 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").

`+. 3j4`	`*. 3j4`
`3 4`	`5 0.927295`

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
```

mag * (cos ang) + 0j1 * sin ang	mag r. ang
3j4	3j4

A complex constant with magnitude `X` and angle (in radians) `Y` can be written in the form `XarY`, meaning `X r. Y`. Similarly, if the angle is given in degrees, we can write `XadY`.

5ar0.9272952	5ad53.1301
3j4	3j4

19.1.8 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:

a =: *: 10000000001	b =: *: 10000000001x
1e20	100000000020000000001

Here `a` is an approximation while `b` is an exact result.

type a	type b
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

19.1.9 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 are exact representations using extended integers. Arithmetic with rationals gives exact results.

2r3 + 1r7	2r3 * 4r7	2r3 % 5r7
17r21	8r21	14r15

Rationals can be constructed by dividing extended integers. Compare the following:

```
2 % 3
0.666667
```

```
2x % 3x
2r3
```

A rational can be constructed from a given floating-point number with the verb `x:`

```
x: 0.3
3r10
```

```
x: 1 % 3
1r3
```

A rational number can be converted to a floating-point approximation with the inverse of `x:` , that is, verb `x: ^: _1`

```
float =: x: ^: _1
+--+--+--+
|x:|^:|_1|
+--+--+--+
```

```
float 2r3
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.

```
nd =: 2 & x:
+-+-+--+
|2|&|x:|
+-+-+--+
```

```
nd 2r3
2 3
```

19.1.10 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.

`1 + 10^19x`	`1r3 * 0.75`
`10000000000000000001`	`0.25`

The general scheme is that the six types form a progression: from boolean to integer to extended to rational to floating-point to complex. We say that boolean is the simplest or "lowest" type and complex as the most general or "highest" type

Where we have two numbers of different types, the one of lower type is converted to match the type of the higher. and the result is of the "higher".

`type 1r3`	`type 1%3`	`z =: 1r3, 1%3`	`type z`
`rational`	`float`	`0.333333 0.333333`	`float`

19.2 Special Numbers

19.2.1 "Infinity"

A floating-point number can (on a PC) be no larger than about

1e308, because of the way it is stored in the computer's memory. Any arithmetic which attempts to produce a larger result will in fact produce a special number called "infinity" and written _ (underscore). For example:

1e308 * 0 1 2	1e400	1 % 0
0 1e308 _	_	_

There is also "negative infinity" written as __ (underscore underscore). Infinity is a floating-point number:

```
   type _
float
```

19.2.2 "Indeterminate" Numbers

A floating-point number is a 64-bit value, but not all 64-bit values are valid as floating-point numbers. Any which is not valid is said to be "Not a Number", or a "NaN". Such a value might occur, for example, in data imported into a J program from an unreliable external source.

When displaying the values of numbers, the J system reports any supposed floating-point number, which is in fact "Not a Number", as the symbol _. (underbar dot, called "Indeterminate").

Floating-point arithmetic on _. arguments cannot be relied upon to produce meaningful results. Thus _. is best regarded solely as a

mark of error.

The sole reliable test for `_.` is the verb `128 !: 5` . In the following example note the difference between results of the unreliable test `x = _.` and the reliable test `128 !: 5 x` .

`x =: 1.5 _. 2.5`	`x = _.`	`128!:5 x`
`1.5 _. 2.5`	`0 0 0`	`0 1 0`

19.3 Number Bases

The number `5` is represented in binary as `1 0 1`. There is a built-in function, monadic `#:` (hash colon, called "Antibase Two") to compute the binary digits. Note that the result is a list.

```
   #: 5
1 0 1
```

We say that the binary digits are the base-2 representation. More generally, a base-n representation can be produced. The left argument of dyadic `#:` (called "Antibase") specifies the both the base and the number of digits. To get four binary digits we can write:

```
   2 2 2 2 #: 5
0 1 0 1
```

63 as two base-8 (octal) digits:

```
   8 8 #: 63
7 7
```

A mixed-base representation is possible. How many hours, minutes and seconds are there in 7265 seconds?

```
   24 60 60 #: 7265
2 1 5
```

The inverse functions produce numbers from lists of digits in specified bases. Monadic `#.` is called "Base Two". Binary `1 0 1` is `5`

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

Dyadic `#.` is called "Base". Its left argument specifies a number-base for the digits of the right argument.

```
   2 #. 1 0 1
5
```

Equivalently:

```
   2 2 2 #. 1 0 1
5
```

There must be a base specified on the left for each digit on the right, otherwise an error is signalled

```
    2 2 #. 1 0 1
|length error
|   2 2    #.1 0 1
|[-531] c:\users\homer\13\js\19.ijs
```

Again, mixed bases are possible: 2 hours 1 minute 5 seconds is 7265 seconds

```
   24 60 60  #. 2 1 5
7265
```

19.4 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 `XpY` means `X * pi ^ Y` where `pi` is the familiar value 3.14159265....

pi =: 1p1	twopi =: 2p1	2p_1
3.14159	6.28319	0.63662

Similarly, a numeral written with letter `x`, of the form `XxY` 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 `NbDDD.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
```

19.4.1 Combining the Notations

The notation-letters `e`, `r`, `j ar ad 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 `XrY` denotes the number `X%Y`. A numeral in the form `XeYrZ` denotes the number `(XeY) % Z` because `e` is considered before `r`.

1.2e2	(1.2e2) % 4	1.2e2r4
120	30	30

A numeral in the form `XjY` denotes the complex number `(X j. Y)` (that is, `(X + (%: _1) * Y)`. A numeral in the form `XrYjZ` denotes the number `(XrY) j. Z` because `r` is considered before `j`

3r4	(3r4) j. 5	3r4j5
3r4	0.75j5	0.75j5

A numeral in the form `XpY` denotes the number `X*pi^Y`. A numeral in the form `XjYpZ` denotes `(XjY) *pi^Z` because `j` is considered before `p`.

3j4p5	(3j4) * pi ^ 5
918.059j1224.08	918.059j1224.08

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`.

(3*pi)+5	1p1b35
14.4248	14.4248

19.5 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.

`10 ^ 3`	`2.7182818285`	`2.718281828 * 10 ^ 7`
`1000`	`2.71828`	`2.71828e7`

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`

19.5.1 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

Note that monadic format shows a complex number in the usual way, but dyadic format shows only the real part of a complex number.

`z =: 3.14j2.72`	`": z`	`6j2 ": z`
`3.14j2.72`	`3.14j2.72`	`3.14`

More formatting options are provided by the built-in verbs `8!:n`. Here is a small example to show a few of the many options described in the J dictionary.

Suppose our table of numbers to be formatted is `N`

```
    ] N =: 3 2 $  3.8  _2000 0  123.45 _3.14 15000
  3.8  _2000
    0 123.45
_3.14  15000
```

We can format each column of `N` separately. Suppose numbers in the first column are to be presented as blank when zero, 6 characters wide with 0 decimal places. We write a "formatting phrase" like this

```
   fp1 =: 'b6.0'
```

Here the `'b'` means blank when zero.

Suppose for the second column we require a comma between each 3 digits. We require negative numbers to be shown with a following "CR" and therefore non-negative numbers should be followed by two blank characters, so that decimal points line up vertically. We require a 12-character width with 2 decimal places.

A suitable formatting phrase is like this:

```
   fp2 =: 'cn{CR}q{  }12.2'
```

Here the `'c'` means commas, `n{CR}` means CR after a negative number and `q{ }` means 2 spaces after a non-negative.

Applying the formatting verb `8!:2` we see

N	(fp1;fp2) (8!:2) N
3.8 _2000 0 123.45 _3.14 15000	4 2,000.00CR 123.45 -3 15,000.00

The formatted result is a character table of dimensions 3 by 18, because `N` has 3 rows, and we specified widths of 6 and 12 for first and second columns.

```
   $ (fp1;fp2) (8!:2) N
3 18
```

19.6 Random Numbers

19.6.1 Roll

There are built-in functions for generating random numbers. Monadic `?` is called "Roll", because `? n` gives the result of rolling a die with `n` faces marked `0` to `n-1`.

```
   ? 6
4
```

That is, `? n` is selected from the items of `i. n` randomly with equal probability.

A list of random numbers is generated by repeating the n-value. For example, four rolls of a six-sided die is given by

```
   ? 6 6 6 6
3 4 5 2
```

or more conveniently by:

```
   ? 4 $ 6
2 3 5 3
```

19.6.2 Uniform Distribution

With an argument of zero, monadic `?` generates random reals uniformly distributed, greater than `0` and less than `1`.

```
   ? 0 0 0 0
0.14112615 0.083891464 0.41388488 0.055053198
```

19.6.3 Other Distributions

The built-in verb `?` generates equiprobable numbers. Various other distributions are provided by the J Application Library stats/distribs Addon

19.6.4 Deal

Dyadic `?` is called "Deal". `x ? y` is a list of `x` integers randomly chosen from `i. y` without replacement, that is, the `x` integers are all different.

Suppose that cards in a deck are numbered `0` to `51`, then `13 ? 52` will deal a single hand of 13 randomly selected cards, all different.

```
   13 ? 52
44 2 36 30 0 6 43 26 28 1 34 48 41
```

A shuffle of the whole deck is given by `52 ? 52`. To shuffle and then deal four hands:

```
   4 13 $ 52 ? 52
15 18  3  8 11 25 27 51 20 31 50 48 35
45 39 21 29 10 33 32 41 36  0 34 16 22
19 14 37  2 24 42  6  7 30 46 47 28 26
38 23 40 13  4  9  5 12 49  1 44 43 17
```

This brings us to the end of Chapter 19.

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 Plus and Conjugate

Dyadic + is arithmetic addition.

2 + 2	1.5 + 0.25	3j4 + 5j4	2r3 + 1r6
4	1.75	8j8	5r6

Monadic + is "Conjugate". For a real number y, the conjugate is y. For a complex number xjy (that is, x + 0jy), the conjugate is x - 0jy.

+ 2	+ 3j4
2	3j_4

20.2 Minus and Negate

Dyadic - is arithmetic subtraction.

2 - 2	1.5 - 0.25	3 - 0j4	2r3 - 1r6
0	1.25	3j_4	1r2

Monadic - is "Negate".

- 2	- 3j4
_2	_3j_4

20.3 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.

`<: 3`	`<: 2.5`	`<: 2r3`	`<: 2j3`
`2`	`1.5`	`_1r3`	`1j3`

20.4 Times and Signum

Dyadic `*` is multiplication.

`2 * 3`	`1.5 * 0.25`	`3j1 * 2j2`	`2r3 * 7r11`
`6`	`0.375`	`4j8`	`14r33`

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

20.5 Division and Reciprocal

Dyadic `%` is division.

2 % 3	1.4 % 0.25	3j4 % 2j1	12x % 5x
0.666667	5.6	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`

20.6 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`

20.7 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	<. 4r3
2	3	_4	1

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`

>. 3.4j3.4	>. 3.8j3.8
3j4	4j4

20.8 Power and Exponential

Dyadic ^ is the "power" verb: (x^y) is x raised-to-the-power y

10 ^ 2	10 ^ _2	100 ^ 1%2
100	0.01	10

Monadic ^ is exponentiation (or antilogarithm): ^y means (e^y) where e is Euler's constant, 2.71828...

^ 1	^ 1.5	^ 3r2	^ 0j1
2.71828	4.48169	4.48169	0.540302j0.841471

Euler's equation, supposedly engraved on his tombstone is:
$e^{i\pi}+1=0$

```
   (^ 0j1p1) + 1
0j1.22465e_16
```

The example of ^ 3r2 above shows that rationals are in the domain of ^ , but the result is real, not rational. A rational argument is in effect first converted to real.

20.9 Square

Monadic `*:` is "Square".

`*: 4`	`*: 2j1`
`16`	`3j4`

20.10 Square Root

Monadic `%:` is "Square Root".

`%: 9`	`%: 3j4`	`2j1 * 2j1`
`3`	`2j1`	`3j4`

20.11 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 ^ % x)`

x =: 3 3.1	x %: 8	8 ^ % x
3 3.1	2 1.95578	2 1.95578

20.12 Logarithm and Natural Logarithm

Dyadic `^.` is the base-x logarithm function, that is, `(x ^. y)` is the logarithm of `y` to base `x` :

10 ^. 1000	2 ^. 8	10 ^. 2j3	10 ^. 2r3
3	3	0.556972j0.426822	_0.176091

Monadic `^.` is the "natural logarithm" function.

e =: ^ 1	^. e	^. 2j3	^. 2r3
2.71828	1	1.28247j0.982794	_0.405465

The example of `^. 2r3` above shows that rationals are in the domain of `^.` but the result is real, not rational. A rational argument is in effect first converted to real.

20.13 Factorial and OutOf

The factorial function is monadic `!`.

! 0 1 2 3 4	! 5x 6x 7x 8x
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`

1 ! 4	2 ! 4	3 ! 4
4	6	4

20.14 Magnitude and Residue

Monadic `|` is called "Magnitude". For a real number `y` the magnitude of `y` is the absolute value:

| 2	| _2
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`	`0j_1`	`0`	`1`

20.15 GCD and LCM

The greatest common divisor (GCD) of `x` and `y` is given by `(x +. y)`. Reals and rationals are 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

The Least Common Multiple of `x` and `y` is given by `(x *. y)`.

(2 * 3) *. (3 * 5)	2*3*5
30	30

20.16 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...

o. 1	o. 2	o. 1r6	o. 2j3
3.14159	6.28319	0.523599	6.28319j9.42478

The example of `o. 1r6` above shows that rationals are in the domain of `sin` but the result is real, not rational. A rational argument is in effect first converted to real.

A result with arbitrary precision can be be produced by giving an argument of an extended integer to the verb `<. @ o.` for which there is special code in the J interpreter

```
   ] z =: (<. & o.) 10^40x
3141592653589793238462643383279502884197 1
   datatype z
extended
```

20.17 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 (π over 6) is 0.5

`y =: o. 1r6`	`1 o. y`
`0.523599`	`0.5`

The general scheme for dyadic `o.` is that `(k o. y)` means: apply to `y` a function selected by `k`. Here `y` is an angle in radians.

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 hyperb.  tangent
```

y	sin y	asin sin y	sin 1r4
0.523599	0.5	0.523599	0.247404

The example of `sin 1r4` above shows that rationals are in the domain of `sin` but the result is real, not rational. A rational argument is in effect first converted to real.

20.18 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

and a further group of functions on complex numbers:

```
 9 o. xjy   means  x                  (real part)
10 o. xjy   means  %: (x^2) + (y^2)   (magnitude)
11 o. xjy   means  y                  (imag part)
12 o. xjy   means  atan (y % x)       (angle)
```

9 o. 3j4	10 o. 3j4	11 o. 3j4	12 o. 3j4
3	5	4	0.927295

and finally

```
 _9 o. xjy   means xjy                  (identity)
_10 o. xjy   means x j -y               (conjugate)
_11 o. xjy   means 0j1 * xjy            (j. xjy)
_12 o. a     means (cos a)+(j. sin a)   (inverse
                                           angle)
```

For example:

a =: 12 o. 3j4	_12 o. a
0.927295	0.6j0.8

This is the end of chapter 20

Chapter 21: Factors and Polynomials

In this chapter we look at the built-in functions:

- monadic `q:` which computes the prime factors of a number
- dyadic `q:` which represents a number as powers of primes
- monadic `p:` which generates prime numbers
- dyadic `p.` which evaluates polynomials
- monadic `p.` which finds roots of polynomials

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 a large number (greater than about `2^53`), its value should be specified as an extended integer to ensure all its significant digits are supplied to `q:` ..

```
   q: 1 + 2 ^ 53x
3 107 28059810762433
```

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: 2^27x`
`1339484207`	`error`	`error`

21.2 Polynomials

21.2.1 Coefficients

If `x` is a variable, then an expression in conventional notation such as

$$a + bx + cx^2 + dx^3 + \ldots$$

is said to be a polynomial in `x`. If we write `C` for the list of coefficients `a,b,c,d ...`, for example,

```
C =: _1 0 1
```

and assign a value to `x`, for example,

```
x=:2
```

then the polynomial expression can be written in J in the form `+/ C * x ^ i. # C`

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`,

+/C*x^i.# C	C p. x
3	3

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.`

```
   p =: _1 0 1 & p.    p x
_1 0 1&p.              3
```

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`.

```
   C      p =: C & p.    Z =: p. C
_1 0 1    _1 0 1&p.      +-+----+
                         |1|1 _1|
                         +-+----+
```

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
+-+----+         1 _1   0 0
|1|1 _1|
+-+----+
```

The significance of the multiplier `M` is as follows. If we write `r,s,t...` 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	+-+----+ |1|1 _1| +-+----+	_1 0 1

21.2.3 Multiplier and Roots

We saw above that the left argument of `p.` can be a list of coefficents, with the scheme

```
C p. x   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) p. x   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	+-+----+ |1|1 _1| +-+----+	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 coefficents and instead specifying a list of coefficient-exponent pairs. Here is an example. With the polynomial `_1 0 1 & p.`, the nonzero coefficents are the first and third, `_1 1`, and the corresponding exponents are `0 2`. We form the pairs thus:

coeffs =: _1 1	exps=: 0 2	pairs =: coeffs ,. exps
_1 1	0 2	_1 0 1 2

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.

x	pairs	(< pairs) p. x	_1 0 1 p. x
2	_1 0 1 2	3	3

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

x=: 2	f x	g x
2	_0.0337641j3.4936 2	_0.0337641j3.49362

This is the end of Chapter 21.

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. A sum-of-products verb can be composed from **`sum`** and **`product`** with the `@:` conjunction.

`P =: 2 3 4`	`Q =: 1 0 2`	`P * Q`	`+/ P * Q`	`P (+/ @: *) Q`
`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.

`P`	`Q`	`P (+/ @: *) Q`	`P (+/ . *) Q`
`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`:

`dot =: +/ . *`	`P`	`Q`	`P dot Q`
`+/ .*`	`2 3 4`	`1 0 2`	`10`

A textbook definition of scalar product of vectors `P` and `Q` may appear in the form:

```
(magnitude P) * (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`.

`ma 3 4`	`P ca P`
`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 Generalizations

22.4.1 Various Verbs

The "Dot Product" conjunction forms the dot-product verb with `(+/ . *)`. 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 `(+./ . *.)`.

```
g   =: +. / . *.
```

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

By 'symbolic arithmetic' is meant, for example, symbolically adding 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 (boxed) 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
+-+ |a| +-+	+-+ |b| +-+	+-+ |c| +-+	+---+ |a+b| +---+	+-------+ |a*(b+c)| +-------+

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
+---+ |a*b| +---+	+--+ |ab| +--+

As arguments to the "Dot Product" conjunction we could supply verbs to perform symbolic arithmetic. 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|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)| +----------+----------+

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`.

$A	$B	$Z
1 2 3	3 2 2	1 2 2 2

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 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 : 'u b. 0'
LRANK   =: 1 : '1 { (u 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'
+--+--+
|a |b |
+--+--+
|ma|mb|
+--+--+
```

```
sdet A
+-------+
|amb-mab|
+-------+
```

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:

```
3x + 4y = 11

2x + 3y =  8
```

Rewriting as a matrix equation, we have, informally,

```
M 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:

```
   M =: 3 4 ,: 2 3
3 4
2 3
   R =: 11 8
11 8
```

The vector of unknowns `U` (that is, `x,y`) can be found by dividing `R` by matrix `M`.

```
   M
3 4
2 3
   R
11 8
   U =: R %. M
   M dot U
11 8
```

We see that `M 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
3 4
2 3
   R =: 15j22 11j16
15j22 11j16
   U =: R %. M
1j2 3j4
   M dot U
15j22 11j16
```

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 dot U
3 4 2 3	15r22 11r16	_31r44 123r176	15r22 11r16

In the previous examples the unknowns in `U` were scalars. Now suppose the unknowns are vectors and our equations for solving are:

```
3x + 4y = 15 22

2x + 3y = 11 16
```

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 =: 10 20 30
y =: 31 49 70
```

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:

```
1 . a  +  10 . b  =  31

1 . a  +  20 . b  =  49

1 . a  +  30 . b  =  70
```

so we take the matrix of coefficients `M` to be

```
M =: 3 2 $ 1 10  1 20  1 30
```

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 10`
`1 20`
`1 30`
	`31 49 70`	`11 1.95`	`30.5 50 69.5`

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 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)`
`1.5`	`1.6523`

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:

```
x =: 0   1  2  3
y =: 1   6 17 34.1
```

The four equations to be solved are:

$$\begin{aligned}
1.a + bx_0 + cx_0^2 &= y_0 \\
1\ a + bx_1 + cx_1^2 &= y_1 \\
1.a + bx_2 + cx_2^2 &= y_2 \\
1.a + bx_3 + cx_3^2 &= y_3
\end{aligned}$$

and so the columns of matrix `M` are `1, x, x^2`, conveniently given by: `x ^/ 0 1 2`

```
M =: x ^/ 0 1 2
```

and the unknowns `a, b, c` are given by vector `U` as follows:

M	y	U =: y %. M	M dot U
1 0 0 1 1 1 1 2 4 1 3 9	1 6 17 34.1	1.005 1.955 3.025	1.005 5.985 17.02 34.09

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
```

M	R	U =: R %. M	M dot U	M dot"2 1 U
3 4 2 3 0 3 1 2 3 1 2 3	21 42	_105 84 28 7 3 12	error	21 42 21 42 21 42

The dyadic rank of `%.` is `_ 2`,

```
   %. b. 0
2 _ 2
```

and so 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 dot 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 dot 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 7 0 0 4 6 0 3	1 0 0 0 1 0 0 0 1	0 _0.125 0.1667 0.25 _0.3438 _0.125 0 0.25 0	0 _0.125 0.1667 0.25 _0.3438 _0.125 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	0.08824 0.1471	1	1

This is the end of Chapter 22.

