Warwick Guide to J
(Version 3.05)

J. E. H. Shaw

March 11, 1998
Abstract

J is both a formal mathematical language and a general-purpose programming language, implemented on a wide range of computers. It handles vectors, matrices, higher-dimensional arrays and more complicated data structures in a consistent and efficient way. Thus J is an ideal tool both for research and for day-to-day programming.

Like many powerful mathematical constructions however, J has a reputation for being difficult to learn. This Guide therefore provides a less formal description of J than is contained in most of the official documentation, together with many examples, and descriptions of additional facilities such as graphics and inter-process communication. The aim is to encourage other people, whether at Warwick or elsewhere, in Statistics or in other areas, to use J.

This Guide has been under construction since December 1997: currently sections 1–2.3.6.3 and 4.3 are complete. Updated versions of the Guide will be made available as they are produced. Comments are welcome, and constructive comments doubly so!
# Contents

1 Introduction .................................................. 3
  1.1 What is J? .................................................... 3
  1.2 Why Use J? ................................................... 4
  1.3 Why This Document? .......................................... 5

2 The J Language ................................................ 7
  2.1 J Vocabulary ................................................ 7
    2.1.1 Verbs ................................................... 7
    2.1.2 Nouns ................................................... 9
    2.1.3 Conjunctions ......................................... 9
    2.1.4 Adverbs ............................................... 9
    2.1.5 J Words in Context .................................. 9
  2.2 Data ........................................................ 11
    2.2.1 Characters ............................................. 11
    2.2.2 Numbers ................................................. 12
    2.2.3 Boxes .................................................. 20
    2.2.4 Arrays ................................................. 20
    2.2.5 Data Structures ...................................... 28
  2.3 J Grammar .................................................. 31
    2.3.1 Order of Evaluation .................................. 31
    2.3.2 Trains and Hooks and Forks ........................ 32
    2.3.3 Some Adverbs ......................................... 37
    2.3.4 Actions on Arrays .................................... 40
    2.3.5 Matrix Algebra ....................................... 46
    2.3.6 More Conjunctions ................................... 52
    2.3.7 Miscellaneous Constructions ....................... 58
  2.4 J Programming ............................................. 55
    2.4.1 Assignment ............................................. 55
    2.4.2 Tacit Definition ..................................... 55
    2.4.3 Explicit Definition ................................... 55
    2.4.4 Recursion and Self-Reference ($:)$ ................. 55
    2.4.5 Examples .............................................. 56
    2.4.6 Control Structure .................................... 58

3 J Implementation ............................................. 59
  3.1 J for Windows .............................................. 59
  3.2 Foreign Conjunction ...................................... 60
    3.2.1 Families of Foreign Conjunctions .................. 60
    3.2.2 List of Foreign Conjunctions ....................... 62
  3.3 Associated Scripts and Packages ......................... 64
    3.3.1 Mathematics .......................................... 64
3.3.2 Regular Expressions ................................................. 64
3.4 Lab Sessions .......................................................... 64
3.5 Package Development ............................................... 65
  3.5.1 Locales ............................................................. 65
  3.5.2 Debugging ......................................................... 65
  3.5.3 Distribution ....................................................... 65
3.6 Graphics Interface .................................................. 65
3.7 J Support .............................................................. 65
  3.7.1 Help Files ......................................................... 65
  3.7.2 J Books and Papers ............................................. 65
  3.7.3 J on the Net ....................................................... 65

4 J Examples ............................................................. 66
  4.1 Utilities .............................................................. 66
    4.1.1 Standard J Utilities ........................................ 66
    4.1.2 My Utilities ................................................... 66
  4.2 Statistics ............................................................ 72
  4.3 Timetabling .......................................................... 73
    4.3.1 J Timetabling Script ....................................... 73
    4.3.2 TeX Timetables Produced by J ............................ 79

References ............................................................. 84
Chapter 1

Introduction

1.1 What is J?

J is a concise and powerful language for communicating mathematical ideas unambiguously, not least between humans and computers.

J has the usual built-in operations like + (plus), - (minus), * (times), % (divided by), and ^ (exponential or power). Many other important mathematical ideas are expressed in J as a character followed by a colon or a full stop. For example, - represents not, ♯ is matrix inverse, %: is root (including square root), ^ .* is log, p. is polynomial, and T. is Taylor series approximation.