Chapter 23: Calculus

This chapter covers J operators for differentiation and integration. It covers

- The conjunction `d.` which differentiates and integrates analytically, that is, it transforms expressions denoting functions into expressions denoting functions.
- The conjunction `D.` which differentiates numerically, and thus broadens the range of functions which can be differentiated. It also covers partial derivatives.
- A library script with functions for numerical integration.

23.1 Differentiation

There is a built-in conjunction `d.`(lowercase d dot). Its left argument is a function to be differentiated. Its right argument is `1` if the first derivative is required, or `2` for the second derivative, and so on. The first derivative of the "cube" function `^&3` is "3 times the square".

```
   ^&3  d. 1
3&*@(^&2)
```

The general scheme is that if `e` is (an expression denoting) a function, then `e d. n` is (an expression denoting) the n'th derivative of `e`. Here is another example, expressing the cube

function as the polynomial `0 0 0 1 & p.`

```
   0 0 0 1 &p. d. 1
0 0 3&p.
```

Suppose we define a verb `cube`:

`cube =: 0 0 0 1 & p.`	`cube 2`
`0 0 0 1&p.`	`8`

Differentiating with `d.`, we see that the derivative is, as expected, 3-times-the-square, but the expression for the derivative is not very informative.

`(cube d. 1) 2`	`cube d. 1`
`12`	`cube d.1`

The reason is that `cube` is a name denoting a verb, and such names are in general not evaluated until the verb is applied. (See Appendix 1 .) If we want to inspect the derivative of `cube`, we can force evaluation of the name `cube` by applying the `f.` adverb.

`cube d. 1`	`(cube f.) d. 1`
`cube d.1`	`0 0 3&p.`

Alternatively, we could force evaluation of the expression for the derivative, again by applying `f.`

cube d. 1	(cube d. 1) f.
cube d.1	0 0 3&p.

23.2 Integration

With a right argument of _1, the conjunction d. integrates the left argument. The integral of "3 times the square" is "cube".

```
   0 0 3 & p. d. _1
0 0 0 1&p.
```

23.3 The Domain of d.

Functions which are differentiable or integrable with d. must firstly be scalar. That is, they must take scalar arguments and deliver scalar results, and all intermediate quantities must be scalars. Here is an example. The function "(x-1)*(x-2)" can be written in several different ways. Here are two:

f =: -&1 * -&2	g =: (*/) @: (- & 1 2)	f 3	g 3
-&1 * -&2	*/@:(-&1 2)	2	2

f is scalar, and in the domain of d. However, g is not scalar, because it forms the intermediate quantity x - 1 2 which is a vector. Thus g is not in the domain of d.. To demonstrate this, we force evaluation of the derivatives.

`(f d. 1) f.`	`(g d. 1) f.`
`_3 2x&p.`	`error`

Secondly, `d.` can differentiate constant functions, polynomials, exponentials `^x` and integral powers `x^n`.

`3: d. 1`	`0 2 &p. d. 1`	`^ d. 1`	`^&4 d. 1`
`0"0`	`2"0`	`^`	`4&*@(^&3)`

If `f` and `g` are differentiable with `d.`, then so are the forks `(f+g)`, `(f-g)`, `(f*g)` and `f%g`.

`f =: ^&3`	`g =: 0 2 & p.`	`((f + g) d. 1) f.`
`^&3`	`0 2&p.`	`2 0 3x&p.`

Trigonometric functions are differentiable with `d.` The derivative of the fork `(sin + cos)` is `(cos - sin)`.

`sin =: 1&o.`	`cos =: 2&o.`	`(sin + cos) f. d. 1`
`1&o.`	`2&o.`	`2&o. + -@(1&o.)`

If `f` and `g` are differentiable with `d.`, then so are the compositions `f@g` and `f@:g`

f	g	(f @ g d. 1) f.
^&3	0 2&p.	0 0 24x&p.

23.4 The Conjunction D.

The conjunction `D.` (uppercase D dot) computes derivatives. It differs from `d.` in two ways.

- By using numerical methods it can differentiate arbitrary functions, that is, it is not limited to the domain of `d.`
- It is not limited to scalar functions: it can differentiate functions with vector arguments to produce partial derivatives.

23.4.1 The Domain of D.

Since `D.` can use numerical methods, its arguments can be arbitrary functions. For example, recall the function `g` above, to compute "(x-1)*(x-2)",which was demonstrated above to be outside the domain of `d.` . However it is within the domain of `D.`. Its derivative is "2x-3"

g =: (*/) @: (- & 1 2)	(g d. 1) 3	(g D. 1) 3
*/@:(-&1 2)	error	3

23.4.2 Partial Derivatives with D.

Next we look at functions which compute a scalar from a vector argument. For example consider a surface where the height at a point `(x,y)` is given by

```
(sin x) * (cos y)
```

The height-function, with the vector argument `(x,y)` might be written:

```
h =: (sin @ {.) * (cos @ {:)
```

The expression `(h D.1)(x,y)` computes the numerical values of the two slopes, in the x-direction and the y-direction, of the function `h` at the point `(x,y)`.

x=: 0.4	y =: 0.5	p =: h D. 1 x,y
0.4	0.5	0.8083071 _0.1866971

The result `p` gives the values of the partial derivatives with respect to `x` and with respect to `y`.

We can check this result. Suppose we define a function `q` say, for the height along the line `y=0.5`. We want `q(x)` to be `h(x,0.5)` and thus

```
q =: h @: (, & 0.5)
```

The idea now is that the derivative of `q` applied to argument `x` should be the same as the first partial derivative of `h` at `x,0.5`.

h D.1 x,y	q D. 1 x
0.8083071 _0.1866971	0.8083071

Now we look at partial derivatives of functions which compute vectors from vectors. Here is an example, a function which takes the point (`x,y,z`) in 3-space to the point `(2x,3y)` in 2-space.

foo =: (2 3 & *) @: (1 1 0 & #)	foo 1 1 1
2 3&*@:(1 1 0&#)	2 3

In general such a function will take an argument-vector of length `m` and produce a result-vector of length `n`. Hence there will be `m*n` partial derivatives, one for each element of the result with respect to each element of the argument. The six partial derivatives of `foo` at the point `xyz = 1 1 1` are shown by:

pd =: foo D. 1	pd 1 1 1
foo D.1	2 0 0 3 0 0

Consider now a function such as `cube` which produces scalars from scalars. Given a vector argument, it will produce a vector result of the same length, where an element of the result depends only on the corresponding element of the argument.

cube	cube a =: 1 2 3
0 0 0 1&p.	1 8 27

Therefore, for a scalar function, all partial derivatives are zero except those for elements of the result with respect to the corresponding elements of the argument.

pd =: cube D. 1	pd 2 3 4
cube D.1	12 0 0 0 27 0 0 0 48

If a scalar function is given in fully-evaluated form, and is in the domain of `d.`, the `D.` conjunction will produce an analytic expression for the partial derivatives function:

PD =: (0 0 0 1 & p.) D.1	PD 2 3 4
(* =/~@(i.@$))@:(0 0 3&p.)	12 0 0 0 27 0 0 0 48

23.5 Numerical Integration

There is a library script-file called `integrat.ijs` . It contains several different operators for integration. Documentation is given in the script file.

It can be be downloaded from the JSoftware website: here is a link to integrat.ijs

Assuming that we have downloaded into a directory, say `C:\temp` for example, then we load it into the J session with:

```
load 'c:\temp\integrat.ijs'
```

One of the integration operators provided is the conjunction `adapt` ("numeric integration by adaptive quadrature"). The expression `f adapt (L,U)` computes the numeric value of the definite integral of `f` between limits `L` and `U`. For example, we expect the integral of `3&*` between `0` and `1` to be `1.5`

`f =: 3&*`	`f adapt 0 1`
`3&*`	`1.5`

This is the end of Chapter 23.

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 **N_L_** (N underbar 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 **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) < '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) < '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) < '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)  < '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`. The resulting long list of names can be conveniently displayed with the library-verb `list` .

```
   list (4 !: 1) 0 1 2 3  NB. show all names in current locale

ASSERTING CH        IP        RUN       RUNR      TD        THIS
and       cd        corr      dir       drop      e         first
fst       hello     implies   indexfile indexing  is_bool   is_box
is_char   is_cmplx  is_extint is_float  is_int    is_list   is_number
is_rat    is_real   is_scalar is_string last      most      not
p         print     pwd       rest      run       s         snd
take      thd       y
```

We can define a new verb, and see its name appear in the list:

```
   foo  =: +/
   list (4 !: 1) 0 1 2 3

ASSERTING CH        IP        RUN       RUNR      TD        THIS
and       cd        corr      dir       drop      e         first
foo       fst       hello     implies   indexfile indexing  is_bool
is_box    is_char   is_cmplx  is_extint is_float  is_int    is_list
is_number is_rat    is_real   is_scalar is_string last      most
not       p         para      print     pwd       rest      run
s         snd       take      thd       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**)

```
   (4 !: 1) 0 1 2 3
```

```
   list (4 !: 1) 0 1 2 3
ASSERTING CH        IP        RUN       RUNR      TD        THIS
and       cd        corr      dir       drop      e         first
foo       fst       hello     implies   indexfile indexing  is_bool
is_box    is_char   is_cmplx  is_extint is_float  is_int    is_list
is_number is_rat    is_real   is_scalar is_string last      most
not       p         para      print     pwd       rest      run
s         snd       take      thd       y
```

but nevertheless we can evaluate the simple name **q** to get its value as defined in locale **z**.

```
   q
99
```

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:

`coname ''`	show name of current locale
`conl 0`	show names of all locales
`names ''`	show all names in current locale
`nl ''`	show all names in current locale (as a boxed list)
`cocurrent 'foo'`	locale `foo` becomes current
`clear 'foo'`	remove all defns from locale `foo`
`coerase 'A';'B';'C'`	erase locales `A` `B` and `C`

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

Suppose `A_X_` is an adverb in locale `X`. The application of `A_X_` to an argument takes place in locale `X` rather than in the current locale.

To demonstrate this, we start by entering definitions in fresh locales `X` and `Y`.

```
A_X_ =: 1 : 'u & k'    NB. an adverb
k_X_ =: 17
k_Y_ =: 6
```

now make `Y` the current locale:

```
cocurrent 'Y'
```

and apply adverb `A_X_` to argument `+` .

`view 'X'`	`view 'Y'`	`+ A_X_`
`A =: 1 : 'u & k'` `k =: 17`	`k =: 6`	`+&17`

Evidently the result is produced by taking `k` from locale `X` rather than from the current locale which is `Y`.

The scheme is that if the current locale is `Y`, and `A` is an adverb, the expression `f A_X_` means:

1. evaluate `f` in locale `Y` to get a value `F` say.
2. cocurrent `X`
3. evaluate `F A` in locale `X` to get a result `G` say.
4. cocurrent `Y`
5. `G`

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
g =: +&1 k =: 0 n =: 7	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
g =: +&1 k =: 0 n =: 7	f =: g g =: +&2 k =: 6 n =: 2 u =: 3 : 'y+k'	2	1

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_ < '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'
g k n
```

24.10 Erasing Names from Locales

Suppose we create a variable with the name **var**, say,

```
   var =: 'hello'
```

and demonstrate that it exists, that is, that the name **var** is one of the names in the namelist of the **base** locale:

```
   (<'var') e. nl_base_ ''
1
```

Now we can erase it with the **erase** library verb:

```
   erase <'var'
1
```

and demonstrate that it no longer exists

```
   (<'var') e. nl_base_ ''
0
```

Now suppose that we create a variable **foo**, say, in the base locale, and another, also called **foo**, in the **z** locale. Recall that there is always a path from **base** to **z**

```
   foo    =: 'hello'
   foo_z_ =: 'goodbye'
```

we demonstrate they both exist:

```
   (<'foo') e. nl_base_ ''
1
   (<'foo') e. nl_z_ ''
1
```

erase **foo** from **base**, demonstrate that it has gone but that **foo** in **z** is still there:

erase <'foo'	(<'foo') e. nl_base_ ''	(<'foo') e. nl_z_ ''
1	0	1

Now if we erase again, **foo** will be found along the path and erased from **z**.

erase <'foo'	(<'foo') e. nl_base_ ''	(<'foo') e. nl_z_ ''
1	0	0

This is the end of Chapter 24

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, mostly with names beginning "co", meaning "class and object". 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 which we choose to call "Stack". A new class is introduced with the library function **coclass**.

```
   coclass 'Stack'
```

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, a method for inspecting the "top" (most-recently added)

item on the stack. It returns that value of that item. The stack is unchanged.

```
top =: 3 : '> {. items'
```

Next a method to remove the top item of the stack.

```
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 `codestroy`.

```
destroy =: codestroy
```

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:

```
o =: conew 'Class'
```

```
create__o arg
```

can be abbreviated as:

```
o =: 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 Listing the Classes and Objects

In this section we look at inspecting the population of objects and classes we have created. The expression **(18!:1) 0 1** produces a list of all existing locales.

```
   (18!:1) 0 1
+-+-+-----+----+----+-+-------+--------+------+-----+-+
|0|1|Stack|base|ctag|j|jadetag|jcompare|jregex|jtask|z|
+-+-+-----+----+----+-+-------+--------+------+-----+-+
```

We see here the names of locales of 3 different kinds. Firstly, there are ordinary locales such as **base**, and **z**, described in Chapter 24. These are created automatically by the J system. Depending on

the version of J you are using, you may see a list different from the one shown here.

Secondly, there are locales such as `Stack`. The `Stack` locale defines the `Stack` class. If we view this locale (with the `view` utility function from Chapter 24)

```
   view 'Stack'
IP      =: 1
create  =: 3 : 'items =: 0 $ 0'
destroy =: codestroy
pop     =: 3 : '#  items =: }. items'
push    =: 3 : '# items =: (< y) , items'
top     =: 3 : '> {. items'
```

we see a variable `IP` (created automatically) and our methods which we defined for `Stack`.

Thirdly, we have locales such as `0`. Here the name is a string of numeric digits (that is, `'0'`). Such a locale is an object. The variable `S` has the value `<'0'`, so that here object `S` is locale `'0'`.

```
   S
+-+
|0|
+-+
   view >S
COCREATOR =: <'base'
items     =: <;._1 '|how are you?|hello'
```

We see a variable `COCREATOR`, which identifies this locale as an object, and the field(s) of the object.

The path from an object is given by the verb `18!:2`

```
   18!:2 S
+-----+-+
|Stack|z|
+-----+-+
```

Since `S` is a `Stack` object, the first locale on its path is `Stack`. Recall from Chapter 24 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 the `items` variable of locale `'0'`) are available.

2. 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.
3. Restore the current locale to the status quo.

Here is a utility function to list all the existing objects and their classes.

```
   obcl =: 3 : '(, ({. @: (18!:2)))"0  (18!:1) 1'
```

Currently we have variables `S` and `T` each referring to a `Stack` object.

S	T	obcl ''
+-+ |0| +-+	+-+ |1| +-+	+-+-----+ |0|Stack| +-+-----+ |1|Stack| +-+-----+

(Again, depending on the version of J you are using, you may see further objects and classes automatically generated by the J system for its own use.)

A Stack, `S` say, can be removed using the `destroy` method of the `Stack` class.

```
   destroy__S  ''
1
```

We see it has gone.

```
   obcl ''
+-+-----+
|1|Stack|
+-+-----+
```

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'
   create  =: 3 : 'items =: 0 $ 0'
   add     =: 3 : '# items =: (< y) , items'
   remove  =: 3 : '# items =: items -. < y'
   inspect =: 3 : 'items'
   destroy =: codestroy
```

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 `coinsert`.

```
   coclass 'Set'
   coinsert 'Collection'
```

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 : 0
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 : 0
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 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.

```
   add_Collection_ f.
3 : '# items =: (< y) , items'
```

25.7 Library Verbs

Here is a brief summary of selected library verbs.

`coclass 'foo'`	introduce new class `foo`
`coinsert 'foo'`	this class to be a child of `foo`
`conew 'foo'`	introduce a new object of class `foo`
`conl 0`	list locale names
`conl 1`	list ids of object locales
`names_foo_ ''`	list the methods of class `foo`
`copath <'foo'`	show path of class `foo`
`coname ''`	show name of current locale

This brings us to the end of Chapter 25

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.

The plan for this chapter is to look at

- built-in verbs for loading scripts
- the `load` verb and its advantages, including convenient loading of library scripts
- the "profile" script automatically loaded at the beginning of a J session

26.1 Creating Scripts

It will be useful to identify a directory where we intend to store our own scripts.

There is a directory `j701-user` convenient for the purpose. It is installed automatically as part of a J installation. Its full pathname is given by

```
   jpath '~user'
c:/users/homer/j701-user
```

A variable, `scriptdir` say, can hold the name of our chosen script directory together with a trailing '/'

```
   ] scriptdir =: (jpath '~user') , '/'
c:/users/homer/j701-user/
```

Scripts are usually created using a text editor, but we can use J to create small examples of scripts as we need them. Here is an example of creating a tiny script, with a filename of say `example.ijs`, using the built-in verb `1!:2` thus:

```
   (0 : 0) (1!:2) < scriptdir,'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 a filename as a boxed string.

```
   0!:1 < scriptdir,'example.ijs'
   plus =: +
   k    =: 2 plus 3
   k plus 1
6
```

We see on the screen a display of 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

The verb `0!:1`, as we saw, loads a script with a display. If there is an error in the script, `0!:1` will stop. We can choose whether or not to display, and whether to stop or to continue loading after an error. There are four similar verbs:

`0!:0`	no display	stopping on error
`0!:1`	with display	stopping on error
`0!:10`	no display	continuing on error
`0!:11`	with display	continuing on error

For example:

```
0!:0 < scriptdir,'example.ijs'
```

We see nothing on the screen. The value computed in the script for `k plus 1` is discarded.

26.3 The load Verb

There is a verb `load` which is predefined, that is, automatically available in the standard J setup. It can be used just like `0!:0` to load a script

```
   load < scriptdir,'example.ijs'
```

The script is loaded without a display and stopping on error. There is a companion verb `loadd` which loads with a display, stopping on error.

```
   loadd < scriptdir, 'example.ijs'
   plus =: +
   k    =: 2 plus 3
   k plus 1
6
```

`load` and `loadd` have several advantages compared with `0!:n` . The first of these is that the filename need not be boxed.

```
   loadd scriptdir, 'example.ijs'
   plus =: +
   k    =: 2 plus 3
   k plus 1
6
```

26.4 Local Definitions in Scripts

Now we look at the treatment of local variables in scripts. Here is an example of a script.

```
   (0 : 0) (1!:2) < scriptdir, 'ex1.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.

Firstly, we need to 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) < scriptdir, 'ex2.ijs'
w   =. 1 + 1
foo =: + & w
)
```

Secondly, we need something for `w` to be local to, that is, an explicit function, because outside any explicit function (that is, "at the top level") `=.` is the same as `=:` All that would be needed is the merest wrapper of explicit definition around `0!:n`, such as:

```
   LL =: 3 : '0!:0 y'
```

If we now load our script

```
   LL < scriptdir, 'ex2.ijs'
```

and then look at the results:

foo	w
+&2	error

we see that `foo` is as expected, and, as intended, 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`.

An advantage of the `load` verb is that it provides the explicit function needed to make `w` local.

```
   erase 'foo';'w'
1 1

   load scriptdir, 'ex2.ijs'
```

foo	w
+&2	error

26.4.1 Local Verbs in Scripts

In the previous example, the local variable `w` was a noun. With a local verb, there is a further consideration. 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) < scriptdir, 'ex3.ijs'
sum  =. +/
mean =: sum % #
)

   load  < scriptdir, 'ex3.ijs'
```

We see that this will not work, because mean needs `sum` and `sum`,

being local, is no longer available.

mean	sum
sum % #	error

The remedy is to "fix" the definition of `mean`, with the adverb `f.` (as we did in Chapter 12). Our revised script becomes

```
   (0 : 0) (1!:2) < scriptdir, 'ex4.ijs'
sum =. +/
mean =: (sum % #)  f.
)
```

Now `mean` is independent of `sum`

```
   load  < scriptdir, 'ex4.ijs'
```

mean	sum
+/ % #	error

26.5 Loading Into Locales

We looked at locales in Chapter 24. When we load a script with `0!:n` or `load` it is the current locale that becomes populated with definitions from the script.

By default, the current locale is `base`. In general, we may wish to load a script into a specified locale, say locale `one`.

Here is one way:

```
   load_one_  scriptdir, 'example.ijs'

   plus_one_
+
```

Another way is to let the script itself specify the locale. For example,

```
   (0 : 0) (1!:2) < scriptdir, 'ex5.ijs'
18!:4 < 'two'
w   =. 1 + 1
foo =: + & w
)
```

and then the script steers itself into locale `two`

```
   load scriptdir, 'ex5.ijs'

   foo_two_
+&2
```

Here is a further advantage of `load` compared with `0!:n`. Notice that the current locale is `base`.

```
   18!:5 ''  NB. current locale before loading
+----+
|base|
+----+
```

If we now `load ex5.ijs`, the current locale is still `base` afterwards, regardless of the fact that the script visited locale `two`.

```
   load scriptdir,'ex5.ijs'
   18!:5 ''  NB. current locale after loading
+----+
|base|
+----+
```

However, loading the same script with `0!:n` does NOT restore the previously current locale.

```
   18!:5 '' NB. current locale before loading
+----+
|base|
+----+
   0!:0 < scriptdir,'ex5.ijs'
   18!:5 '' NB.  current locale after loading
+---+
|two|
+---+
```

so we conclude that self-steering scripts should be loaded with `load` and not with `0!:n`.

We return to base.