J genuinely is best thought of as a language—objects are nouns, functions are verbs whose action may be modified by adverbs like / (see below), etc. Individual J words are combined to form sentences (instructions to the computer). For example, the following brief dialogue between myself and the computer:

```
   y=. i. 10
   y
   0 1 2 3 4 5 6 7 8 9
   +/ y
   45
   # y
   10
   mean=. +/ % #
   mean y
   4.5
```

may be read as:

*ME*: Define y to be a list of the first 10 nonnegative integers.

[i. is an inbuilt J verb.]

*ME*: What is y?

[J sentences typed into a computer are either definitions, containing =. (or =:), or else instructions to evaluate something, i.e. implicit questions of the type ‘what do you understand by...?’]

*PC*: <Answers>.

[Note that whatever I type is indented but the computer’s responses are not.]
ME: Sum over y.
   [More formally: insert a ‘*’ between the items of y, and evaluate the resulting expression.]

PC: <Answers>.
   [Correctly!]

ME: What is the number of items in y?
   [# is another inbuilt J verb.]

PC: <Answers>.

ME: Define mean to be ‘sum over’ divided by ‘number of items in’
   [Note that verbs—functions—in J can be defined directly, without reference to dummy arguments etc.
      This is appropriate because much of mathematics, such as group theory, studies verbs rather than nouns.]

ME: What is the mean of y?

PC: 4.5

   [I have now taught the computer the definition of ‘mean’]

J instructions such as mine above are conveniently stored in text files called scripts. J comes with various
scripts to handle files, graphics, inter-process communication etc., together with example packages for linear
algebra, statistical analysis and various other subjects.

For example, a standard library script stdlib is automatically read on starting J. It defines several useful
nouns such as CR LF (carriage return + line feed) and verbs such as names which lists objects of a specified
type (e.g. names noun lists all currently named nouns). See Section 4.1.1

1.2 Why Use J?

J is a descendant of APL [13], but improves over it in many important respects [4, 6]. Whether or not you
have already met APL, I urge you to try J because of its many strengths:

Power
   Like APL (but much more so), J is a powerful language for expressing general mathematical concepts,
   for quickly trying out crazy ideas to see which ones aren’t so crazy after all, and for catalysing insights
   into the underlying mathematics. It is thus an ideal research tool.

Usability
   On the other hand, by learning some basic J vocabulary and grammar, beginners can use J productively
   without being held up by either the generality or the intricacies of the language. An expression like
   mean=. +/ % # is easy to read and write without knowing the precise rules that make it work.

Portability
   Unlike APL, J uses just standard ASCII characters. This makes for easier implementation and communi-
cation: J is fully portable to any computer system with a C compiler.

Coherence
   J is not designed by a committee—it is the brainchild of one person (Ken Iverson), with additional input
   by Roger Hui and feedback from other users. J didn’t have to be downwardly compatible with anything
   (don’t think of it as APL++)
Programming Environment
The J development environment makes it easy to split up a huge programming task into manageable chunks.

Windows Interface
The PC implementation makes MS-Windows programming bearable. Impressive-looking user interfaces can be produced with minimum pain.

Versatility
The PC implementation for professional developers can be linked (using DDE, DLLs, OLE etc.) to other software, such as browsers, spreadsheets, and packages for graphics, statistics or number-crunching. J can access the features of other software, and vice versa.

Value
A version of J, including all documentation, is freely available from http://www.jsoftware.com: you can try before you buy. Before 1996 I used a commercial APL, whose vendors wanted £1500 per annum from the Statistics Department just for continuing support.

Efficiency
The complete J system takes around 5 Mb disk space. Small is beautiful.

Vitality
J is a living language: new features appear frequently.

Support
J is well-supported on the Internet, in the newsgroup comp.lang.apl and at ISI’s homepage http://www.jsoftware.com/.

J manuals [2, 3, 12, 15, 16] are available from ISI (Iverson Software Inc.), and Windows help file versions are freely downloadable from http://www.jsoftware.com/. ISI also publish other books featuring J applications [10, 14, 19].

1.3 Why This Document?

This report is by no means a substitute for the standard documentation, which is comprehensive (if terse), or for experimenting yourself with J. Indeed, the best way to learn any language is to converse with a native speaker (in J’s case, the computer), while consulting a phrase-book [2] and dictionary [15] as necessary! However, there are several reasons for this document:

1. To provide an informal introduction to J, aiming to give a feel for how J can be learnt and used in practice.

2. To contain information not in the standard documentation [2, 3, 12, 15], such as features of J developed only since the documentation was written.

3. To present some particular applications of J—in research, teaching and even administration!

4. To encourage other researchers, particularly at Warwick, to use J to our mutual benefit.

5. To have a readily-available introductory document on J that can be read even without access to J on a computer, and that serves as a basic reference for other research reports and papers.
Chapter 2 outlines the main features of the J language, concentrating on the features that I found most useful soonerest, and giving pointers to where further details are available.

Chapter 3 describes the Windows 3.1 implementation of J, including the programming environment.

Finally, Chapter 4 contains examples of general J utilities (4.1), and uses of J in statistics (4.2) and departmental timetabling (4.3).
Chapter 2

The J Language

2.1 J Vocabulary

The full J Vocabulary, extracted and augmented from the J Dictionary help file, is shown in Table 2.1. Note that I., $; x: and {:: have all been added since J 3.01, when the current help files were created.

Verbs in Table 2.1 are shown in Sans serif font, nouns in Slanted, conjunctions in Bold (if linked to verbs) or CAPITALS (if linked to nouns), and adverbs in Bold Slanted.

2.1.1 Verbs

Most verbs (functions) can be monadic with just a right argument (denoted y.), or dyadic with left and right arguments x. and y. respectively. For example, Table 2.1 shows that % is a verb with monadic—dyadic meanings Reciprocal—Divided by: % 4 5 evaluates to 0.25 0.2 (since here y. is the list of two numbers 4 5), and 4%5 evaluates to 0.8. Similarly, < is a verb that has monadic and dyadic forms:

\[
\begin{array}{l}
< 3 1 4 1 & \text{NB. Monadic verb } < \text{ means 'box y.'} \\
3 1 4 1 & \text{NB. Dyadic } < \text{ means 'is x. less than y.?'} (1 \text{ if true, 0 if false})
\end{array}
\]

Note that some features of J will occur in this document before they are formally defined; boxes are explained in Section 2.2.3. Note also how a comment is implemented in J as the verb NB. (comment) which swallows its right argument (up to the end of the line), and then itself disappears!

An expression like \( n \log \log n \) in mathematics, and the corresponding J expression \( n * ^._. n \), can be read as 'n times the result of log of the result of log of n'. Thus in J, as (usually) in maths, a sequence of verbs is in effect evaluated from right to left. You may generally find it helpful to intone 'the result of' after every verb when reading a complicated J expression.
| Mirror  | Self-Classify — Equal  | =: | Is (Global) |
| Box — Less Than | <: | <: | Decrem — Less Or Eq |
| Open — Larger Than | >: | >: | Increm — Larg Or Eq |
| Negative Sign / Infinit | _ | _ | Infinity |
| Conjugate — Plus | + | + | Double — Not-Or |
| Signum — Times | * | * | Square — Not-And |
| Negate — Minus | - | - | Halve — Match |
| Reciprocal — Divided by | % | % | Square Root — Root |
| Exponential — Power | $ | $ | Power |
| Shape Of — Shape | ~ | ~ | Self-Reference |
| Magnitude — Residue | | | |
| Det — Dot Product | | | |
| Explicit — Monad/Dyad | | | |
| Ravel — Append | | | |
| Raze — Link | | | |
| Tally — Copy | | | |
| Factorial — Out Of | | | |
| Insert — Table — Insert | / | / | |
| Prefix — Infix — Train | \ | \ | |
| Same — Left | [ | [ | |
| Same — Right | ] | ] | |
| Catalogue — From | { | { | |
| Item Amend — Amend | \ | \ | |
| Rank — Constant | : | : | |
| Tie (Gerund) | @ | @ | |
| Atop | & | & | |
| Bond/Compose | ? | ? | |
| Roll — Deal | :: | :: | |

| Alphabet  | Ace | A. | Atomic Permute |
| Boolean/Basic | Characteristic values | C. | Cycle-Dir — Perm |
| Derivative | Secant Slope | e. | Raze In — Memb |
| Member of Interval | Fix | H. | Hypergeometric |
| Integer — Index of | Integral | j. | Imaginary — Complex |
| Level | Level | NB. | Comment |
| Pi Times — Circl | Polynomial | p. | Prime |
| Prime Factors | Angle — Polar | S. | Spread |
| Assign Taylor coef | Wgd Taylor coef | T. | Taylor Function |
| Extend | _9: to 9: | Constant functions |

Table 2.1: Vocabulary (based on the J Introduction and Dictionary[15])
2.1.2 Nouns

Table 2.1 shows that _ on its own is the noun ‘Infinity’, and _*. _* is ‘Indeterminate’. Thus _* evaluates to _._ (‘infinity over infinity is indeterminate’). Similarly a. is a built-in noun containing the J alphabet:

NB. 33rd to 96th characters of alphabet a.
32 } 96 { a.
!"#$% & ’()++-.0123456789:;<=?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`a\b\c\d\e\f\g\h\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\y\z\{\|\}~\`