```
   18 !: 4 < 'base'
```

26.6 Repeated Loading, and How to Avoid It

Another 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. The predefined verb **require** is like **load** but does not load a script if it is already loaded.

Here is a demonstration. Suppose we have these two lines for the first script:

```
    (0 : 0) (1!:2) < scriptdir,'first.ijs'
     require scriptdir, '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) < scriptdir, 'second.ijs'
     b  =: b + 1
)
```

We set the counters **a** and **b** to zero, load the first script and inspect the counters:

(a =: 0),(b =: 0)	load scriptdir, 'first.ijs'	a,b
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:

load scriptdir,'first.ijs'	a,b
	2 1

26.7 Load Status

The J system keeps track of ALL scripts loaded in the session, 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. Here are the last few entries in this report for the current session.

```
   ,. _4 {. 4!:3 ''
+----------------------------------+
|c:\users\homer\j701-user\ex4.ijs   |
+----------------------------------+
|c:\users\homer\j701-user\ex5.ijs   |
+----------------------------------+
|c:\users\homer\j701-user\first.ijs |
+----------------------------------+
|c:\users\homer\j701-user\second.ijs|
+----------------------------------+
```

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`

`i =: 4!:4 < 'plus'`	`i { 4!:3 ''`

`14`	`+----------------------------------+`
	`|c:\users\homer\j701-user\example.ijs|`
	`+----------------------------------+`

26.8 Library Scripts

26.8.1 The Standard Library

The J system comes supplied with script files containing a useful collection of predefined functions.

We can look at the list of scripts loaded automatically at the beginning of the session. For this we use the built-in verb `4!:3` to generate a boxed list of file-names. Here are the first 9 scripts:

```
   ,. 9 {. 4 !: 3 ''
+-------------------------------------------+
|C:\users\homer\j701\bin\profile.ijs         |
+-------------------------------------------+
|C:\users\homer\j701\system\util\boot.ijs    |
+-------------------------------------------+
|C:\users\homer\j701\system\main\stdlib.ijs  |
+-------------------------------------------+
|C:\users\homer\j701\system\util\scripts.ijs |
+-------------------------------------------+
|C:\users\homer\j701\system\main\regex.ijs   |
+-------------------------------------------+
|C:\users\homer\j701\system\main\task.ijs    |
+-------------------------------------------+
|C:\users\homer\j701\system\util\configure.ijs|
+-------------------------------------------+
|c:\users\homer\j701-user\config\recent.dat  |
+-------------------------------------------+
|c:\users\homer\j701\system\main\ctag.ijs    |
+-------------------------------------------+
```

We see that among these is the script-file **stdlib.ijs**

Functions defined in **stdlib.ijs** end up in the **z** locale where they are conveniently available to the programmer. There are more than 300 things defined in the **z** locale:

```
   # nl_z_ ''
369
```

For example, the file-handling utility functions documented in the J User Manual are found in the **z** locale with names beginning with the letter 'f'.

```
   6 6 $ 'f' nl_z_ ''
+--------+---------+--------+----------+---------+-------------+
|f2utf8  |fappend  |fappends|fapplylines|fboxname |fc           |
+--------+---------+--------+----------+---------+-------------+
|fcompare|fcompares|fcopynew|fdir      |ferase   |fetch        |
+--------+---------+--------+----------+---------+-------------+
|fexist  |fexists  |fgets   |fi        |flatten  |fliprgb      |
+--------+---------+--------+----------+---------+-------------+
|fmakex  |foldpara |foldtext|fpathcreate|fpathname|fputs        |
+--------+---------+--------+----------+---------+-------------+
|fread   |freadblock|freadr |freads    |frename  |freplace     |
+--------+---------+--------+----------+---------+-------------+
|fsize   |fss      |fssrplc |fstamp    |fstring  |fstringreplace|
+--------+---------+--------+----------+---------+-------------+
```

26.8.2 The J Application Library

There is an extensive collection of script-files forming the J Application Library (JAL). The JAL is documented here

26.9 User-Defined Startup Script

Suppose we have a collection of our own definitions which we wish to be loaded automatically at the beginning of every J session. To achieve this we can put our definitions into a script-file which must be named `startup.ijs`. The full pathname for this file is given by the expression

```
   jpath '~config/startup.ijs'
c:/users/homer/j701-user/config/startup.ijs
```

Here is an example. We create the script-file with a few definitions. For this example we could define a few verbs useful for type-checking.

```
   (0 : 0)  (1 !: 2) < jpath '~config/startup.ijs'
is_int    =: 4 = 3 !: 0
is_char   =: 2 = 3 !: 0
is_number =: 1 4 8 16 64 128  e.~  3!:0
is_scalar =: 0 = # @: $
is_list   =: 1 = # @: $
is_string =: is_char *. is_list
)
```

With this script-file in place, the next session should automatically load it. We verify this by looking at the list of scripts loaded at the beginning of the new session.

```
   ,. 11 {. 4 !: 3 ''
```

```
+-----------------------------------------+
|C:\users\homer\j701\bin\profile.ijs       |
+-----------------------------------------+
|C:\users\homer\j701\system\util\boot.ijs  |
+-----------------------------------------+
|C:\users\homer\j701\system\main\stdlib.ijs|
+-----------------------------------------+
|C:\users\homer\j701\system\util\scripts.ijs|
+-----------------------------------------+
|C:\users\homer\j701\system\main\regex.ijs |
+-----------------------------------------+
|C:\users\homer\j701\system\main\task.ijs  |
+-----------------------------------------+
|C:\users\homer\j701\system\util\configure.ijs|
+-----------------------------------------+
|c:\users\homer\j701-user\config\recent.dat|
+-----------------------------------------+
|c:\users\homer\j701\system\main\ctag.ijs  |
+-----------------------------------------+
|c:\users\homer\j701\system\util\jadetag.ijs|
+-----------------------------------------+
|c:\users\homer\j701-user\config\startup.ijs|
+-----------------------------------------+
```

We see that `startup.ijs` has been loaded and its definitions are available.

```
   is_string 'hello'
1
```

This is the end of Chapter 26.

Chapter 27: Representations and Conversions

In this chapter we look at various transformations of functions and data.

27.1 Classes and Types

If we are transforming things into other things, it is useful to begin with functions which tell us what sort of thing we are dealing with.

27.1.1 Classes

Given an assignment, `name =: something`, then `something` is an expression denoting a noun or a verb or an adverb or a conjunction. That is, there are 4 classes to which `something` may belong.

There is a built-in verb `4!:0` which here we can call `class`.

```
class =: 4!:0
```

We can discover the class of `something` by applying `class` to the argument `<'name'`. For example,

`n =: 6`	`class < 'n'`
`6`	`0`

The result of `0` for the class of `n` means that `n` is a noun. The cases are:

```
0  noun

1  adverb

2  conjunction

3  verb
```

and two more cases: the string `'n'` is not a valid name, or `n` is valid as a name but no value is assigned to `n`.

```
_2  invalid

_1  unassigned
```

For example:

C =: &	class <'C'	class <'yup'	class <'1+2'
&	2	_1	_2

The argument of `class` identifies the object of interest by quoting its name to make a string, such as `'C'`.

Why is the argument not simply the object? Because, by the very purpose of the `class` function, the object may be a verb, noun, adverb or conjunction, and an adverb or conjunction cannot be supplied as argument to any other function.

Why not? Suppose the object of interest is the conjunction `C`. No

matter how `class` is defined, whether verb or adverb, any expression of the form `(class C)` or `(C class)` is a bident or a syntax error. In no case is function `class` applied to argument `C`. Hence the need to identify `C` by quoting its name.

27.1.2 Types

A noun may be an array of integers, or of floating-point numbers or of characters, and so on. The type of any array may be discovered by applying the built-in verb `3!:0`

```
type =: 3!:0
```

For example

`type 0.1`	`type 'abc'`
`8`	`2`

The result of `8` means floating-point and the result `2` means character. Possible cases for the result are (amongst others):

```
     1  boolean
     2  character  (that is, 8-bit characters)
     4  integer
     8  floating point
    16  complex
    32  boxed
    64  extended integer
   128  rational
 65536  symbol
131072  wide character (16-bit)
```

27.2 Execute

There is a built-in verb `".` (doublequote dot, called "Execute"). Its argument is a character-string representing a valid J expression, and the result is the value of that expression.

```
   ". '1+2'
3
```

The string can represent an assignment, and the assignment is executed:

". 'w =: 1 + 2'	w
3	3

If the string represents a verb or adverb or conjunction, the result is null, because Execute is itself a verb and therefore its results must be nouns. However we can successfully Execute assignments to get functions.

". '+'	". 'f =: +'	f
		+

27.3 On-Screen Representations

When an expression is entered at the keyboard, a value is computed and displayed on-screen. Here we look at how values are represented in on-screen displays. For example, if we define a function `foo`:

```
   foo =: +/ % #
```

and then view the definition of `foo`:

```
   foo
+-----+-+-+
|+-+-+|%|#|
||+|/|| | |
|+-+-+| | |
+-----+-+-+
```

we see on the screen some representation of `foo`. What we see depends on which of several options is currently in effect for representing functions on-screen.

By default the current option is the "boxed representation", so we see above `foo` depicted graphically as a structure of boxes. Other options are available, described below. To select and make current an option for representing functions on-screen, enter one of the following expressions:

```
(9!:3) 2  NB. boxed (default)

(9!:3) 5  NB. linear

(9!:3) 6  NB. parenthesized

(9!:3) 4  NB. tree

(9!:3) 1  NB. atomic
```

The current option remains in effect until we choose a different option.

27.3.1 Linear Representation

If we choose the the linear representation, and look at `foo` again:

```
   (9!:3) 5  NB. linear

   foo
+/ % #
```

we see `foo` in a form in which it could be typed in at the keyboard, that is, as an expression.

Notice that the linear form is equivalent to the original definition, but not necessarily textually identical: it tends to minimize parentheses.

```
   bar =: (+/) % #

   bar
+/ % #
```

Functions, that is, verbs, adverbs and conjunctions, are shown in the current representation. By contrast, nouns are always shown in the boxed representation, regardless of the current option. Even though linear is current, we see:

```
   noun =: 'abc';'pqr'

   noun
+---+---+
|abc|pqr|
+---+---+
```

27.3.2 Parenthesized

The parenthesized representation is like linear in showing a function as an expression. Unlike linear, the parenthesized form

helpfully adds parentheses to make the logical structure of the expression more evident.

```
   (9!:3) 6  NB. parenthesized

   zot =: f @: g @: h

   zot
(f@:g)@:h
```

27.3.3 Tree Representation

Tree representation is another way of displaying structure graphically:

```
   (9!:3) 4  NB. tree

   zot
             +- f
       +- @: -+- g
-- @: -+- h
```

27.3.4 Atomic Representation

See below

Before continuing, we return the current representation option to linear.

```
   (9!:3) 5
```

27.4 Representation Functions

Regardless of the current option for showing representations on-

screen, any desired representation may be generated as a noun by applying a suitable built-in verb.

If `y` is a name with an assigned value, then a representation of `y` is a noun produced by applying one of the following verbs to the argument `<'y'`

```
br =:  5!:2    NB. boxed
lr =:  5!:5    NB. linear
pr =:  5!:6    NB. parenthesized
tr =:  5!:4    NB. tree
ar =:  5!:1    NB. atomic
```

For example, the boxed and parenthesized forms of `zot` are shown by:

```
   br < 'zot'
+--------+--+-+
|+-+--+-+|@:|h|
||f|@:|g||  | |
|+-+--+-+|  | |
+--------+--+-+

   pr < 'zot'
(f@:g)@:h
```

We can get various representations of a noun, for example the boxed and the linear:

```
   br <'noun'
+---+---+
|abc|pqr|
+---+---+

   lr <'noun'
<;._1 ' abc pqr'
```

Representations produced by `5!:n` are themselves nouns. The linear form of verb `foo` is a character-string of length 6.

`foo`	`s =: lr <'foo'`	`$ s`
`+/ % #`	`+/ % #`	`6`

The 6 characters of `s` represent an expression denoting a verb. To capture the verb expressed by string `s`, we could prefix the string with characters to make an assignment, and Execute the assignment.

`s`	`$ s`	`a =: 'f =: ' , s`	`". a`	`f 1 2`
`+/ % #`	`6`	`f =: +/ % #`		`1.5`

27.4.1 Atomic Representation

We saw in Chapter 10 and Chapter 14, that it is useful to be able to form sequences of functions. By this we mean, not trains of verbs, but gerunds. A gerund, regarded as a sequence of verbs, can for example be indexed to find a verb applicable in a particular case of the argument.

To be indexable, a sequence must be an array, a noun. Thus we are interested in transforming a verb into a noun representing that verb, and vice versa. A gerund is a list of such nouns, containing atomic representations. The atomic representation is suitable for this purpose because it has an inverse. None of the other representation functions have true inverses.

The atomic representation of anything is a single box with inner structure. For an example, suppose that `h` is a verb defined as a hook. (A hook is about the simplest example of a verb with non-trivial structure.)

```
h =: + %
```

compare the boxed and the atomic representations of `h`

```
   br <'h'
+-+-+
|+|%|
+-+-+

   ar < 'h'
+---------+
|+-+-----+|
||2|+-+-+||
||  ||+|%|||
||  |+-+-+||
|+-+-----+|
+---------+
```

The inner structure is an encoding which allows the verb to be recovered from the noun efficiently without reparsing the original definition. It mirrors the internal form in which a definition is stored. It is NOT meant as yet another graphic display of structure.

The encoding is described in the Dictionary. We will not go into much detail here. Very briefly, in this example we see that `h` is a hook (because `2` is an encoding of "hook") where the first verb is `+` and the second is `%`.

The next example shows that we can generate atomic representations of a noun, a verb, an adverb or a conjunction.

```
N =: 6
```

```
V =: h
A =: /
C =: &
```

ar <'N'	ar <'V'	ar <'A'	ar <'C'
+-----+ |+-+-+| ||0|6|| |+-+-+| +-----+	+-+ |h| +-+	+-+ |/| +-+	+-+ |&| +-+

27.4.2 Inverse of Atomic Representation

The inverse of representation is sometimes called "abstraction", (in the sense that for example a number is an abstract mathematical object represented by a numeral.) The inverse of atomic representation is `5!:0` which we can call `ab`.

```
ab =: 5!:0
```

`ab` is an adverb, because it must be able to generate any of noun, verb, adverb or conjunction. For example, we see that the abstraction of the atomic representation of `h` is equal to `h`

h	r =: ar < 'h'	r ab
+ %	+---------+ |+-+-----+| ||2|+-+-+|| || ||+|%||| || |+-+-+|| |+-+-----+| +---------+	+ %

and similarly for an argument of any type. For example for noun `N` or conjunction `C`

N	rN=: ar <'N'	rN ab	C	(ar <'C') ab
6	+-----+ |+-+-+| ||0|6|| |+-+-+| +-----+	6	&	&

27.4.3 Execute Revisited

Here is another example of the use of atomic representations. Recall that Execute evaluates strings expressing nouns but not verbs. Since Execute is itself a verb it cannot deliver verbs as its result.

". '1+2'	". '+'
3	

To evaluate strings expressing values of any class we can define an adverb `eval` say, which delivers its result by abstracting an atomic representation of it.

```
   eval =: 1 : 0
". 'w =. ' , u
(ar < 'w') ab
)
```

`'1+2' eval`	`mean =: '+/ % #' eval`	`mean 1 2`
`3`	`+/ % #`	`1.5`

27.4.4 The Tie Conjunction Revisited

Recall from Chapter 14 that we form gerunds with the Tie conjunction `` ` ``. Its arguments can be two verbs.

```
G =: (+ %) ` h
```

Its result is a list of atomic representations. To demonstrate, we choose one, say the first in the list, and abstract the verb.

`G`	`r =: 0 { G`	`r ab`

`+---------+-+`
`|+-+-----+|h|`
`||2|+-+-+|| |`
`|| ||+|%||| |`
`|| |+-+-+|| |`
`|+-+-----+| |`
`+---------+-+`
	`+---------+`
	`|+-+-----+|`
	`||2|+-+-+||`
	`|| ||+|%|||`
	`|| |+-+-+||`
	`|+-+-----+|`
	`+---------+`
		`+ %`

The example shows that Tie can take arguments of expressions denoting verbs. By contrast, the atomic representation function (`ar` or `5!:1`) must take a boxed name to identify its argument.

Here is a conjunction `T` which, like Tie, can take verbs (not names) as arguments and produces atomic representations.

```
T =: 2 : '(ar <''u.'') , (ar <''v.'')'
```

(+ %) T h	(+ %) ` h
+---------+-+ |+-+-----+|h| ||2|+-+-+|| | || ||+|%||| | || |+-+-+|| | |+-+-----+| | +---------+-+	+---------+-+ |+-+-----+|h| ||2|+-+-+|| | || ||+|%||| | || |+-+-+|| | |+-+-----+| | +---------+-+

27.5 Conversions for Binary Data

Binary data is, briefly, values represented compactly as character strings. Here we look at functions for converting between values in J arrays and binary data, with a view to handling files with binary data. Data files will be covered in Chapter 28 .

In the following, a 32-bit PC is assumed, so it is assumed that a character occupies one byte and a floating point number occupies 8.

A J array, of floating-point numbers for example, is stored in the memory of the computer. Storage is required to hold information about the type, rank and shape of the array, together with storage for each number in the array. Each floating-point number in the array needs 8 bytes of storage.

There are built-in functions to convert a floating-point number to a character-string of length 8, and vice versa.

```
   cf8 =:   2 & (3!:5)   NB. float to 8 chars
   c8f =:  _2 & (3!:5)   NB. 8 chars to float
```

In the following example, we see that the number `n` is floating-point, `n` is converted to give the string `s` which is of length 8, and `s` is converted back to give a floating-point number equal to `n`.

n =: 0.1	$ s =: cf8 n	c8f s
0.1	8	0.1

Characters in the result `s` are mostly non-printable. We can inspect the characters by locating them in the ASCII character-set:

```
   a. i. s
154 153 153 153 153 153 185 63
```

Now consider converting arrays of numbers. A list of numbers is converted to a single string, and vice versa::

a =: 0.1 0.1	$ s =: cf8 a	c8f s
0.1 0.1	16	0.1 0.1

The monadic rank of `cf8` is infinite: `cf8` applies just once to its whole argument.

```
   RANKS =: 1 : 'u b. 0'
   cf8 RANKS
_ _ _
```

but the argument must be a scalar or list, or else an error results.

b =: 2 2 $ a	$ w =: cf8 b	$ w =: cf8"1 b
0.1 0.1 0.1 0.1	error	2 16

A floating-point number is convertible to 8 characters. There is an option to convert a float to and from a shorter 4-character string, sacrificing precision for economy of storage.

```
cf4 =:  1 & (3!:5)   NB. float to 4 chars
c4f =: _1 & (3!:5)   NB. 4 chars to float
```

As we might expect, converting a float to 4 characters and back again can introduce a small error.

```
p =: 3.14159265358979323
```

p	$ z =: cf4 p	q =: c4f z	p - q
3.14159	4	3.14159	_8.74228e_8

A J integer needs 4 bytes of storage. There are functions to convert between J integers and 4-character strings.

```
ci4 =:  2 & (3!:4)  NB. integer to 4 char
c4i =: _2 & (3!:4)  NB. 4 char  to integer
```

i =: 1 _100	$ s =: ci4 i	c4i s
1 _100	8	1 _100

We see that the length of `s` is 8 because `s` represents two integers.

Suppose `k` is an integer and `c` is the conversion of `k` to 4 characters.

`k =: 256+65`	`$ c =: ci4 k`
`321`	`4`

Since characters in `c` are mostly non-printable, we inspect them by viewing their locations in the ASCII alphabet. We see that the characters are the base-256 digits in the value of `k`, stored in `c` in the order least-significant first (on a PC)..

`k`	`a. i. c`	`256 256 256 256 #: k`
`321`	`65 1 0 0`	`0 0 1 65`

Integers in the range `_32768` to `32767` can be converted to 2-character strings and vice versa.

```
ci2 =:  1 & (3!:4)  NB. integer to 2 char
c2i =: _1 & (3!:4)  NB. 2 char  to int
```

`i`	`$ s =: ci2 i`	`c2i s`
`1 _100`	`4`	`1 _100`

Integers in the range `0` to `65535` can be converted to 2-character strings and vice versa. Such strings are described as "16bit unsigned".

```
   ui2 =: ci2          NB. integer to 2-char,
unsigned
   u2i =: 0 & (3!:4)   NB. 2 char  to integer,
unsigned
```

m =: 65535	$ s =: ui2 m	u2i s
65535	2	65535

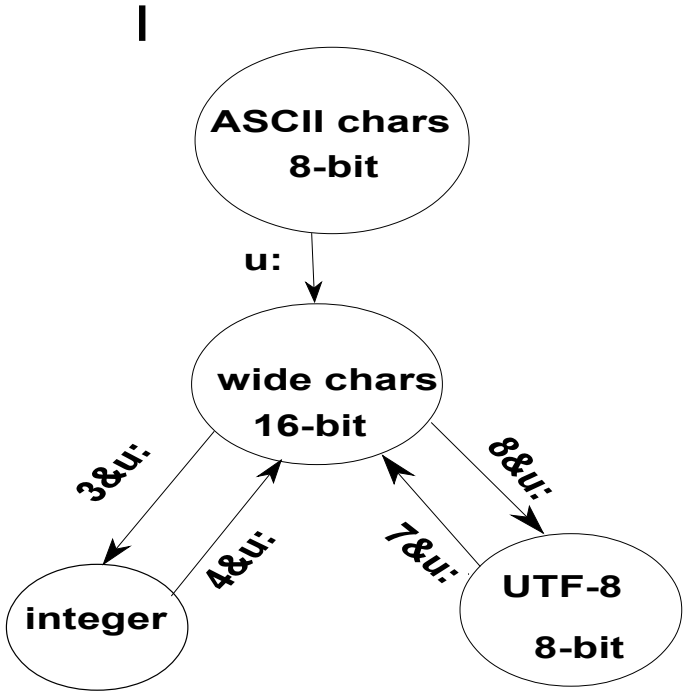
27.6 Unicode

In this section we look at J support for Unicode.

There are three kinds of character data in J.

- Ordinary character data we have seen already as 8-bit ASCII
- 16-bit characters, called "wide characters" for Unicode.
- Sequences of 8-bit characters, which represent Unicode characters, for the purpose of writing Unicode in files. This representation is called the UTF-8 encoding.

The following diagram shows the J functions available for converting character data from one kind to another. The functions are members of the `u:` family.



We have seen that J supports character data. For example

```
   C =: 'this is a string'
```

The built-in verb `3 !: 0` shows the type of a data value.

```
   3!:0  C
2
```

The result of `2` indicates that the data type of `C` is 8-bit characters, called "char".

J also provides another data type with 16-bit characters, called "wchar" ("wide character"). The built-in function monadic `u:` converts char data to wchar.

```
   ] W =: u: C
this is a string
```

wchar data is displayed as before, but its data-type is shown as 131072

```
   3!:0 W
131072
```

A 16-bit wchar character can be one of the many characters in the Unicode standard. The built-in function `4&u:` produces a wchar character specified by the argument, which is an integer in the range 0-65536, called a Unicode "code point".

A code point is often given as 4 hex digits. For example, the code point for the Greek letter alpha is hex 03b1 which we can write as `16b03b1`

```
   ] alpha =: 4&u: 16b03b1
α
```

alpha is a wchar:

```
   3!:0  alpha
131072
```

We can build a wchar string including alpha :

```
] U= (u: 'the Greek letter alpha looks like this:  '),alpha
the Greek letter alpha looks like this:  α
```

Suppose now that our wchar data `U` is to be exported, say by writing it to a data file . We will need to encode our 16-bit wchar data as a sequence of 8-bit bytes, according to some recognised standard encoding scheme. The UTF-8 standard is suitable.

The built-in function `8&u:` produces a character string which is a UTF-8 encoding of wchar data

```
   ] Z =: 8&u: U
the Greek letter alpha looks like this:  α
```

We see that `Z` is of data type 2, (that is 8-bit char) and that the number of bytes in `Z` is one more than the number of characters in `U`, because alpha is encoded as two bytes.

3!:0 Z	# U	# Z
2	42	43

The inverse of `8&u:` is the built-in function `7&u:` which produces wchar characters from a UTF-8 string.

```
   ] A =: 7&u: Z
the Greek letter alpha looks like this:  α
```

We can view the Unicode code-points of the letters in `A`. The built-in function `3&u:` produces code-point integers from wchar data. If we look at the last few characters of A, we see as we expect that the code-point integer of alpha is decimal 945, that is, hex 03b1.

```
   ] L =:  _6 {. A      NB. last few of A
is:  α
```

```
   3 & u: L
105 115 58 32 32 945
```

This is the end of Chapter 27

Chapter 28: Data Files

The subject of file-handling in general, and how data is organized in files, is a major topic in itself. In this chapter we will cover only a selection of the facilities available in J.

J functions to read files produce results in the form of character-strings, and similarly functions to write files take strings as arguments. Such a string can be the whole data content of a file when the available memory of the computer is sufficient.

Our approach here will be to look first at some J functions for input and output of strings. Then we look at a few examples of dealing with strings as representing data in various formats. Finally we look at mapped files as an alternative to conventional file-handling.

28.1 Reading and Writing Files

28.1.1 Built-in Verbs

In the following, a filename is a string which is valid as a filename for the operating-system of the computer where we are running J. For example:

```
   F =: 'c:\temp\demofile.xyz'        NB. a filename
```

The built-in verb `1!:2` writes data to a file. The right argument is a boxed filename. The left argument is a character-string, the data to be written. The effect is that the file is created if it does not already exist, and the data becomes the whole content of the file. The result is null.

```
   'some data' 1!:2 < F    NB. write to file F
```

The built-in verb `1!:1` reads data from a file. The right argument is a boxed filename. The result is a character-string, the data read.

```
   data =: 1!:1 < F     NB.  read from file F
```

data	$ data
some data	9

28.1.2 Screen and Keyboard As Files

Screen and keyboard can be treated as files, to provide a simple facility for user-interaction with a running program.

The expression `x (1!:2) 2` writes the value of `x` to "file 2", that is, to the screen. A verb to display to the screen can be written as

```
   display =: (1!:2) & 2
```

For example, here is a verb to display the stages in the computation of least-common-denominator by Euclid's algorithm.

```
   E =: 4 : 0
display x , y
if. y = 0 do. x else. (x | y) E x end.
)

   12 E 15
12 15
3 12
0 3
3 0
```

```
3
```

The value to be displayed by `(1!:2) &2` is not limited to strings: in the example above a list of numbers was displayed.

User-input can be requested from the keyboard by reading "file 1", that is, by evaluating `(1!:1) 1`. The result is a character-string containing the user's keystrokes. For example, a function for user-interaction might be:

```
   ui =: 3 : 0
display 'please type your name:'
n  =.  (1!:1) 1
display 'thank you ', n
''
)
```

and then after executing

```
        ui ''
```

a dialogue appears on the screen, like this:

```
please type your name:

Waldo

thank you Waldo
```

28.1.3 Library Verbs

There are a number of useful verbs for file-handling in the "standard library" (Chapter 26). Here is a brief summary of a selection:

`s fwrite F`	write string s to file F
`fread F`	read string from file F
`s fappend F`	append string s to file F
`fread F;B,L`	read slice from file F, starting at B, length L
`s fwrites F`	write text s to file F
`freads F`	read text from file F
`fexist F`	true if file F exists
`ferase F`	delete file F

From now on we will use these library verbs for our file-handling.

The library verb `fwrite` writes data to a file. The right argument is a filename. The left argument is a character-string, the data to be written. The effect is that the file is created if it does not already exist, and the data becomes the whole content of the file.

```
   'some data' fwrite F    NB. file write
9
```

The result shows the number of characters written. A result of `_1` shows an error: either the left argument is not a string or the right argument is not valid as a filename, or the specified file exists but is read-only.

```
   (3;4) fwrite F
_1
```

The library verb **fread** reads data from file. The argument is a filename and the result is a character-string.

z =: fread F	$z
some data	9

A result of _1 shows an error: the specified file does not exist, or is locked.

fread 'qwerty'	fexist 'qwerty'
_1	0

28.2 Large Files

For large files, the memory of the computer may not be sufficient to allow the file to be treated as a single string. We look at this case very briefly.

Write a file with some initial content:

```
   'abcdefgh' fwrite F
8
```

We can append some data to the file with library verb **fappend**.

```
   'MORE' fappend F
4
```

To see the effect of `fappend` (just for this demonstration, but not of course for a large file) we can read the whole file again :

```
   fread F
abcdefghMORE
```

We can read a selected slice of the file, say 8 bytes starting from byte `4`. In this case we use `fread` with a right argument of the form `filename;start,size`.

```
   start =: 4
   size  =: 8
   fread F ; start, size
efghMORE
```

28.3 Data Formats

We look now at a few examples of how data may be organized in a file, that is, represented by a string. Hence we look at converting between character strings, with various internal structures, and J variables.

We take it that files are read and written for the purpose of exchanging data between programs. Two such programs we can call "writer" and "reader". Questions which arise include:

1. Are writer and reader both to be J programs? If so, then there is a convenient J-only format, the "binary representation" covered below. If not, then we expect to

work from a programming-language-independent description of the data.

2. Are writer and reader to run on computers with the same architecture? If not, then even in the J-to-J situation, some finesse may be needed.

3. Is the data organized entirely as a repetition of some structure (for example, "fixed length records"). If so then we may usefully be able to treat it as one or more J arrays. If not, we may need some ad-hoc programming.

28.3.1 The Binary Representation for J-0nly Files

Suppose we aim to handle certain files only in J programs, so that we are free to choose any file format convenient for the J programmer. The "binary representation" is particularly convenient.

For any array **A**,

```
A =:  'Thurs'; 19 4 2001
```

the binary representation of **A** is a character string. There are built-in verbs to convert between arrays and binary representations of arrays.

```
arrbin  =: 3!:1   NB. array to binary rep.
binarr  =: 3!:2   NB. binary rep. to array
```

If **B** is the binary representation of **A**, we see that **B** is a character string, with a certain length.

```
A                    $ B =: arrbin A
+-----+---------+    88
|Thurs|19 4 2001|
+-----+---------+
```

We can write `B` to a file, read it back, and do the inverse conversion to recover the value of `A` :

```
B fwrite F    $ Z =: fread F    binarr Z
88            88                +-----+---------+
                                |Thurs|19 4 2001|
                                +-----+---------+
```

From J4.06 on, there are variations of the binary representation verbs above to allow for different machine architectures: see the Dictionary under `3!:1`.

28.3.2 Text Files

The expression `a.` (lower-case a dot) is a built-in noun, a character-string containing all 256 ASCII characters in sequence.

```
65 66 67 { a.    $ a.
ABC              256
```

In the ASCII character set, that is, in `a.`, the character at position 0 is the null, at position 10 is line-feed and at position 13 is

carriage return . In J, the names **CR** and **LF** are predefined in the standard profile to mean the carriage-return and linefeed characters.

```
   a. i. CR,LF
13 10
```

We saw **fread** and **fwrite** used for reading and writing character files. Text files are a special kind of character file, in that lines are delimited by **CR** and/or **LF** characters.

On some systems the convention is that lines of text are delimited by a single **LF** and on other systems a **CR,LF** pair is expected. Regardless of the system on which J is running, for J text variables, the convention is always followed of delimiting a line with single **LF** and no **CR**.

Here is an example of a text variable.

```
   t =: 0 : 0
There is physics
and there is
stamp-collecting.
)
```

Evidently it is a string (that is, a 1-dimensional character list) with 3 **LF** characters and no **CR** characters.

$ t	+/t=LF	+/t=CR
49	3	0

If we aim to write this text variable **t** to a text file, we must

choose between the single-`LF` or `CRLF` conventions. There are two useful library verbs, `fwrites` and `freads` to deal with this situation.

- Under Windows, `x fwrites y` writes text-variable `x` to file `y`, in the process converting each `LF` in `x` to a `CRLF` pair in `y`.
- Under Linux, `x fwrites y` writes text-variable `x` to file `y`, with no conversion.
- Under Windows or Linux `z =: freads y` reads file `y`, converting any `CRLF` pair in `y` to a single `LF` in text-variable `z`.

For convenience in dealing with a text variable such as `t`, we can cut it into lines. A verb for this purpose is `cut` (described more fully in Chapter 17).

```
   cut =: < ;. _2
```

`cut` produces a boxed list of lines, removing the `LF` at the end of each line.

```
   lines =: cut t
   lines
+---------------+------------+----------------+
|There is physics|and there is |stamp-collecting.|
+---------------+------------+----------------+
```

The inverse of `cut` we can call `uncut`. It restores the `LF` at the end of each box and then razes to make a string.

```
   uncut =: ; @: (,&LF &. >)
   uncut lines
There is physics
and there is
```

```
stamp-collecting.
```

28.3.3 Fixed Length Records with Binary Data

Suppose our data is in two J variables: a table `cnames`, of customer-names, and a list `amts` in customer order with for each customer an amount, a balance say.

`cnames =: 'Mr Rochester' ,: 'Jane'`	`,. amts =: _10000 3`
`Mr Rochester` `Jane`	`_10000` `3`

Now suppose the aim is to write this data to a file, formatted in 16-byte records. Each record is to have two fields: customer-name in 12 bytes followed by amount in 4 bytes, as a signed integer. Here is a possible approach.

The plan is to construct, from `cnames` and `amts`, an n-by-16 character table, to be called `records`. For this example, `n=2`, and `records` will look like this:

```
Mr Rochester####
Jane        ####
```

where `####` represents the 4 characters of an integer in binary form.

We build the `records` table by stitching together side by side an n-by-12 table for the customer names field, and an n-by-4 table for the amounts field.

For the customer-names field we already have `cnames` which is suitable, since it is 12 bytes wide:

```
   $ cnames
2 12
```

For the amounts field we convert `amts` to characters, using `ci4` from Chapter 27. The result is a single string, which is reshaped to be n-by-4.

```
   ci4 =:  2 & (3!:4)  NB. integer to 4 char

   amtsfield =: ((# amts) , 4) $ ci4 amts
```

Now we build the n-by-16 `records` table by stitching together side-by-side the two "field" tables:

```
   records =: cnames ,. amtsfield
```

To inspect `records`, here is a utility verb which shows a non-printing character as `#`

```
   inspect =: 3 : ('A=.a.{~32+i.96';'(A i.y)
{ A,''#''')
```

inspect records	$ records
Mr Rochester#### Jane ####	2 16

The outgoing string to be written to the file is the ravel of the records.

```
   (, records) fwrite F
32
```

The inverse of the process is to recover J variables from the file. We read the file to get the incoming string.

```
   instr =: fread F
```

Since the record-length is known to be 16, the number of records is

```
   NR =: (# instr) % 16
```

Reshape the incoming string to get the `records` table.

```
   inspect records =: (NR,16) $ instr
Mr Rochester####
Jane        ####
```

and extract the data. The customer-names are obtained directly, as columns 0-11 of `records`.

```
   cnames =: (i.12) {"1 records
```

For the amounts, we extract columns 12-15, ravel into a single string and convert to integers with `c4i`.

```
   c4i =: _2 & (3!:4)  NB. 4 char  to integer

   amts   =: c4i  , (12+i.4) {"1  records
```

cnames	,. amts
Mr Rochester Jane	_10000 3

28.4 Mapped Files

A file is said to be mapped when the file is temporarily incorporated into the virtual-address-translation mechanism of an executing program. The data in a mapped file appears to the J programmer directly as the value of a J variable - an array. Changes to the value of the variable are changes to the data in the file.

In such a case, we can say, for present purposes, that the file is mapped to the variable or, equivalently, that the variable is mapped to the file.

Mapped files offer the following advantages:

- Convenience. Data in a file is handled just like data in any J array. There is no reading or writing of the file.
- Persistent variables. A variable mapped to a file lives in the file, and can persist from one J session to another.

There are two cases. In the first case, any kind of existing file can be mapped to a variable. We take as given the structure of the data in the file, and then the J program must supply a description of the desired mapping. For example, a file with fixed-length records could be mapped to a character table.

In the second case, a file can be created in J in a special format (called "jmf") specifically for the purpose of mapping to a variable. In this case, the description is automatically derived from the variable and stored in the file along with the data. Thus a "jmf" file is self-describing.

We look first at creating jmf files, and then at mapping given files..

28.4.1 Library Script for Mapped Files

There is a library script, **jmf.ijs**, for handling mapped files. For present purposes it is simplest to download it directly from the J Application Library. Here is a link to jmf.ijs .

Assuming we have downloaded it into say, directory **C:\temp** for example, we can load it into our J session with:

```
   load 'c:\temp\jmf.ijs'
```

The script will load itself into the locale **jmf** .

28.4.2 jmf Files and Persistent Variables

Suppose we have constructed an array **v** with some valuable data, which from now on we aim to use and maintain over a number of J sessions. Perhaps **v** is valuable now, or perhaps it will become valuable over subsequent sessions as it is modified and added-to.

Our valuable data **v** can be an array of numbers, of characters, or of boxes. For a simple example we start with **v** as a table of numbers.

```
   ] V =:  2 2 $ 1 2 3 4
1 2
3 4
```

We can make a persistent variable from **v** as follows.

Step 1 is to estimate the size, in bytes, of a file required for the value of **v**. Since we expect that over time **v** may grow from its present size ultimately to, say, 64 KB, then our estimate **s** is

```
   S =: 64000
```

If in doubt, allow plenty. The size must be given as a positive integer (not a float) and therefore less than 2147483648 (2Gb) on a 32-bit machine.

Step 2 is to choose a file-name and, for convenience, define a variable **F** to hold the the file name as a string. For example:

```
   F =: 'c:\temp\persis.jmf'
```

Step 3 is to create file **F** as a jmf file, large enough to hold **S** bytes of data. For this purpose the utility function **createjmf** is available (in locale **jmf**) so we can write:

```
   createjmf_jmf_ F;S
```

(On your system, with a different version of J, you may see a response different from what is shown here.)

At this point, file **F** exists. If we inspect it we see its actual size is a little larger than **S**, to accommodate a header record which makes the file self-describing.

```
   fdir F
+----------+------------------+-----+---+------+
|persis.jmf|2012 12 16 8 37 22|64284|rw-|-----a|
+----------+------------------+-----+---+------+
```

The content of file `F` is initially set by `createjmf_jmf_` to represent a J value, in fact a zero-length list. The important point is that file `F` now contains a definite value.

Step 4 is to map the content of file `F` to a new variable, for which we choose the name `P`.

```
map_jmf_ 'P'; F
```

This statement means, in effect:

```
P =:  value-currently-in-file-F
```

and we can verify that `P` is now an empty list:

P	$ P
	0

Notice particularly that the effect of mapping file `F` to variable `P` is to assign the value in `F` to `P` and not the other way around. Hence we avoided mapping file `F` directly onto our valuable array `V` because `V` would be overwritten by the preset initial value in `F`, and lost.

Step 5 is to assign to `P` the desired value, that of `V`

```
P =: V
```

Variable `P` is now a persistent variable, since it is mapped to file `F`. We can amend `P`, for example by changing the value at row 0 column 1 to `99`.

P	P =: 99 (<0 1) } P
1 2 3 4	1 99 3 4

or by appending a new row:

```
   ] P =: P ,  0 0
1 99
3  4
0  0
```

Step 6 is needed before we finish the current session. We `unmap` variable `P`, to ensure file `F` is closed.

```
   unmap_jmf_ 'P'
0
```

The result of `0` indicates success. The variable `P` no longer exists:

P	$ P
error	$ P

To demonstrate that the value of `P` persists in file `F` we repeat the mapping, processing and unmapping in this or another session. The name `P` we chose for our persistent variable is only for this session. In another session, the persistent variable in file `F` can be mapped to any name.

This time we choose the name `Q` for the persistent variable. We map file `F` to `Q`:

```
   map_jmf_ 'Q' ; F

   Q
1 99
3  4
0  0
```

modify `Q`:

```
   ] Q =: Q , 7 8
1 99
3  4
0  0
7  8
```

and unmap `Q` to close file `F`.

```
   unmap_jmf_ 'Q'
0
```

28.4.3 Mapped Files are of Fixed Size

Recall that we created file `F` large enough for `S` bytes of data.

```
   S
64000
   fdir F
+----------+------------------+-----+---+------+
|persis.jmf|2012 12 16 8 37 22|64284|rw-|-----a|
+----------+------------------+-----+---+------+
```

The variable in file `F` is currently much smaller than this, and the unused trailing part of the file is filled with junk. However, if we

continue to modify `Q` by appending to it, we reach a limit, by filling the file, and encounter an error. To demonstrate, with a verb `fill` for the purpose:

```
   fill =: 3 : 0
try.   while. 1 do. Q =: Q , 99 99 end.
catch. 'full'
end.
)

   map_jmf_ 'Q'; F
   fill ''
full
```

The amount of data now in `Q` can be estimated as 4 bytes per integer (since `Q` is integer) multiplied by the number of integers, that is, altogether `4 * */$ Q`. This result for the final size of `Q` accords with our original size estimate `S`.

`4 * */ $ Q`	`S`
`64000`	`64000`

```
   unmap_jmf_ 'Q'
0
```

28.4.4 Given Files

Now we look at mapping ordinary data files (that is, files other than the special jmf-format files we considered above).

The way the data is laid out in the file we take as given, and our task is specify how this layout is to be represented by the type, rank and shape of a J variable, that is, to specify a suitable mapping.

For example, suppose we aim to read a given file `G` with its data laid out in fixed-length records, each record being 8 characters. Suppose file `G` was originally created by, say:

```
        G =: 'c:\temp\data.xyz'

   'ABCD0001EFGH0002IJKL0003MNOP0004' fwrite G
32
```

The next step is to decide what kind of a variable will be suitable for mapping the data in file `G`. We decide on an n-by-8 character table. The number of rows, `n`, will be determined by the amount of data in the file, so we do not specify `n` in advance.

It is convenient to start with a small example of an n-by-8 character table, which we call a prototype. The choice of `n` is unimportant.

```
   prototype =: 1 8 $ 'a'
```

Now the mapping can be defined by:

```
   ] mapping =: ((3!:0) ; (}. @: $)) prototype
+-+-+
|2|8|
+-+-+
```

We see that `mapping` is a boxed list. The first item is the data-type. Here `2`, meaning "character", is produced by `3!:0 prototype`. The

second item is the trailing dimensions (that is, all but the first) of the prototype. Here `8` is all but the first of `1 8`, produced by `(}.@:$) prototype`. Thus `mapping` expresses or encodes "n-by-8 characters".

Now `mapping` is supplied as left argument to (dyadic) `map_jmf_`. We map file `G` onto a variable for which we choose the name `W` thus:

```
   mapping map_jmf_ 'W'; G
```

We see that `W` is now a variable. Its value is the data in the file.

W	$ W
ABCD0001 EFGH0002 IJKL0003 MNOP0004	4 8

We can amend the data in the ususal way:

```
   ] W =: 'IJKL9999' 2 } W
ABCD0001
EFGH0002
IJKL9999
MNOP0004
```

What we cannot do is add another row to the data, because all the space in file `G` is occupied by the data we already have.

W	W =: W , 'WXYZ0000'
ABCD0001 EFGH0002 IJKL9999 MNOP0004	error

We close file G by unmapping variable W:

```
   unmap_jmf_ 'W'
0
```

28.4.5 Mapped Variables Are Special

Mapping files to variables offers the programmer significant advantages in functionality and convenience.

The price to be paid for these advantages is that there are some considerations applying to mapped variables which do not apply to ordinary variables. The programmer needs to be aware of, and to manage, these considerations. This is our topic in this section and the next.

If A is an ordinary variable, not mapped, then in the assignment B=: A the value of A is in effect copied to B. A subsequent change to A does not affect the value of B.

A =: 1	B =: A	B	A =: 2	B
1	1	1	2	1

By contrast, consider a variable mapped to a file. If the file is very large, there may not be enough space for another copy of the value. Hence copying is to be avoided.

Compare the previous example with the case when `A` is a mapped variable.

```
map_jmf_ 'A';F
```

A =: 1	B =: A	B	A =: 2	B
1	1	1	2	2

We see that `B` changes with changes to `A`. In effect `B =: A` means that `B` is another name for `A`, not a copy of the value of `A`. That is, both `A` and `B` refer to the same thing - the value in the file.

Hence it is also the case that `A` changes with changes to `B`.

A	B =: 7	A
2	7	7

Consider now an explicit verb applied to a mapped variable. Here `y` becomes another name for the data in the file. Hence assignment to `y` (even a local assignment) may cause an unintended change the mapped variable in the file. For example

```
foo =: 3 : ' 3 * y =. y + 1'
```

foo 2	A	foo A	A
9	7	24	8

28.4.6 Unmapping Revisited

The current status of mapped files and variables is maintained by the J system in a "mapping table". The mapping table can be displayed by entering the expression `showmap_jmf_ ''` but for present purposes here is a utility function to display only selected columns.

```
   status =: 0 1 9 & {"1  @: showmap_jmf_
   status ''
+-------+----------------+----+
|name   |fn              |refs|
+-------+----------------+----+
|A_base_|c:\temp\persis.jmf|3   |
+-------+----------------+----+
```

We see that currently variable `A` in locale `base` is mapped to file `F` (persis.jmf).

Under "refs", the value `3` means that the data in file `F` is the target of 3 references. One of these is variable `A`, a second is the variable `B` (which we know to be another name for `A`) and the third is for the system itself.

Variables `A` and `B` are both in existence:

A	B
8	8

For the sake of simplicity, a recommended procedure for closing the file is first to erase all variables such as `B` which are alternative names for the originally-mapped variable `A`

```
   erase <'B'
1
```

The status shows the number of references is reduced.

```
   status ''
+-------+------------------+----+
|name   |fn                |refs|
+-------+------------------+----+
|A_base_|c:\temp\persis.jmf|2   |
+-------+------------------+----+
```

Now we can unmap `A`.

```
   unmap_jmf_ 'A'
0
```

The result of `0` means the file is closed and `A` erased. The status table shows no entries, that is, that no files are mapped.

```
   status ''
+----+--+----+
|name|fn|refs|
+----+--+----+
```

Let us recreate the situation in which **A** is mapped to **F** and **B** is another name for **A**, so there are 3 references to (the data in) file **F**.

```
   map_jmf_ 'A'; F
   B =: A
   status ''
+-------+------------------+----+
|name   |fn                |refs|
+-------+------------------+----+
|A_base_|c:\temp\persis.jmf|3   |
+-------+------------------+----+
```

What happens if we erase all the variables referring to **F** ?

```
   erase 'A';'B'
1 1
   status ''
+-------+------------------+----+
|name   |fn                |refs|
+-------+------------------+----+
|A_base_|c:\temp\persis.jmf|1   |
+-------+------------------+----+
```

We see there is still a single reference, under the name **A** even though there is no variable **A**. This single reference reflects the fact

that file **F** is not yet unmapped.

Thus when we said earlier that file **F** gets mapped to variable **A**, it would be more accurate to say that file **F** gets mapped to the name **A**, and a variable of that name is created. Even though the variable is subsequently erased, the name **A** still identifies the mapped file, and can be used as an argument to **unmap**.

```
   unmap_jmf_ 'A'
0
   status ''
+----+--+----+
|name|fn|refs|
+----+--+----+
```

For more information, see the "Mapped Files" lab.

This is the end of Chapter 28

Chapter 29: Error Handling

The plan for this chapter is to look at some of the J facilities for finding and dealing with programming errors. It is beyond the scope of this chapter to consider debugging strategies, but (in my view) the use of assertions is much to be recommended. We look at:

- Assertions
- Continuing after failure
- Suspended execution
- Programmed error-handling

29.1 Assertions

A program can be made self-checking to some degree. Here is an example of a verb which computes the price of an area of carpet, given a list of three numbers: price per unit area, length and width.

```
   carpet =: 3 : 0
*/ y
)

   carpet 2 3 4
24
```

Assume for the sake of example that the computation `*/y` is large and problematic, and we want some assurance that the result is correct. We can at least check that the result is reasonable; we

expect the price of a carpet to lie between, say, $10 and $10,000.

We can redefine `carpet`, asserting that the result `p` must be between 10 and 10000:

```
   carpet =: 3 : 0
p =. */y
assert. p >: 10
assert. p <: 10000
p
)
```

If an assertion is evaluated as true (or "all true") there is no other effect, and the computation proceeds.

```
   carpet 2 3 4
24
```

If an assertion is evaluated as false, the computation is terminated and an indication given:

```
   carpet 0 3 4
|assertion failure: carpet
|   p>:10
```

Assertions can only be made inside explicit definitions, because `assert.` is a "control word", that is, an element of syntax, not a function.

It always a matter for judgement as to where an assertion can usefully be placed, and what can usefully be asserted. Assertions are best kept as simple as possible, since it is highly undesirable to make an error in an assertion itself.

It is often useful to make assertions which check the correctness of

arguments of functions. For example, we could assert that, for `carpet` the argument `y` must be a list of 3 strictly positive numbers.

The order of assertions may be important. For example, we should check that we have numbers before checking the values of those numbers. The type of a noun is given by `3!:0`; here we want integers (type=4) or reals (type=8).

```
   carpet =: 3 : 0

assert. (3!:0 y) e. 4 8   NB. numeric
assert. 1 = # $ y         NB. a list (rank = 1)
assert. 3 = # y           NB. of 3  items
assert. *. / y > 0        NB. all positive

p =. */y

assert. p >: 10
assert. p <: 10000

p
)

   carpet 2 3 4
24

   carpet 'hello'
|assertion failure: carpet
|   (3!:0 y)e.4 8
```

29.1.1 Assertions and the Tacit Style

Assertions are good for correctness. The tacit style is good for crispness and clarity.

The two are not readily combined, however. Evidently the natural place for an assertion is as a line in an explicit definition. By contrast, a tacit definition offers no place for an assertion.

What would it take to add assertions to a set of purely tacit definitions? Just to be able to make assertions about the arguments of functions would be a lot better than nothing. Here is a possibility.

Suppose we have an example of a purely tacit definition,

```
sq =: *:
```

and we wish to assert that any argument to `sq` must be a number, that is, it must satisfy the predicate:

```
is_number =: 4 8 16 128 e. ~ (3 !: 0)
```

Now our aim is to redefine `sq`, while making use of the previous definition of `sq`. Convenient for this purpose is a conjunction `ASSERTING`, which is defined below.

We can write

```
sq =: sq ASSERTING is_number
```

and we see:

```
   sq
3 : 0
assert. (is_number) y
(*:) y
)
   sq 3
9
   sq 'abc'
|assertion failure: sq
|   (is_number)y
```

The definition of **ASSERTING** is:

```
   ASSERTING
2 : 0
    U =. 5!:5  < 'u'
    if. (< U) e. nl 3 do. U =. 5!:5 < U end.
    V =. 5!:5  < 'v'
    z =: 'assert. (', V , ') y', LF
    z =. z , '(', U,  ') y'
    3 : z
)
```

The ASSERTING conjunction is written in this string-building style so that its result can be easily inspected. We can see that the new **sq** combines the predicate **is_number** with the value (not the name!) of the old **sq**. Finally, note that ASSERTING as here defined is good only for monadic verbs.

29.1.2 Enabling and Disabling Assertions

When we are confident of correctness, we can consider removing assertions from a program, particularly if performance is an issue. Another possibility is to leave the assertions in place, but to disable them. In this case, asserted expressions are not evaluated, and

assertions always succeed. There is a built-in function `9!:35` to enable or disable assertions. For example:

```
   (9!:35) 0      NB. disable assertions
   carpet 0 3 4   NB. an error
0

   (9!:35) 1      NB. enable assertions
   carpet 0 3 4   NB. an error
|assertion failure: carpet
|   */y>0
```

The built-in function `9!:34` tests whether assertions are enabled. Currently they are:

```
   9!:34 ''   NB. check that assertions are enabled
1
```

29.2 Continuing after Failure

There are several ways to continue after a failure.

29.2.1 Nonstop Script

In testing a program, it may be useful to write a script for a series of tests. Here is an example of a test-script.

```
   (0 : 0) (1!:2) <'test.ijs'   NB. create test-
script

NB. test 1
carpet 10  0 30
```

```
NB. test 2
carpet 10 20 30
)
```

A test may give the wrong result, or it may fail altogether, that is, it may be terminated by the system. We can force the script to continue even though a test fails, by executing the script with the built-in verb `0!:10` or `0!:11`

```
   0!:11 <'test.ijs'            NB. execute test-
script

   NB. test 1
   carpet 10  0 30
|assertion failure: carpet
|   */y>0

   NB. test 2
   carpet 10 20 30
6000
```

29.2.2 Try and Catch Control Structure

Here is an example of a verb which translates English words to French using word-lists.

```
   English =: 'one'; 'two';  'three'
   French  =: 'un';  'deux'; 'trois'

   ef =:  3 : '> (English i. < y) { French'
```

A word not in the list will produce an error.

ef 'two'	ef 'seven'
deux	error

This error can be handled with the `try. catch. end.` control structure. (Chapter 12 introduces control structures)

```
   EF =: 3 : 'try. ef y catch. ''don''''t know'' end.'
```

EF 'two'	EF 'seven'
deux	don't know

The scheme is that

```
try. B1 catch. B2 end.
```

means: execute block B1. If and only if B1 fails, execute block B2.

29.2.3 Adverse Conjunction

A tacit version of the last example can be written with the "Adverse" conjunction `::` (colon colon).

```
   TEF =: ef :: ('don''t know' " _)
```

TEF 'two'	TEF 'seven'
deux	don't know

Notice that the left and right arguments of `::` are both verbs. The scheme is:

```
(f :: g) y
```

means: evaluate `f y`. If and only if `f y` fails, evaluate `g y`

29.3 Suspended Execution

Suppose we have, as an example of program to be debugged, a verb `main` which uses a supporting verb `plus`

```
   main  =: 3 : 0
k =. 'hello'
z =. y plus k
'result is'; z
)

   plus =: +
```

Clearly there is an error in `main`: the string `k` is inconsistent with the numeric argument expected by `plus`.

If we type, for example, `main 1` at the keyboard, then when the error is detected the program terminates, an error-report is displayed and the user is prompted for input from the keyboard.

```
   main 1
|domain error: plus
|   z=.y     plus k
```

To gather more information about the cause of the error, we can arrange that the program can be suspended rather than terminated when control returns to the keyboard. To enable suspension we use the command `(13!:0) 1` before running `main` again.

```
   (13!:0) 1
```

Now when `main` is re-run, we see a slightly different error message

```
   main 1
|domain error: plus
|plus[:0]
```

At this point execution is suspended. In the suspended state, expressions can be typed in and evaluated. Notice that the prompt is 6 spaces (rather than the usual 3) to identify the suspended state.

```
      1+1
2
```

We can view the current state of the computation, by entering at the keyboard this expression, to show (selected columns of) what is called the "execution stack".

```
      0 2 6 7 8 { " 1 (13!:13 '')
+----+-+---------+----------+-+
|plus|0|+-+-----+|          |*|
|    | ||1|hello||          | |
|    | |+-+-----+|          | |
+----+-+---------+----------+-+
|main|1|+-+      |+--+-----+| |
|    | ||1|      ||k |hello|| |
|    | |+-+      |+--+-----+| |
|    | |         ||y |1    || |
|    | |         |+--+-----+| |
+----+-+---------+----------+-+
```

The stack is a table, with a row for each function currently executing. We see that **`plus`** is the function in which the error was detected, and **`plus`** is called from **`main`**.

The stack has 9 columns, of which we selected only 5 for display (columns 0 2 6 7 8). The columns are:

0	Name of suspended function. Only named functions appear on the stack.
1	(not shown above) error-number or 0 if not in error
2	Line-number. **`plus`** is suspended at line 0 and **`main`** is at line 1
3	(not shown above) Name-class: 1 2 or 3 denoting adverb, conjunction or verb
4	(not shown above) Linear representation of suspended function
5	(not shown above) name of script from which definitions were loaded
6	Values of arguments. **`plus`** was called with arguments **`1`** and **`'hello'`**
7	Names and values of local variables. **`plus`** being a tacit verb has no local variables, while **`main`** has **`k`** and also **`y`**, since arguments of explicit functions are regarded as local variables.
8	An asterisk, or a blank. **`plus`** is asterisked to show it is the function in which suspension was caused. Normally this the top function on the stack, (but not necessarily, as we will see below).

While in the suspended state we can inspect any global variables, by entering the names in the usual way. In this simple example there are none.

Finally, we can terminate the suspended execution, and escape from the suspended state, by entering the expression:

```
(13!:0) 1
```

29.4 Programmed Error Handling

By default, when suspension is enabled, and an error is encountered, the program suspends and awaits input from the keyboard.

We can arrange that instead of taking input from the keyboard, when an error is encountered, our own error-handling routine is automatically entered.

Suppose we decide to handle errors by doing the following:

- display the error message generated by the system
- display (selected columns of) the stack
- cut short the execution of the the suspended function, and cause it to return the value `'error'` instead of whatever it was intended to return.
- resume executing the program. (This may or may not result in a cascade of further errors.)

Here is a verb to perform this sequence of actions:

```
   handler =: 3 : 0
(1!:2&2) 13!:12 ''              NB. display error message
(1!:2&2) 0 2 6 7 8 {" 1 (13!:13 '') NB. display stack
13!:6 'error'                   NB. resume returning 'error'
)
```

The next step is to declare this verb as the error-handler. To do this we set an appropriate value for what is called the "latent expression". The latent expression is represented by a string which, if non-null, is executed automatically whenever the system is about to enter the suspended state. The latent expression can be queried and set with `13!:14` and `13!:15`. What is the current value of the latent expression?

```
   13!:14 ''
```

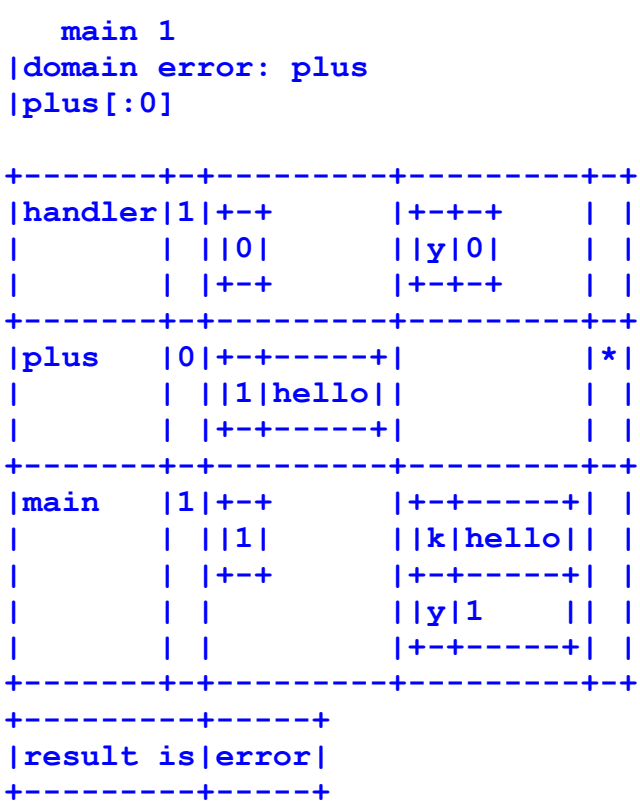
A null string. We set the latent expression to be a string, representing an expression meaning "execute the verb `handler`".

```
   13!:15 'handler 0'
```

Now we make sure suspension is enabled:

```
   (13!:0) 1 NB. enable suspension
```

and try a debugging run on `main`



We see that the topmost stack-frame is for `handler`, because we are in `handler` when the request to view the stack is issued. The suspended function is `plus`.

The display `result is error` demonstrates that `plus` returned the value (`'error'`) supplied by `handler`.

This is the end of Chapter 29.

Chapter 30: Sparse Arrays

30.1 Introduction

The sparse array facility of J allows a large array to be stored in the computer in a moderate amount of memory if many of the array's elements are all the same. In this case a value which occurs many times need be stored only once.

For an example, sparse representation might be considered for a connection matrix describing a network. In this chapter we will look at the J machinery for handling sparse arrays.

Suppose that `D` is a matrix with most of its elements the same:

```
   ] D =: 2 3 4 (2 2; 3 6; 4 4) } 16 16 $ 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

This array can be stored in a compact form, called a "sparse array", where only its non-zero elements occupy storage. An ordinary array which is not sparse may be called a "dense" array.

There is a built-in function, **$.** (dollar dot) to compute a sparse array from a dense.

```
   S =: $. D
```

For many purposes dense matrix **D** and sparse matrix **S** are equivalent: **S** matches **D**, and therefore it has the same dimensions, and gives the same result on indexing:

S -: D	($S) -: ($D)	((< 0 0){ S) -: (<0 0) { D
1	1	1

30.2 Sparse Array is Compact

Compared to matrix `D`, matrix `S` is economical in storage because the value which occurs many times in `D` is stored only once in `S`. This value is known as the "sparse element" of `S`, or the "zero" element of `S`. It happens to be `0` in the case of `S`, but need not be `0` always.

We can measure the size of the storage occupied by an array with the built-in `7!:5`. We see that the size of `S` (which the sparse form of `D`) is smaller than the size `D` itself:

7!:5 <'S'	7!:5 <'D'
384	2048

30.3 Inspecting A Sparse Array

There is a useful function `datatype` in the standard library. It shows the type of its argument.

datatype D	datatype S
integer	sparse integer

Recall that the built verb `3!:0` also gives the type of its argument. For a sparse array, the possible types reported by 3!:0 are

1024	sparse boolean
2048	sparse character
4096	sparse integer
8192	sparse floating point
16384	sparse complex
32768	sparse boxed

If we display `s` in the usual way , we see, not the familiar representation of a matrix, but instead a list of index-value pairs, one pair for each (in this example) non-zero element.

```
   s
2 2 | 2
3 6 | 3
4 4 | 4
```

This display does not show that the sparse element of `s` is in fact integer zero. To show this, we can extract the sparse element with the verb `3 & $.` .

`se =: 3 $. S`	`datatype se`
`0`	`integer`

If we now compute a new matrix from `s`

```
T =: S + 5
```

we see that `T` is sparse, and the sparse element of `T` is not zero but `5`

`T`	`3 $. T`

`2 2 | 7`
`3 6 | 8`
`4 4 | 9`
	`5`

Another way to view a sparse array is simply to convert it to dense with `0 & $.`

```
view =: 0 & $.
```

T	view T
2 2 | 7 3 6 | 8 4 4 | 9	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 8 5 5 5 5 5 5 5 5 5 5 5 5 5 9 5

30.4 Computing with Sparse Arrays

Computations with sparse arrays are pretty much the same as with dense arrays, except that they tend to produce sparse arrays as results. We saw this with `S+5` above. Here is another example. Summing over `T` produces a vector of column-sums which is sparse

```
   ] V =: +/ T
2 | 82
4 | 84
6 | 83
```

but the "zero" element of `V` is the sum of a column of "zero" elements of `T`

```
   3 $. V
80
```

At the time of writing, there are still some limitations on what can be done with sparse arrays compared with dense arrays. See the Dictionary under `$.` for more information.

30.5 Constructing A Sparse Array

At this point it will be helpful to define a few terms. First note that, according to context, the numerals `0` or `1` or `0.0` or `1.0` could be valid as boolean or integer or real. However in the absence of any context the J system takes them all to be in fact boolean.

datatype 0	datatype 1	datatype 0.0	datatype 1.0
boolean	boolean	boolean	boolean

It will be useful to define some values of unambiguous type.

INTEGERZERO =: 3 - 3	datatype INTEGERZERO
0	integer

INTEGERONE =: 3 - 2	datatype INTEGERONE
1	integer

REALZERO =: 0.0*0.1	datatype REALZERO
0	floating

REALONE =: ^ 0	datatype REALONE
1	floating

Returning now to sparse arrays, the recommended method of constructing them is to begin by making an empty array of the required shape and type, but with no actual data.

An empty array is built by evaluating the expression

```
1 $. shape;axes;zero
```

where

- **shape** specifies the dimensions
- **axes** specifies which of those dimensions will be sparse, as a list of axis-numbers. For example, with 2 dimensions both sparse the list would be **0 1**

  So far, in the examples of sparse arrays, all axes have been sparse but we will see below mixed sparse and dense axes.
- **zero** specifies the value of the "zero" element, and hence the type of the array as a whole. An unambiguous value is evidently needed.

If **zero** is omitted the default is **REALZERO**. If both **axes** and **zero** are omitted, the default is all axes sparse and **REALZERO**.

So to build a 6 by 6 matrix, sparse in all dimensions (that is, on axis 0 and axis 1), of type integer with "zero" element of 0 we can write:

```
U =: 1 $. 6 6 ; 0 1; INTEGERZERO
```

At this point, **U** is empty, that is, all "zero", so displays as nothing:

```
U
```

Populate it by inserting a few non-zero elements into it

```
U =: 4 5 6 7 ( 0 0 ; 1 1; 2 2; 3 3) } U
```

and check that **U** is what we expect by viewing it:

```
view U
```

```
4 0 0 0 0 0
0 5 0 0 0 0
0 0 6 0 0 0
0 0 0 7 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

30.6 Sparse and Dense Axes

An array may be sparse on some axes and dense on others. In the following example `W` is sparse on its first axis and dense on its second, because its list of sparse axes is just `0`

```
   saw   =: ,0   NB. sparse axes for W

   W =: 1 $. 3 5; saw ; INTEGERZERO

   W =: 4 5 6 (0 1; 0 2; 1 3) } W
```

It looks as expected:

```
   view W
0 4 5 0 0
0 0 0 6 0
0 0 0 0 0
```

but we see that it is stored as two dense rows only:

```
   W
0 | 0 4 5 0 0
1 | 0 0 0 6 0
```

Compare with an array sparse on second axis axis only, because its list of sparse axes is `1`

```
   saz=: ,1   NB. sparse axes for Z
   Z =: 1 $. 3 5; saz; INTEGERZERO
   Z =: 4 5 6 (0 1; 0 2; 1 3) } Z
```

Z looks just like `w`

```
   view Z
0 4 5 0 0
0 0 0 6 0
0 0 0 0 0
```

but we see it is stored as three dense colums.

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

30.7 Deconstructing a Sparse Array

As we noted above, if we display `U` itself, we see, not the familiar representation of a matrix, but instead a list of index-value pairs, one pair for each non-zero element.

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

We can extract the index from each pair to get what is called the index-matrix of `U`. This is an ordinary dense array

```
   4 $. U
0 0
1 1
2 2
3 3
```

To extract the value from each pair

```
    5 $. U
4 5 6 7
```

As we noted above, `0 & $.` will produce a dense array from a sparse:

```
   0 $. U
4 0 0 0 0 0
0 5 0 0 0 0
0 0 6 0 0 0
0 0 0 7 0 0
```

```
0 0 0 0 0 0
0 0 0 0 0 0
```

30.8 Sparse Array From Relation

Next we look at representing data as a sparse array as an alternative to representing data as a relation (that is, a table).

The point is that the sparse array may be more convenient than the relation for some computations with the data. Thus we are interested in converting between sparse arrays and relations.

For example, suppose that a given relation `R` represents sales of various commodities in various cities

```
   'Pa Qu Ro Sy' =: s: ' Paris Quebec Rome Sydney'
   'Ap Ba Ch Da' =: s: ' Apples Bananas Cherries
Damsons'


   R =: (". ;. _2) 0 : 0
Ap ; Pa; 99
Ap ; Qu ; 50
Ba ; Qu ; 10
Ch ; Ro ; 19
Da ; Sy ; 110
Da ; Pa ; 88
)
```

```
   R
+---------+-------+---+
|`Apples  |`Paris |99 |
+---------+-------+---+
|`Apples  |`Quebec|50 |
+---------+-------+---+
|`Bananas |`Quebec|10 |
+---------+-------+---+
|`Cherries|`Rome  |19 |
+---------+-------+---+
|`Damsons |`Sydney|110|
+---------+-------+---+
|`Damsons |`Paris |88 |
+---------+-------+---+
```

We can convert the relation `R` to a sparse array as follows.

Firstly, we need to establish the domain -the set of all possible values - of the first column. It can be computed from `R` :

```
   ] Fru =:  > ~. 0 { |: R
`Apples `Bananas `Cherries `Damsons
```

Similarly for the domain of the second column:

```
   ] Cit =: > ~. 1 { |: R
`Paris `Quebec `Rome `Sydney
```

Now the first column converted to indices into its domain:

```
   ] r =: Fru i. > 0 { |: R
0 0 1 2 3 3
```

Similarly for the second column:

```
   ] c =: Cit i. > 1 { |: R
0 1 1 2 3 0
```

and the values from the third

```
   ] v =: > 2 { |: R
99 50 10 19 110 88
```

Now we build an empty sparse array of dimensions `#Fru` by `#Cit` . By default the sparse axes will be 0 and 1 and the "zero" element will be `REALZERO` . The function `1&$.` produces the empty array.

```
   A =: (1 & $.) (#Fru) , (#Cit)
```

Insert the values by amending in the ordinary way:

```
   A =: v (<"1 r,.c) } A
```

and check we have what we expect:

```
   view A
99 50  0   0
 0 10  0   0
 0  0 19   0
88  0  0 110
```

To display `A` with labelling of rows and columns, the list of row-labels is `Fru` computed above, and the list of column-labels is `Cit` :

```
   (a:, <"0 Cit), (<"0 Fru) ,. (<"0 view A)
+---------+------+-------+-----+-------+
|         |`Paris|`Quebec|`Rome|`Sydney|
+---------+------+-------+-----+-------+
|`Apples  |99    |50     |0    |0      |
+---------+------+-------+-----+-------+
|`Bananas |0     |10     |0    |0      |
+---------+------+-------+-----+-------+
|`Cherries|0     |0      |19   |0      |
+---------+------+-------+-----+-------+
|`Damsons |88    |0      |0    |110    |
+---------+------+-------+-----+-------+
```

Now we have finished producing the sparse array from the original relation, so we can can compute with our data as an array.

For example, total value of sales for each city is given by:

```
   +/ A
0 | 187
1 |  60
2 |  19
3 | 110
```

This is sparse, so taking the usual view :

```
   view +/ A
187 60 19 110
```

30.9 Relation from Sparse Array

To complete the circle, we look next at how to produce a relation from a sparse array, `A` for example.

```
   A
0 0 |  99
0 1 |  50
1 1 |  10
2 2 |  19
3 0 |  88
3 3 | 110
```

The first step is to get the index-matrix for the non-zero elements.

```
   ] INDS =: 4 $. A
0 0
0 1
1 1
2 2
3 0
3 3
```

and next the values.

```
   ] VALS =: 5 $. A
99 50 10 19 88 110
```

The first column of the relation we produce by indexing the domain `Fru` which we computed above. The second column is produced similarly from `Cit`.

```
   ] c0 =: (0 { |: INDS) { Fru
`Apples `Apples `Bananas `Cherries `Damsons `Damsons
   ] c1 =: (1 { |: INDS) { Cit
`Paris `Quebec `Quebec `Rome `Paris `Sydney
```

So finally we see that the relation recovered from the sparse array is

```
   (<"0 c0) ,. (<"0 c1) ,. (<"0 VALS)
+---------+-------+---+
|`Apples  |`Paris |99 |
+---------+-------+---+
|`Apples  |`Quebec|50 |
+---------+-------+---+
|`Bananas |`Quebec|10 |
+---------+-------+---+
|`Cherries|`Rome  |19 |
+---------+-------+---+
|`Damsons |`Paris |88 |
+---------+-------+---+
|`Damsons |`Sydney|110|
+---------+-------+---+
```

This is the end of Chapter 30.

Chapter 31: Performance

This chapter is concerned with performance, that is, the time taken to perform a computation, and how to improve it.

There is one golden rule for achieving good performance in a J program. The rule is to try to apply verbs to as much data as possible at any one time. In other words, try to give to a verb arguments which are not scalars but vectors or, in general, arrays, so as to take maximum advantage of the fact that the built-in functions can take array arguments.

The rest of this chapter consists mostly of harping on this single point.

31.1 Measuring the Time Taken

There is a built-in verb `6!:2` . It takes as argument an expression (as a string) and returns the time (in seconds) to execute the expression. For example, given :

```
   mat =: ? 20 20 $ 100x  NB. a random matrix
```

The time in seconds to invert the matrix is given by:

```
   6!:2 '%. mat'
1.92381
```

If we time the same expression again, we see:

```
   6!:2 '%. mat'
1.85601
```

Evidently there is some uncertainty in this measurement. Averaging over several measurements is offered by the dyadic case of `6!:2` . However, for present purposes we will use monadic `6!:2` to give a rough and ready but adequate measurement.

31.2 The Performance Monitor

As well as `6!:2`, there is another useful instrument for measuring execution times, called the Performance Monitor. It shows how much time is spent in each line of, say, an explicit verb.

Here is an example with a main program and an auxiliary function. We are not interested in what it does, only in how it spends its time doing it..

```
   main =: 3 : 0
    m =. ? 10 10 $ 100x    NB. random matrix
    u =. =/ ~ i. 10        NB. unit matrix
    t =. matinv m          NB. inverted
    p =. m +/ . * t
    'OK'
)

   matinv =: 3 : 0
  assert. 2 = # $ y         NB. check y is square
  assert. =/ $ y
  %. y
)
```

We start the monitor:

```
   load 'jpm'
   start_jpm_ ''
357142
```

and then enter the expression to be analyzed

```
   main 0                    NB.  expression to be
analyzed
OK
```

To view the reports available: firstly , the main function:

```
   showdetail_jpm_ 'main'     NB. display
measurements
 Time (seconds)
+--------+--------+---+----------------+
|all     |here    |rep|main            |
+--------+--------+---+----------------+
|0.000007|0.000007|1  |monad           |
|0.000136|0.000136|1  |[0] m=.?10 10$100|
|0.000019|0.000019|1  |[1] u=.=/~i.10  |
|0.102103|0.000011|1  |[2] t=.matinv m |
|0.124305|0.124305|1  |[3] p=.m+/ .*t  |
|0.000024|0.000024|1  |[4] 'OK'        |
|0.226594|0.124502|1  |total monad     |
+--------+--------+---+----------------+
```

and we may wish to look at the auxiliary function:

```
   showdetail_jpm_ 'matinv'
 Time (seconds)
+--------+--------+---+----------------+
|all     |here    |rep|matinv          |
+--------+--------+---+----------------+
|0.000006|0.000006|1  |monad           |
|0.000008|0.000008|1  |[0] assert. 2=#$y|
|0.020002|0.020002|1  |[1] assert. =/$y |
|0.082076|0.082076|1  |[2] %.y         |
|0.102092|0.102092|1  |total monad     |
+--------+--------+---+----------------+
```

Evidently, `main` spends most of its time executing lines `2` and `3` . Notice that the time under "all" of line `2` is near enough equal to the time for line `2` "here", (that is, in `main`) plus the time for "total" of `matinv`

31.3 The Golden Rule: Example 1

Here is an example of a function which is clearly intended to take a scalar argument.

```
   collatz =: 3 : 'if. odd y do. 1 + 3 * y else. halve y end.'
       odd =: 2 & |
       halve  =: -:
```

With a vector agument it gives the wrong results

```
   collatz 2 3 4 5 6 7 8 9
1 1.5 2 2.5 3 3.5 4 4.5
```

So we need to specify the rank to force the argument to be scalar

```
   (collatz "0) 2 3 4 5 6 7 8 9
1 10 2 16 3 22 4 28
```

This is an opportunity for the Golden Rule, so here is a version designed for a vector argument:

```
   veco =: 3 : '(c*1+3*y) + (halve y) * (1-c =. odd
y)'
```

The results are the same:

```
   data =: 1 + i. 10000

   (collatz"0 data) -: (veco data)
1
```

but the vector version is about a hundred times faster:

```
   t1 =: 6!:2 e1 =: 'collatz"0 data '
   t2 =: 6!:2 e2 =: 'veco data '
   2 2 $ e1 ; t1; e2;t2
+---------------+-----------+
|collatz"0 data |0.0667271  |
+---------------+-----------+
|veco data      |0.000561943|
+---------------+-----------+
```

31.4 Golden Rule Example 2: Conway's "Life"

J. H. Conway's celebrated "Game of Life" needs no introduction. There is a version in J at Rosetta Code, reproduced here:

```
   pad=: 0,0,~0,.0,.~]
   life=: (_3 _3 (+/ e. 3+0,4&{)@,;._3 ])@pad
```

To provide a starting pattern, here is a function `rp` which generates an r-pentomino in a y-by-y boolean matrix.

```
   rp =: 3 : '4 4 |.  1 (0 1; 0 2; 1 0; 1 1; 2 1) }
(y,y) $ 0'

   ] M =: rp 8
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0
0 0 0 0 1 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0

   life  M
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0
0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0
```

We notice that the `life` verb contains `;._3` - it computes the count of neighbours of each cell separately, by working on the 3-by-3 neighbourhood of that cell.

By contrast here is a version which computes all the neighbours-counts at once, by shifting the whole plane to align each cell with

its neighbours.

```
   sh   =: |. !. 0
   E    =: 0 _1 & sh
   W    =: 0 1  & sh
   N    =: 1    & sh
   S    =: _1   & sh
   NS   =: N + S
   EW   =: E + W
   NeCo =: NS + (+ NS) @: EW                NB.
neighbour-count
   evol =: ((3 = ]) +. ([ *. 2 = ])) NeCo
```

The last line expresses the condition that (neighbour-count is 3) or ("alive" and count is 2). The shifting method `evol`, and the Rosetta method `life` give the same result

```
   (life M) -: (evol M)
1
```

However, the shifting method is faster:

```
   G =: rp 200   NB. a 200-by-200 grid

   t3 =: 6!:2 e3 =: 'r3 =: life ^: 100 G '
                    NB. 100 iterations of Rosetta method

   t4 =: 6!:2 e4 =: 'r4 =: evol ^: 100 G'
                          NB. and of shifting method
```

```
   2 2 $ e3;t3;e4;t4
+--------------------+---------+
|r3 =: life ^: 100 G |14.6997  |
+--------------------+---------+
|r4 =: evol ^: 100 G |0.0959352|
+--------------------+---------+
```

Checking for correctness again:

```
   r3 -: r4
1
```

31.5 Golden Rule Example 3: Join of Relations

31.5.1 Preliminaries

Recall from **Chapter 18** the author-title and title-subject relations. We will need test-data in the form of these relations in various sizes. It is useful to define a verb to generate test-data from random integers. (Integers are adequate as substitutes for symbols for present purposes.) The argument **y** is the number of different titles required.

```
   maketestdata =: 3 : 0
    T  =. i. y                                NB. titles domain
    A  =. i. <. 4 * y % 5                     NB. authors domain
    S  =. i. <. y % 2                         NB. subjects domain
    AT =. (? (#T) $ # A) ,. (? (#T) $ #T)     NB. AT relation
    TS =. (? (#T) $ # T) ,. (? (#T) $ #S)     NB. TS relation
    AT;TS
)


   'AT1 TS1' =: maketestdata 8        NB. small   test-data
   'AT2 TS2' =: maketestdata 1000     NB. medium
   'AT3 TS3' =: maketestdata 10000    NB. large
```

31.5.2 First Method

Recall also from **Chapter 18** a verb for the join of relations, which we will take as a starting-point for further comparisons. We can call this the "first method".

```
   VPAIR =: 2 : 0
    :
    z =.  0 0 $ ''
    for_at. x do. z=.z , |: v (#~"1  u) |: at , "1 y end.
    ~. z
)

   first  =: (1&{ =  2&{) VPAIR (0 3 & {)
```

AT1	TS1	AT1 first TS1
3 4	4 2	3 2
1 6	5 0	1 2
0 1	5 0	5 0
5 5	2 3	5 1
2 1	2 2	5 2
2 1	5 1	
5 4	6 2	
2 7	2 1	

31.5.3 Second Method: Boolean Matrix

Here is another method. It computes a boolean matrix of equality on the titles. Row i column j is true where the title in `i{AT` equals the title in `j{TS` . The authors and titles are recovered by by converting the boolean matrix to sparse representation, then taking its index-matrix.

```
   second =: 4 : 0
    'a t'  =. |: x
    'tt s' =. |:  y
    bm     =. t =/ tt        NB. boolean matrix of matches
    sm     =. $. bm          NB. convert to sparse
    im     =: 4 $. sm        NB. index-matrix
    'i j'  =. |: im
    (i { a),. (j { s)
)
```

Now to check the second method for correctness, that is, giving the same results as the first. We don't care about ordering, and we don't care about repetitions, so let us say that two relations are the same iff their sorted nubs match.

```
   same =: 4 : '(~. x /: x) -: (~. y /: y)       '

   (AT2 second TS2)  same (AT2 first TS2)
1
```

Now for some times

```
   t1 =: 6!:2 'AT2 first TS2'
   t2 =: 6!:2 'AT2 second AT2'
   t3 =: 6!:2 'AT3 first TS3'
   t4 =: 6!:2 'AT3 second TS3'

   3 3 $ ' '; (#AT2) ; (#AT3) ; 'first' ; t1; t3 ; 'second' ; t2; t4
+------+----------+--------+
|      |1000      |10000   |
+------+----------+--------+
|first |0.0707548 |5.9401  |
+------+----------+--------+
|second|0.00396342|0.397783|
+------+----------+--------+
```

We see that the advantage of the second method is reduced at the larger size, and we can guess this is because the time to compute the boolean matrix is quadratic in the size. We can use the performance monitor to see where the time goes.

```
   require 'jpm'
   start_jpm_ ''
357142
   z =: AT3 second TS3
```

```
   showdetail_jpm_ 'second'     NB. display
measurements
 Time (seconds)
+--------+--------+---+---------------+
|all     |here    |rep|second         |
+--------+--------+---+---------------+
|0.000007|0.000007|1  |dyad           |
|0.000209|0.000209|1  |[0] 'a t'=.|:x  |
|0.000182|0.000182|1  |[1] 'tt s'=.|:y |
|0.242920|0.242920|1  |[2] bm=.t=/tt   |
|0.139628|0.139628|1  |[3] sm=.$.bm    |
|0.000013|0.000013|1  |[4] im=:4$.sm   |
|0.000194|0.000194|1  |[5] 'i j'=.|:im |
|0.008760|0.008760|1  |[6] (i{a),.(j{s)|
|0.391913|0.391913|1  |total dyad      |
+--------+--------+---+---------------+
```

Evidently much of the time went into computing the boolean matrix at line 2. Can we do better than this?

31.5.4 Third method: boolean matrix with recursive splitting

Here is an attempt to avoid the quadratic time. If the argument is smaller than a certain size, we use the second method above (which is quadratic, but not so bad for smaller arguments).

If the argument is larger than a certain size, we split it into two smaller parts, so that there are no titles shared between the two. Then the method is applied recursively to the parts.

By experimenting, the "certain size" appears to be about 256 on my computer.

```
   third =: 4 : 0
    if. 0 = # x do. return.  0 2 $ 3 end.
    if. 0 = # y do. return.  0 2 $ 3 end.
    'a t'  =. |: x
    'tt s' =. |: y
    if. 256 > # x do.
        bm     =. t =/ tt       NB. boolean matrix of matches
        sm     =. $. bm
        im     =. 4 $. sm       NB. index-matrix
        'i j'  =. |: im
       (i { a),. (j { s)
    else.
        p  =:  <. -: (>./t) + (<./t)  NB. choose "pivot" title
        pv =: t <: p
        x1 =. pv # x
        x2 =. (-. pv) # x
        assert. (#x1) < (#x)
        assert.  (#x2) < (#x)
        qv =. tt <: p
        y1 =. qv # y
        y2 =. (-. qv) # y
        assert. (#y1) < (#y)
        assert. (#y2) < (#y)
        (x1 third y1) , (x2 third y2)
    end.
)
```

Check correctness :

```
    (AT2 third TS2) same (AT2 second TS2)
1
```

And timings. Experiment on my computer shows the second method will run out of space where the third method will succeed.

```
    'AT4 TS4' =: maketestdata  30000
    'AT5 TS5' =: maketestdata 100000
```

```
   t4a =: 6!:2 'AT4 second TS4'
   t5  =: 6!:2 'AT2 third TS2'
   t6  =: 6!:2 'AT3 third TS3'
   t7  =: 6!:2 'AT4 third TS4'
   t8  =: 6!:2 'AT5 third TS5'

   a =:  ' ';     (#AT2); (#AT3) ; (#AT4); (#AT5)
   b =:  'second'; t2;    t4;       t4a;  'limit error'
    c =:   'third' ; t5;     t6;         t7   ;   t8

   3 5 $a,b,c
+------+----------+---------+--------+-----------+
|      |1000      |10000    |30000   |100000     |
+------+----------+---------+--------+-----------+
|second|0.00396342|0.397783 |3.59719 |limit error|
+------+----------+---------+--------+-----------+
|third |0.00178682|0.0204814|0.064824|0.251812   |
+------+----------+---------+--------+-----------+
```

In conclusion, the third method is clearly superior but considerably more complex.

31.6 Golden Rule Example 4: Mandelbrot Set

The Mandelbrot Set is a fractal image needing much computation to produce. In writing the following, I have found to be helpful both the Wikipedia article and the Rosetta Code treatment for the Mandelbrot Set in J:

Computation of the image requires, for every pixel in the image, iteration of a single scalar function until a condition is satisfied. Different pixels will require different numbers of iterations. The final result is the array of counts of iterations for each pixel. Hence

it may appear that the Mandelbrot Set is an inescapably scalar computation. It is not, as the following is meant to show. The Golden Rule applies.

31.6.1 Scalar Versions

The construction of an image begins with choosing a grid of points on the complex plane, one for each pixel in the image. Here is a verb for conveniently constructing the grid.

```
   makegrid =: 3 : 0
    'LL UR delta' =. y
    'xmin ymin' =. +. LL
    'xmax ymax' =. +. UR
    xn =. <. (xmax-xmin) % delta
    yn =. <. (ymax-ymin) % delta
    (|.(ymin + delta * i. yn))  (j. ~/) (xmin + delta * i. xn)
)
```

The arguments are the complex numbers for lower-left and upper-right corners of the image, and a value for the spacing of the points. To demonstrate with a tiny grid

```
   makegrid _2j3 4j5 1.0
_2j4 _1j4 0j4 1j4 2j4 3j4
_2j3 _1j3 0j3 1j3 2j3 3j3
```

For an image, a list of arguments more suitable for present purposes is shown by :

```
   GRID =:  makegrid _2.5j_1 1j1 0.005
```

The image is computed by applying a Mandelbrot function to each pixel in the grid. Here is a suitable Mandelbrot function. It follows

the design outlined in the Wikipedia article,

```
   mfn1 =: 3 : 0
    NB.         y is one pixel-position
    v =. 0j0
    n =. 0
    while. (2 > | v) *. (n < MAXITER) do.
        v  =. y + *: v
        n =. n+1
    end.
    n
)
```

We need to choose a value for `MAXITER`, the maximum number of iterations. The higher the maximum, the more complex the resulting image. For present purposes let us choose a maximum of 64 iterations, which will give a recognisable image.

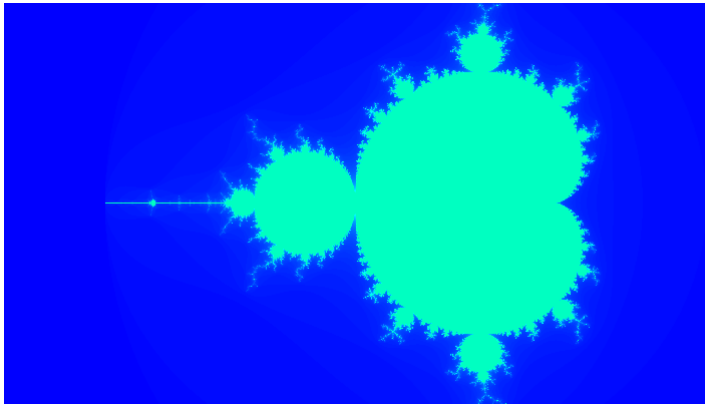
```
   MAXITER =: 64   NB. maximum number of iterations

   image1 =: mfn1 " 0 GRID
```

The result `image1` is a matrix of integers, which can be mapped to colors and then displayed on-screen with:

```
   require '~addons/graphics/viewmat/viewmat.ijs'
   viewmat image1
0
```

to produce an image appear looking something like this:



The time to compute the image:

```
   e1;t1 =: 6!:2 e1 =: 'image1 =: mfn1 " 0 GRID NB. Wikipedia scalar'

+-----------------------------------------------+-------+
|image1 =: mfn1 " 0 GRID NB. Wikipedia scalar|41.4486|
+-----------------------------------------------+-------+
```

The **mfn1** function above was designed to show the algorithm for the scalar one-pixel-at-at-a-time method. Regarding its performance, here is some evidence that its performance is reasonable, that is, comparable to the published Rosetta Code version.

The verb **mfn2** is adapted from the verb **mcf** of the Rosetta Code treatment. It differs only by replacing a numerical constant by the parameter MAXITER.

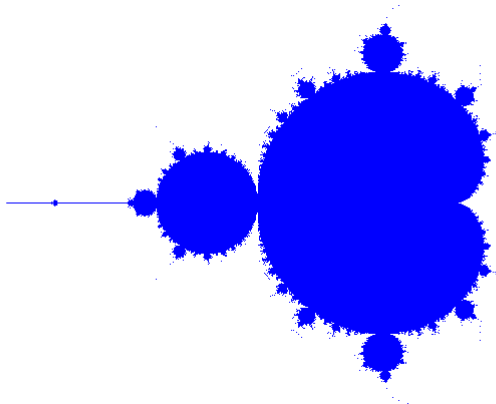
```
   mfn2 =: (<: 2:)@|@(] ((*:@] + [)^:((<: 2:)@|@])^: MAXITER ) 0:)
```

We see that times for **mfn1** and **mfn2** are not very different:

```
   t2 =: 6!:2 e2 =: 'image2 =: mfn2 "0 GRID NB. Rosetta '
   2 2 $ e1; t1; e2; t2
+-------------------------------------------+-------+
|image1 =: mfn1 " 0 GRID NB. Wikipedia scalar|41.4486|
+-------------------------------------------+-------+
|image2 =: mfn2 "0 GRID NB. Rosetta          |35.3052|
+-------------------------------------------+-------+
```

and the image from the Rosetta code is recognisably similar to that from the Wikipedia design,

```
   viewmat image2
0
```



31.6.2 Planar Version

Now we look at a version which computes all pixels at once. Here is a first attempt. It is is a straightforward developement of `mfn1` but here all the computations for every pixel are allowed to run for the maximumum number of iterations.

```
   mfn3  =: 3 : 0           NB. y is entire grid
    N =. ($ y) $ 0
    v =. 0j0
    for_k. i. MAXITER-1 do.
        v =. y + *: v
        N =. N + (2 > | v)
    end.
    1 + N
)
```

For small values of **MAXITER**, this is OK. A quick demonstration, firstly of correctness: it produces the same result as **mfn1**.

```
   MAXITER =: 12

   (mfn1 " 0 GRID) -: (mfn3 " 2 GRID)
1
```

And it's faster:

```
   t1a =: 6!:2 e1a =: 'mfn1 " 0 GRID  NB. Wikip. with MAXITER=12'
   t3a =: 6!:2 e3a =: 'mfn3 " 2 GRID  NB. Planar with MAXITER=12'

   2 2 $  e1a ; t1a; e3a; t3a
+----------------------------------------+--------+
|mfn1 " 0 GRID  NB. Wikip. with MAXITER=12|15.2705 |
+----------------------------------------+--------+
|mfn3 " 2 GRID  NB. Planar with MAXITER=12|0.676646|
+----------------------------------------+--------+
```

Unfortunately there is a problem with any larger values of **MAXITER**. The repeated squaring of the complex numbers in **v** will ultimately produce, not infinity, but a "NaN Error", caused by subtracting infinities. Observe:

```
   (*: ^: 10) 1j3  NB. this is OK
__j__

   (*: ^: 30) 1j3  NB. but this is not
|NaN error
|       (*:^:30)1j3
|[-587] c:\users\homer\13\js\31.ijs


   MAXITER =: 64
```

Here is an attempt to avoid the NaN errors. One cycle in every 10, those values in **v** which have "escaped", (that is, no longer contribute to the final result **N**) are reset to small values to prevent them increasing without limit.

```
   mfn4 =: 3 : 0
    N =. ($ y) $ 0
    v =. 0j0
    for_k.  i. MAXITER - 1  do.
        if. 0 = 10 | k do.
            e =. 2 < | v
            v =. (v * 1-e) + (1.5j1.5 * e )
        end.
    v =. y + *: v
    N =. N + 2 > | v
    end.
    N+1
)
```

In spite of the burden of resetting, the timing looks about 8 times faster than the scalar method:

```
   t4 =: 6!:2 e4 =: 'image4 =: mfn4 " 2 GRID NB. Planar, resetting'

   2 2 $  e1; t1 ;  e4;t4
+-----------------------------------------------+-------+
|image1 =: mfn1 " 0 GRID NB. Wikipedia scalar |41.4486|
+-----------------------------------------------+-------+
|image4 =: mfn4 " 2 GRID NB. Planar, resetting|5.20131|
+-----------------------------------------------+-------+
```

and we check the result is correct:

```
   image4 -: image1
1
```

Some further improvement is possible. The idea is to avoid computing the magnitudes of complex numbers because this involves computing square roots. Instead of requiring the magnitude to be less than 2, we will require the square of the magnitude to be less than 4. To do this complex numbers will be represented as a pair of reals. The resetting business is simplified.

```
   mfn5 =: 3 : 0
    'r0 i0' =. ((2 0 1 & |:) @: +. ) y    NB. Real , imag planes of y
    assert. y -: r0 j. i0
    N =. 0
    a =. r0
    b =. i0
    for_i. i. MAXITER-1 do.
        p =. *: a
        q =. *: b
        r =. p+q                  NB. square of magnitudes
        N =. N + r < 4
        b =. (i0 + +: a*b)    <. 100
        a =. (r0 + p - q)     <. 100
    end.
    N+1
)
```

Timing is improved:

```
   t5 =: 6!:2 e5 =: 'image5 =: mfn5 " 2 GRID NB. no square
roots'
   3 2 $  e1;t1; e4;t4; e5;t5
+-------------------------------------------+-------+
|image1 =: mfn1 " 0 GRID NB. Wikipedia scalar |41.4486|
+-------------------------------------------+-------+
|image4 =: mfn4 " 2 GRID NB. Planar, resetting|5.20131|
+-------------------------------------------+-------+
|image5 =: mfn5 " 2 GRID NB. no square roots  |3.18585|
+-------------------------------------------+-------+
```

and the result is correct

```
   image5 -: image1
1
```

31.7 The Special Code of Appendix B of the Dictionary

In Appendix B of the J Dictionary there are listed about 80 different expressions which are given special treatment by the interpreter to improve performance. Many more expressions are listed in the Release Notes

An example is `+/` . Notice that the speedup only occurs when `+/` itself (as opposed to something equivalent) is recognised.

```
   data =: ? 1e6 $ 1e6  NB. a million random
integers

   plus =: +


   t20 =: 6!:2 e20 =: 'plus / data'
   t21 =: 6!:2 e21 =: '+/ data'

   2 2 $ e20 ; t20; e21; t21
+----------+----------+
|plus / data|0.424388  |
+----------+----------+
|+/ data    |0.00184018|
+----------+----------+
```

The special expressions can be unmasked with the `f.` adverb which translates all defined names into the built-in functions.

```
   foo =: plus /

   t22 =: 6!:2 e22 =: 'foo data'
   t23 =: 6!:2 e23 =: 'foo f. data'

   2 2 $ e22 ; t22; e23; t23
+-----------+----------+
|foo data   |0.430226  |
+-----------+----------+
|foo f. data|0.00188495|
+-----------+----------+
```

The recommendation here is NOT that the programmer should look for opportunities to use these special cases. The recommendation is ONLY to allow the interpreter to find them, by giving, where appropriate, a final little polish to tacit definitions with `f.` .

This brings us to the end of Chapter 31

Chapter 32: Trees

32.1 Introduction

Data structures consisting of boxes within boxes may be called trees. J provides several special functions in support of computations with trees.

Here is an example of a tree:

```
   ] T =:  'the cat' ; 'sat' ; < 'on' ; < ('the';'mat')
+-------+---+--------------+
|the cat|sat|+--+---------+|
|       |   ||on|+---+---+||
|       |   ||  ||the|mat|||
|       |   ||  |+---+---+||
|       |   |+--+---------+|
+-------+---+--------------+
```

Those boxes with no inner boxes will be called leaves. We see that `T` has 7 boxes of which 5 are leaves.

32.2 Fetching

Evidently, the content of any box can be fetched from tree `T` by a combination of indexing and unboxing.

```
   ] a =: > 2 { T
+--+---------+
|on|+---+---+|
|  ||the|mat||
|  |+---+---+|
+--+---------+

   ] b =: > 1 { a
+---+---+
|the|mat|
+---+---+

   ] c =: > 1 { b
mat
```

but there is a built-in verb, "Fetch" (dyadic `{::`) , for this purpose. Its left argument is a sequence of indexes (called a path):

```
   (2;1;1) {:: T
mat
```

Further examples:

```
   2 {:: T
+--+---------+
|on|+---+---+|
|  ||the|mat||
|  |+---+---+|
+--+---------+
```

```
   (2 ;1) {:: T
+---+---+
|the|mat|
+---+---+
```

32.3 The Domain of Fetch

The right argument of `{::` must be a vector, or higher rank, and not a scalar, or else an error results. (Recall that a single box is a scalar).

`0 {:: , <'hello'`	`0 {:: < 'hello'`
`hello`	`error`

Let us say that a full-length path is a path which fetches the data content from a leaf.

Along a full-length path, every index must select a scalar, a box, or else an error results. In other words, we must have a single path.

`T`	`(2; 1 ; 0 1) {:: T`

`+-------+---+-------------+`
`|the cat|sat|+--+---------+|`
`|  |  ||on|+---+---+||`
`|  |  ||  ||the|mat|||`
`|  |  ||  |+---+---+||`
`|  |  |+--+---------+|`
`+-------+---+-------------+`
	`error`

The data fetched from a leaf is the result of opening the last box selected along the path. This data can, as we saw above, be an array, a list say.

```
   (2;1;1) {:: T
mat
```

If this data is an indexable array, then a further index can be appended to a full-length path, giving an over-length path, to fetch a further single scalar. The next example shows fetching `'m'` from `'mat'`.

```
   (2;1;1;0) {:: T
m
```

32.4 The "Map" Verb

Monadic `{::` is called "Map". It shows all the paths to the leaves.

```
   {:: T
+---+---+-----------------------+
|+-+|+-+|+-----+----------------+|
||0|||1|||+-+-+|+-------+-------+||
|+-+|+-+|||2|0|||+-+-+-+|+-+-+-+|||
|   |   ||+-+-+|||2|1|0|||2|1|1||||
|   |   ||     ||+-+-+-+|+-+-+-+|||
|   |   ||     |+-------+-------+||
|   |   |+-----+----------------+|
+---+---+-----------------------+
```

32.5 What is the Height of This Tree?

The verb `L.` ("LevelOf") reports the length of the longest path in a tree, that is, the maximum length of a path to fetch the unboxed data-content of a leaf. In the book "A Programming Language" Kenneth Iverson uses the term "height" for the length of the longest path of a tree.

The length of a path is the number of indexing-and-unboxing steps needed. It is evident that it takes at most 3 steps to fetch any data-content from `T`

```
T                                   L.T
+-------+---+--------------+        3
|the cat|sat|+--+---------+|
|       |   ||on|+---+---+||
|       |   ||  ||the|mat|||
|       |   ||  |+---+---+||
|       |   |+--+---------+|
+-------+---+--------------+
```

One step is needed to fetch the content of the leaf of a tree consisting only of a single leaf, for example `,<6` . The step is `> @: (0&{)`

`A =: ,<6`	`L. A`	`(> @: (0&{)) A`	`0 {:: A`

`+-+`
`|6|`
`+-+`
	`1`	`6`	`6`

and it evidently needs no steps to fetch the content of `'hello'`

`L. 'hello'`	`(0$0) {:: 'hello'`
`0`	`hello`

32.6 Levels and the L: Conjunction

A box with no inner box (a leaf) is said to be at level 0.

Here is another tree:

```
   ] D =: (<'one'; 'two'), (<  'three' ; 'four')
+---------+------------+
|+---+---+|+-----+----+|
||one|two|||three|four||
|+---+---+|+-----+----+|
+---------+------------+
```

We can apply a given function to the values inside the leaves, that is, at level 0, with the aid of the `L:` conjunction (called "Level At").

Reversing the content of each level-0 node, that is, each leaf:

```
   |. L: 0 D
+---------+------------+
|+---+---+|+-----+----+|
||eno|owt|||eerht|ruof||
|+---+---+|+-----+----+|
+---------+------------+
```

Reversing at level 1:

```
   |. L: 1 D
+---------+------------+
|+---+---+|+----+-----+|
||two|one|||four|three||
|+---+---+|+----+-----+|
+---------+------------+
```

and at level 2:

```
   |. L: 2 D
+------------+---------+
|+-----+----+|+---+---+|
||three|four|||one|two||
|+-----+----+|+---+---+|
+------------+---------+
```

We see that we can apply a function at each of the levels 0 1 2 . The level at which the function is applied can also be specified as a negative number:

```
   |. L: _2 D
+---------+------------+
|+---+---+|+-----+----+|
||eno|owt|||eerht|ruof||
|+---+---+|+-----+----+|
+---------+------------+

   |. L: _1 D
+---------+------------+
|+---+---+|+----+-----+|
||two|one|||four|three||
|+---+---+|+----+-----+|
+---------+------------+
```

Levels for trees are analogous to ranks for arrays. `L:` is the analogue of the rank conjunction `"` .

32.7 The Spread Conjunction

We saw above that the result of the `L:` conjunction has the same tree-structure as the argument. There is another conjunction, `S:` (called "Spread") which is like `L:` in applying a function at a level, but unlike `L:` in that the results are delivered, not as a tree but simply as a flat list.

```
   D
+---------+------------+
|+---+---+|+-----+----+|
||one|two|||three|four||
|+---+---+|+-----+----+|
+---------+------------+
```

```
   |. S: 0 D
eno
owt
eerht
ruof
```

The result above is a list (a "flat list") of 4 items, each item being a string.

```
   |. S: 1 D
+----+-----+
|two |one  |
+----+-----+
|four|three|
+----+-----+
```

The result above is a list of 2 items, each item being a list of 2 boxes.

```
   |. S: 2 D
+-----------+---------+
|+-----+----+|+---+---+|
||three|four|||one|two||
|+-----+----+|+---+---+|
+-----------+---------+
```

The result above is a list of 2 items, each item being a box.

32.8 Trees with Varying Path-lengths

In the example tree `D` above all the path-lengths to a leaf are the same length. However, in general path-lengths may vary. For the example tree `T`,

```
   T
+-------+---+--------------+
|the cat|sat|+--+---------+|
|       |   ||on|+---+---+||
|       |   ||  ||the|mat|||
|       |   ||  |+---+---+||
|       |   |+--+---------+|
+-------+---+--------------+
```

the paths are shown by `{:: T` and the lengths of the paths are given by

```
   (# S: 1) {:: T
1 1 2 3 3
```

Reversing the contents of the level-0 nodes gives no surprises:

```
   |. L: 0 T
+-------+---+--------------+
|tac eht|tas|+--+---------+|
|       |   ||no|+---+---+||
|       |   ||  ||eht|tam|||
|       |   ||  |+---+---+||
|       |   |+--+---------+|
+-------+---+--------------+
```

but if we reverse contents of the level-1 nodes we see that some but not all of the level-0 leaves reappear at level 1.

```
   |. L: 1 T
+-------+---+--------------+
|tac eht|tas|+--+---------+|
|       |   ||no|+---+---+||
|       |   ||  ||mat|the|||
|       |   ||  |+---+---+||
|       |   |+--+---------+|
+-------+---+--------------+
```

The explanation is that at level 1 the given verb is applied to

- those nodes strictly at level 1, that is, those for which `1=L. node` AND
- those nodes strictly at level 0 not already accounted for by being contained within a level 1 node.

Similarly, if we reverse the contents of the level-2 nodes we see:

```
   |. L: 2 T
+-------+---+--------------+
|tac eht|tas|+---------+--+|
|       |   ||+---+---+|on||
|       |   |||the|mat||  ||
|       |   ||+---+---+|  ||
|       |   |+---------+--+|
+-------+---+--------------+
```

In this example some of the results of reverse are strings, and some are lists of boxes. They are of different types. These results of different types cannot simply be assembled without more ado

into a flat list as would be attempted by S:

Hence `u S: 1` may fail unless the verb u itself provides uniform results at every node. Compare these two examples:

`|. S: 1 T`	`(< @: |.) S: 1 T`

`error`	`+-------+---+--+---------+`
	`|tac eht|tas|no|+---+---+|`
	`|  |  |  ||mat|the||`
	`|  |  |  |+---+---+|`
	`+-------+---+--+---------+`

The Level conjunction `L:` walks the tree in the same way, that is, it hits the same nodes for reversing,

```
   |. L: 0 T
+-------+---+--------------+
|tac eht|tas|+--+---------+|
|       |   ||no|+---+---+||
|       |   ||  ||eht|tam|||
|       |   ||  |+---+---+||
|       |   |+--+---------+|
+-------+---+--------------+
```

However, Level does not try to build a flat list of results, rather puts each individual result back into its position in the tree. Hence where Spread will fail because it tries to build a flat list, Level will succeed.

|. S: 1 T	|. L: 1 T
	error	+-------+---+--------------+ |tac eht|tas|+--+---------+| | | ||no|+---+---+|| | | || ||mat|the||| | | || |+---+---+|| | | |+--+---------+| +-------+---+--------------+

32.9 L. Revisited

Here we show that the LevelOf a tree can be computed from its Map that is, that L. T, say, can be found from {:: T

```
   {:: T     NB. Map giving the paths to leaves
+---+---+-----------------------+
|+-+|+-+|+-----+-----------------+|
||0|||1|||+-+-+|+-------+-------+||
|+-+|+-+|||2|0|||+-+-+-+|+-+-+-+|||
|   |   ||+-+-+|||2|1|0|||2|1|1||||
|   |   ||     ||+-+-+-+|+-+-+-+|||
|   |   ||     |+-------+-------+||
|   |   |+-----+-----------------+|
+---+---+-----------------------+

   # S: 1 {:: T     NB. the length of each path
1 1 2 3 3
```

```
   >. / # S: 1 {:: T  NB. maximum of the lengths
3

   L. T                  NB.  the LevelOf T
3
```

This is the end of Chapter 32.

Appendix 1: Evaluating Expressions

A1.1 Introduction

Here 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. For example, in the expression `2 * 3 + 4` the right argument of `*` is `3 + 4` so that `2 * 3 + 4` means `2* (3 + 4)`. This we earlier called the "rightmost-first" rule.
- verbs have short left scope. For example in `2 * 3 + 4` the left

argument of `+` is `3`.
- adverbs and conjunctions get applied before verbs. For example `+ & 1 % 2` means `(+ & 1 )% 2`
- adverbs and conjunctions have long left scope and short right scope

These effects describe how an expression is implicitly parenthesized. Of course, we can always produce desired effects by writing explicit parentheses, even though they may not be needed. Further effects are:

- 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.

A1.2 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 step-by-step record, or history, of the evaluation process. This history is displayed by entering the expression `hist ''`

```
  hist ''

Queue          Stack                  Rule

+----------+   +------+---+---+---+   +----+
|mark 2 + 3|   |      |   |   |   |   |    |
+----------+   +------+---+---+---+   +----+
|mark 2 +  |   | 3    |   |   |   |   |    |
+----------+   +------+---+---+---+   +----+
|mark 2    |   | +    | 3 |   |   |   |    |
+----------+   +------+---+---+---+   +----+
|mark      |   | 2    | + | 3 |   |   |    |
+----------+   +------+---+---+---+   +----+
|          |   | mark | 2 | + | 3 |   |dyad|
+----------+   +------+---+---+---+   +----+
|          |   | mark | 5 |   |   |   |    |
+----------+   +------+---+---+---+   +----+
```

We see successive stages of the process. In this example there are six stages. Each stage is defined by the values of two variables. Firstly there is a "queue", initially containing the expression being evaluated, divided into words and preceded by a symbol to mark the beginning. Secondly, there is a "stack", initially empty. The first stage shows queue and stack at the outset.

At each stage the stack is inspected to see if anything can be done, that is, whether the first few words in the stack form a pattern to which a rule applies. There are 9 of these rules, and each one is tried in turn. If no rule applies, then a word is transferred from the tail of the queue to the head of the stack, and we go to the next stage and try again. This process takes us from the first stage to the fifth stage.

At the fifth stage, we find that a rule is applicable. This rule is identified as `dyad` in the rightmost column. Informally, the `dyad` rule is:

if the first four items in the stack are something, noun, verb, noun, then apply verb to noun and noun to get new-noun, and replace the first four items in the stack by two, namely original-something followed by new-noun.

The sixth and last stage shows the results of applying the "dyad" rule recognized at the previous stage. The rules are tried again, with no result, and there are no more words in the queue, so we have finished. The final result is the second item of the stack. The history is maintained in 3 global variables, `Qh` `Sh` and `Rh`. The expression `hist ''` computes a formatted display from these variables.

A1.3 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, adverb, 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'.

A1.3.1 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

```
   hist ''

 Queue          Stack                 Rule

 +---------+   +------+----+---+   +-----+
 |mark *: 4|   |      |    |   |   |     |
 +---------+   +------+----+---+   +-----+
 |mark *:  |   | 4    |    |   |   |     |
 +---------+   +------+----+---+   +-----+
 |mark     |   | *:   | 4  |   |   |     |
 +---------+   +------+----+---+   +-----+
 |         |   | mark | *: | 4 |   |monad|
 +---------+   +------+----+---+   +-----+
 |         |   | mark | 16 |   |   |     |
 +---------+   +------+----+---+   +-----+
```

A1.3.2 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

```
   hist ''
```

```
Queue           Stack                    Rule

+-----------+   +------+-----+----+---+   +------+
|mark - *: 4|   |      |     |    |   |   |      |
+-----------+   +------+-----+----+---+   +------+
|mark - *:  |   | 4    |     |    |   |   |      |
+-----------+   +------+-----+----+---+   +------+
|mark -     |   | *:   | 4   |    |   |   |      |
+-----------+   +------+-----+----+---+   +------+
|mark       |   | -    | *:  | 4  |   |   |      |
+-----------+   +------+-----+----+---+   +------+
|           |   | mark | -   | *: | 4 |   |monad2|
+-----------+   +------+-----+----+---+   +------+
|           |   | mark | -   | 16 |   |   |monad |
+-----------+   +------+-----+----+---+   +------+
|           |   | mark | _16 |    |   |   |      |
+-----------+   +------+-----+----+---+   +------+
```

A1.3.3 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

```
   hist ''

 Queue          Stack                  Rule

 +----------+   +------+----+---+---+   +----+
 |mark 3 * 4|   |      |    |   |   |   |    |
 +----------+   +------+----+---+---+   +----+
 |mark 3 *  |   | 4    |    |   |   |   |    |
 +----------+   +------+----+---+---+   +----+
 |mark 3    |   | *    | 4  |   |   |   |    |
 +----------+   +------+----+---+---+   +----+
 |mark      |   | 3    | *  | 4 |   |   |    |
 +----------+   +------+----+---+---+   +----+
 |          |   | mark | 3  | * | 4 |   |dyad|
 +----------+   +------+----+---+---+   +----+
 |          |   | mark | 12 |   |   |   |    |
 +----------+   +------+----+---+---+   +----+
```

A1.3.4 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

```
   hist ''

 Queue              Stack                                  Rule

+--------------+   +-------+-------+-------+-------+   +-----+
|mark + / 1 2 3|   |       |       |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|mark + /      |   | 1 2 3 |       |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|mark +        |   | /     | 1 2 3 |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|mark          |   | +     | /     | 1 2 3 |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|              |   | mark  | +     | /     | 1 2 3 |   |adv  |
+--------------+   +-------+-------+-------+-------+   +-----+
|              |   | mark  | +/    | 1 2 3 |       |   |monad|
+--------------+   +-------+-------+-------+-------+   +-----+
|              |   | mark  | 6     |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
```

A1.3.5 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

```
   hist ''

Queue            Stack                        Rule

 +------------+   +------+-----+---+---+---+   +-----+
 |mark 1 & + 2|   |      |     |   |   |   |   |     |
 +------------+   +------+-----+---+---+---+   +-----+
 |mark 1 & +  |   | 2    |     |   |   |   |   |     |
 +------------+   +------+-----+---+---+---+   +-----+
 |mark 1 &    |   | +    | 2   |   |   |   |   |     |
 +------------+   +------+-----+---+---+---+   +-----+
 |mark 1      |   | &    | +   | 2 |   |   |   |     |
 +------------+   +------+-----+---+---+---+   +-----+
 |mark        |   | 1    | &   | + | 2 |   |   |     |
 +------------+   +------+-----+---+---+---+   +-----+
 |            |   | mark | 1   | & | + | 2 |   |conj |
 +------------+   +------+-----+---+---+---+   +-----+
 |            |   | mark | 1&+ | 2 |   |   |   |monad|
 +------------+   +------+-----+---+---+---+   +-----+
 |            |   | mark | 3   |   |   |   |   |     |
 +------------+   +------+-----+---+---+---+   +-----+
```

A1.3.6 Trident Rule

The scheme is:

```
trident rule: EAVN VN1 VERB2 VERB3 etc => EAVN Z etc
```

and there are two cases: `VN1` may be a verb or a noun. If `VN1` is the verb `VERB1` then `Z` is the single verb defined as the fork `VERB1 VERB2 VERB3`. Forks and abbreviations for forks are described in Chapter 09.

Here is an example: `1 + *:` is an abbreviation for the fork `1: + *:`

(1: + *:) 2 3	(1 + *:)2 3	'(1 + *:) 2 3' EVM
5 10	5 10	5 10

```
   hist ''
```

```
 Queue                        Stack                                          Rule

 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark ( 1 + *: ) 2 3|        |        |        |    |    |    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark ( 1 + *: )     |       | 2 3    |        |    |    |    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark ( 1 + *:       |       | )      | 2 3    |    |    |    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark ( 1 +          |       | *:     | )      | 2 3|    |    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark ( 1            |       | +      | *:     | )  | 2 3|    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark (              |       | 1      | +      | *: | )  | 2 3|     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark                |       | (      | 1      | +  | *: | )  | 2 3|       |trident|
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark                |       | (      | 1 + *:| )  | 2 3|    |     |       |paren   |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |mark                |       | 1 + *:| 2 3    |    |    |    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |                    |       | mark   | 1 + *:| 2 3|    |    |     |       |monad   |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
 |                    |       | mark   | 5 10   |    |    |    |     |       |        |
 +--------------------+       +--------+--------+----+----+----+-----+       +--------+
```

A1.3.7 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 and the remaining cases in the schemes for bidents in Chapter 15.

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

```
   hist ''

Queue                Stack                          Rule

+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1 & ) 2|   |      |     |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1 & )  |   | 2    |     |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1 &    |   | )    | 2   |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1      |   | &    | )   | 2  |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + (        |   | 1    | &   | )  | 2 |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark +          |   | (    | 1   | &  | ) | 2 |   |bident|
+----------------+   +------+-----+----+---+---+   +------+
|mark +          |   | (    | 1&  | )  | 2 |   |   |paren |
+----------------+   +------+-----+----+---+---+   +------+
|mark +          |   | 1&   | 2   |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark            |   | +    | 1&  | 2  |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|                |   | mark | +   | 1& | 2 |   |   |adv   |
+----------------+   +------+-----+----+---+---+   +------+
|                |   | mark | 1&+ | 2  |   |   |   |monad |
+----------------+   +------+-----+----+---+---+   +------+
|                |   | mark | 3   |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
```

A1.3.8 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`

```
   hist ''

Queue               Stack                    Rule

+---------------+   +------+----+---+---+   +------+
|mark 1 + x =: 6|   |      |    |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark 1 + x =:  |   | 6    |    |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark 1 + x     |   | =:   | 6  |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark 1 +       |   | x    | =: | 6 |   |   |assign|
+---------------+   +------+----+---+---+   +------+
|mark 1 +       |   | 6    |    |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark 1         |   | +    | 6  |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark           |   | 1    | +  | 6 |   |   |      |
+---------------+   +------+----+---+---+   +------+
|               |   | mark | 1  | + | 6 |   |dyad  |
+---------------+   +------+----+---+---+   +------+
|               |   | mark | 7  |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
```

A1.3.9 Parenthesis Rule

The scheme is:

```
paren rule: ( CAVN ) etc => Z etc
```

where `Z` is the value of `CAVN`. For example:

(1+2)*3	'(1+2)*3' EVM
9	9

```
   hist ''

 Queue                   Stack                                Rule

 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark ( 1 + 2 ) * 3|    |      |   |   |   |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark ( 1 + 2 ) *  |    | 3    |   |   |   |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark ( 1 + 2 )    |    | *    | 3 |   |   |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark ( 1 + 2      |    | )    | * | 3 |   |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark ( 1 +        |    | 2    | ) | * | 3 |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark ( 1          |    | +    | 2 | ) | * | 3 |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark (            |    | 1    | + | 2 | ) | * | 3 |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark              |    | (    | 1 | + | 2 | ) | * | 3 |    |dyad  |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark              |    | (    | 3 | ) | * | 3 |   |   |    |paren|
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |mark              |    | 3    | * | 3 |   |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |                  |    | mark | 3 | * | 3 |   |   |   |    |dyad  |
 +------------------+    +------+---+---+---+---+---+---+    +------+
 |                  |    | mark | 9 |   |   |   |   |   |    |      |
 +------------------+    +------+---+---+---+---+---+---+    +------+
```

A1.3.10 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`

```
hist ''
```

Queue	Stack	Rule
mark (a =: 7) , a		
mark (a =: 7) ,	6	
mark (a =: 7)	, 6	
mark (a =: 7	) , 6	
mark (a =:	7) , 6	
mark (a	=: 7) , 6	
mark (	a =: 7) , 6	assign
mark (	7) , 6	
mark	(7) , 6	paren
mark	7 , 6	
	mark 7 , 6	dyad
	mark 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

```
 hist ''

Queue                       Stack                               Rule

+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =: - ) , f ) 4| |      |      |  |  |  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =: - ) , f )  | | 4    |      |  |  |  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =: - ) , f    | | )    | 4    |  |  |  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =: - ) ,      | | f    | )    | 4|  |  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =: - )        | | ,    | f    | )| 4|  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =: -          | | )    | ,    | f| )| 4|  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f =:            | | -    | )    | ,| f| )| 4|  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( ( f               | | =:   | -    | )| ,| f| )| 4|  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( (                 | | f    | =:   | -| )| ,| f| )| 4| |assign |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark ( (                 | | -    | )    | ,| f| )| 4|  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark (                   | | (    | -    | )| ,| f| )| 4|  | |paren  |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark (                   | | -    | ,    | f| )| 4|  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark                     | | (    | -    | ,| f| )| 4|  |  | |trident|
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark                     | | (    | - , f| )| 4|  |  |  |  | |paren  |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|mark                     | | - , f| 4    |  |  |  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|                         | | mark | - , f| 4|  |  |  |  |  | |monad  |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
|                         | | mark | _4 _4|  |  |  |  |  |  | |       |
+-------------------------+ +------+------+--+--+--+--+--+--+ +-------+
```

A1.3.11 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	Verb 1	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	VN1	Verb2	Verb3	EAVN	Z	etc	fork (VN1 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

A1.4 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.

A1.4.1 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`

```
   hist ''

Queue              Stack                    Rule

+--------------+   +------+---+---+---+   +----+
|mark 4 + 3 - 2|   |      |   |   |   |   |    |
+--------------+   +------+---+---+---+   +----+
|mark 4 + 3 -  |   | 2    |   |   |   |   |    |
+--------------+   +------+---+---+---+   +----+
|mark 4 + 3    |   | -    | 2 |   |   |   |    |
+--------------+   +------+---+---+---+   +----+
|mark 4 +      |   | 3    | - | 2 |   |   |    |
+--------------+   +------+---+---+---+   +----+
|mark 4        |   | +    | 3 | - | 2 |   |dyad|
+--------------+   +------+---+---+---+   +----+
|mark 4        |   | +    | 1 |   |   |   |    |
+--------------+   +------+---+---+---+   +----+
|mark          |   | 4    | + | 1 |   |   |    |
+--------------+   +------+---+---+---+   +----+
|              |   | mark | 4 | + | 1 |   |dyad|
+--------------+   +------+---+---+---+   +----+
|              |   | mark | 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".

A1.4.2 Operators Before Verbs

Adverbs and conjunctions get applied first, and then the resulting verbs:

* & 1 % 2	(*&1) % 2	'* & 1 % 2' EVM
0.5	0.5	0.5

```
  hist ''
Queue                Stack                                 Rule
+---------------+   +------+-----+-----+---+---+---+   +-------+
|mark * & 1 % 2|   |      |     |     |   |   |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|mark * & 1 %   |   | 2    |     |     |   |   |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|mark * & 1     |   | %    | 2   |     |   |   |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|mark * &       |   | 1    | %   | 2   |   |   |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|mark *         |   | &    | 1   | %   | 2 |   |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|mark           |   | *    | &   | 1   | % | 2 |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|               |   | mark | *   | &   | 1 | % | 2 |   |conj   |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|               |   | mark | *&1 | %   | 2 |   |   |   |monad2|
+---------------+   +------+-----+-----+---+---+---+   +-------+
|               |   | mark | *&1 | 0.5 |   |   |   |   |monad  |
+---------------+   +------+-----+-----+---+---+---+   +-------+
|               |   | mark | 0.5 |     |   |   |   |   |       |
+---------------+   +------+-----+-----+---+---+---+   +-------+
```

A1.4.3 Operators Have Long Left Scope

In the following examples, note that values of verbs are shown in the "parenthesized representation" (see Chapter 27) to show their structure. An adverb or a conjunction takes as its left argument as much as possible. Look at the structure of these 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`

```
   hist ''
 Queue              Stack                                  Rule

 +--------------+   +------+---------+---+---+---+---+   +----+
 |mark f & g @ h|   |      |         |   |   |   |   |   |    |
 +--------------+   +------+---------+---+---+---+---+   +----+
 |mark f & g @  |   | h    |         |   |   |   |   |   |    |
 +--------------+   +------+---------+---+---+---+---+   +----+
 |mark f & g    |   | @    | h       |   |   |   |   |   |    |
 +--------------+   +------+---------+---+---+---+---+   +----+
 |mark f &      |   | g    | @       | h |   |   |   |   |    |
 +--------------+   +------+---------+---+---+---+---+   +----+
 |mark f        |   | &    | g       | @ | h |   |   |   |    |
 +--------------+   +------+---------+---+---+---+---+   +----+
 |mark          |   | f    | &       | g | @ | h |   |   |    |
 +--------------+   +------+---------+---+---+---+---+   +----+
 |              |   | mark | f       | & | g | @ | h |   |conj|
 +--------------+   +------+---------+---+---+---+---+   +----+
 |              |   | mark | f&g     | @ | h |   |   |   |conj|
 +--------------+   +------+---------+---+---+---+---+   +----+
 |              |   | mark | (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".

A1.4.4 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 / ' EVM`
`(f g h)/`	`f g (h/)`	`f g (h/)`

```
   hist ''

 Queue            Stack                          Rule

+------------+   +------+----------+----+----+   +-------+
|mark f g h /|   |      |          |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark f g h  |   | /    |          |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark f g    |   | h    | /        |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark f      |   | g    | h        | /  |    |   |adv    |
+------------+   +------+----------+----+----+   +-------+
|mark f      |   | g    | h/       |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark        |   | f    | g        | h/ |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|            |   | mark | f        | g  | h/ |   |trident|
+------------+   +------+----------+----+----+   +-------+
|            |   | mark | f g (h/) |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
```

Similarly for a conjunction (with a right argument)

f g h @ +	'f g h @ +' EVM
f g (h@+)	f g (h@+)

```
   hist ''

Queue              Stack                                  Rule

+--------------+   +------+-----------+-----+-----+      +-------+
|mark f g h @ +|   |      |           |     |     |      |       |
+--------------+   +------+-----------+-----+-----+      +-------+
|mark f g h @  |   | +    |           |     |     |      |       |
+--------------+   +------+-----------+-----+-----+      +-------+
|mark f g h    |   | @    | +         |     |     |      |       |
+--------------+   +------+-----------+-----+-----+      +-------+
|mark f g      |   | h    | @         | +   |     |      |       |
+--------------+   +------+-----------+-----+-----+      +-------+
|mark f        |   | g    | h         | @   | +   |      |conj   |
+--------------+   +------+-----------+-----+-----+      +-------+
|mark f        |   | g    | h@+       |     |     |      |       |
+--------------+   +------+-----------+-----+-----+      +-------+
|mark          |   | f    | g         | h@+ |     |      |       |
+--------------+   +------+-----------+-----+-----+      +-------+
|              |   | mark | f         | g   | h@+ |      |trident|
+--------------+   +------+-----------+-----+-----+      +-------+
|              |   | mark | 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)@

```
  hist ''

Queue            Stack                                  Rule

+------------+   +------+----------+---+---+---+   +-------+
|mark f g h @|   |      |          |   |   |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
|mark f g h  |   | @    |          |   |   |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
|mark f g    |   | h    | @        |   |   |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
|mark f      |   | g    | h        | @ |   |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
|mark        |   | f    | g        | h | @ |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
|            |   | mark | f        | g | h | @ |   |trident|
+------------+   +------+----------+---+---+---+   +-------+
|            |   | mark | f g h    | @ |   |   |   |bident |
+------------+   +------+----------+---+---+---+   +-------+
|            |   | mark | (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)/`

```
  hist ''

Queue            Stack                          Rule

+------------+   +------+--------+---+---+---+   +----+
|mark f @ g /|   |      |        |   |   |   |   |    |
+------------+   +------+--------+---+---+---+   +----+
|mark f @ g  |   | /    |        |   |   |   |   |    |
+------------+   +------+--------+---+---+---+   +----+
|mark f @    |   | g    | /      |   |   |   |   |    |
+------------+   +------+--------+---+---+---+   +----+
|mark f      |   | @    | g      | / |   |   |   |    |
+------------+   +------+--------+---+---+---+   +----+
|mark        |   | f    | @      | g | / |   |   |    |
+------------+   +------+--------+---+---+---+   +----+
|            |   | mark | f      | @ | g | / |   |conj|
+------------+   +------+--------+---+---+---+   +----+
|            |   | mark | f@g    | / |   |   |   |adv |
+------------+   +------+--------+---+---+---+   +----+
|            |   | mark | (f@g)/ |   |   |   |   |    |
+------------+   +------+--------+---+---+---+   +----+
```

A1.4.5 Presumption of Verb

A name with no value assigned is presumed to be a verb. For example, in the following the two names make a hook:

Blue Skies	'Blue Skies' EVM
Blue Skies	Blue Skies

```
   hist ''

 Queue              Stack                             Rule

 +---------------+  +-------+-----------+-------+   +------+
 |mark Blue Skies|  |       |           |       |   |      |
 +---------------+  +-------+-----------+-------+   +------+
 |mark Blue      |  | Skies |           |       |   |      |
 +---------------+  +-------+-----------+-------+   +------+
 |mark           |  | Blue  | Skies     |       |   |      |
 +---------------+  +-------+-----------+-------+   +------+
 |               |  | mark  | Blue      | Skies |   |bident|
 +---------------+  +-------+-----------+-------+   +------+
 |               |  | mark  | Blue Skies|       |   |      |
 +---------------+  +-------+-----------+-------+   +------+
```

Appendix 2: Collected Terminology

In this book, the words "data", "function", "argument" and "expression" are used with the meanings usual in programming.

Certain other words are used in this book with meanings given below, in a sequence such that the explanation of each word depends only on words previously explained.

VALUE	Anything which can be produced by evaluating an expression is said to be a value. Every value is a data value or a function.
NOUN	a data value
VERB	a function which computes nouns from nouns.
MONAD	a verb which takes a single argument.
DYAD	a verb which takes two arguments. Every verb is a monad or a dyad.
AMBIVALENT	An expression is said to be ambivalent when it denotes either a monad or a dyad (depending on whether one or two arguments are supplied).
OPERATOR	a function which takes, as its argument(s), nouns or verbs, and produces as its result, a noun or verb or

	operator. Every J function is a verb or an operator.
ADVERB	an operator which takes a single argument.
CONJUNCTION	an operator which takes two arguments. Every operator is an adverb or a conjunction.
BIDENT	a sequence of two expressions for which the J grammar provides an interpretation as a single function.
TRIDENT	a sequence of three expressions for which the J grammar provides an interpretation as a single function.
TRAIN	a sequence of two or more expressions for which the J grammar provides an interpretation as a single function.
HOOK	a verb defined as a sequence of two verbs, that is, a bident.
FORK	a verb defined as a sequence of three verbs, that is, a trident.
EXPLICIT	a function is said to be explicitly defined, or just explicit, when defined by an expression containing argument variables for which values are to be substituted.

TACIT	a function is said to be tacitly defined, or just tacit, when defined without using argument variables. Every J function is either built-in or explicit or tacit.
ARRAY	a noun, that is, a data value, consisting of a number of simpler values arranged on rectangular coordinates, or axes. Every noun is an array, with zero or more axes.
DIMENSION	(of an array) the length of an axis
SHAPE	(of an array) the list of its dimensions
SCALAR	a noun with no dimensions. The shape of a scalar is an empty list.
RANK	(of a noun) the number of its dimensions, that is, the length of its shape.
BOX	A scalar of a special type, such that its value can represent any array.
CELL	The list of dimensions of any array can be arbitrarily partitioned into leading dimensions followed by trailing dimensions. The original array is thus described as an array of cells, where each cell has only the trailing dimensions. The leading dimensions are called a frame for those cells.

FRAME	See Cell.
RANK	(of a verb) The natural, or intrinsic, rank for its argument(s). With an argument of any rank higher than its intrinsic rank, the verb is applied separately to each intrinsic-rank cell of the argument. A monad has one rank, a dyad has two (one each for left and right arguments) and hence an ambivalent verb has three.

Table of Contents
Index

Index

Table of Contents