2.1.3 Conjunctions

Conjunctions join their left and right arguments, giving the result a particular interpretation. Full details are in the J help files. Conjunctions whose right argument is a verb u. are shown in Bold in Table 2.1. For example, D. is a conjunction meaning y. th Derivative of u.; so D. 1 is the first derivative of natural log (**.), and (**. D. 1) 4 evaluates to 0.25. Similarly **: is the Power conjunction, signifying repeated application of a function:

\( (+: ^: 6) 2 \) \( \text{NB. Double, 6 times, starting with } y.=2 \)
128

Other conjunctions, shown in CAPITALS, take a noun x. as their left argument; for example dyadic " (‘CONSTANCE’) creates a constant verb with value x. and rank (as explained in Section 2.3.4.1) y.:

\( (100".) 1 \) \( \text{NB. } 100". \text{ is a verb returning 100 for any argument } y. \)
100

2.1.4 Adverbs

Adverbs modify the action of verbs, for example monadic / is the adverb ‘Insert’: u. / y. inserts the verb u. between the items of y. and then evaluates the result. Thus +/ is a verb that returns the sum of the items of its right argument (informally, ‘sum over’):

\( +/ 3 1 4 1 \) \( \text{NB. } +/ \text{ inserts } + \text{ between each item of } y. \text{ & evaluates result } \)
9

Note that adverbs come after the verb they modify, as usually happens in English (‘run quickly’, ‘run backwards’).

2.1.5 J Words in Context

The interpretation of individual words in any language depends on the context (e.g. ‘time flies like an arrow; fruit flies like a banana’). Similarly in J, the meaning of the words shown in Table 2.1 can be worked out from the context. For example, ~ has three possible interpretations:

1. After a monadic verb f, ~ is the adverb ‘Reflexive’ which turns f into a dyadic reflexive verb: f~ y. means y. f y.
2. After a dyadic verb f, ~ is the adverb ‘Passive’, giving f the passive tense. Thus x. f~ y. means y. f x.
3. After a name 'm', ~ is the conjunction 'Evoke': 'm'~ simply means m

As with all languages, interpreting words in context comes naturally with practice. The different uses of ~ are illustrated in the following J dialogue:

```
^ ~ 3 4 5 6  
27 256 3125  46656  
3 4 ~ 5 6  
125 1296  
'mean' ~ 3 4 5 6  
```

NB. reflexive, i.e. (3~3), (4~4), (5~5), (6~6)

NB. passive tense, i.e. (5~3), (6~4)

NB. evoke verb named 'mean' on the list 3 4 5 6

4.5

The monadic verb ;: (word formation) is useful if you have difficulty working out which are the individual words in a particular J expression:

```
(mean=:+/%#) data=.0 1 2 3 4 5 6 7 8 9
4.5
;:'(mean=:+/%#)data=.0 1 2 3 4 5 6 7 8 9'
```

```
<table>
<thead>
<tr>
<th>mean</th>
<th>=: + / % #</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>
```

Comments:

1. The argument to ;: is a list of characters, i.e. a text string, which in J is delimited by single quotes.

2. Some of the words (, =: ; + / % ; ) are J primitives—inbuilt words that are fully explained in the equivalent of Table 2.1 in the on-line J help.

3. User-defined names (mean, data) are words.

4. The list of numbers 0 1 2 3 4 5 6 7 8 9 is also a single word. This is very important—it implies that in a J expression like ~. 0 1 2 3 4 5 6 7 8 9, the argument of ~. (log) is the whole array of numbers.

5. J (like English) has punctuation: a subexpression in parentheses is in effect evaluated in isolation and the result substituted back into the whole J expression.

6. Extra spaces can be inserted between J words to improve readability, e.g. the first line in the above dialogue would be clearer as (mean=:+/%#) data=.0 1 2 3 4 5 6 7 8 9

   Sometimes spaces are essential as well as recommended: mean data is different from mean data. Similarly, remember that many J primitives end with . or ;, so (for example) (+/ . *) & (3 0) are very different from (+/ . *) & (3 0) or (+/ . *) & (3 : 0), although they could be written (+/ . *) & (3 : 0) respectively.

7. While learning J, I have sometimes found it helpful to point at the words and read aloud! (I'm not proud).
2.2 Data

J stores all manner of data types, including high-dimensional tables and unbalanced arrays, in a systematic and efficient way. A data object in J, i.e. a noun, is a structure made out of individual ‘atoms’, which have just three basic types—character, number, or box.

Atoms can be joined together in a list (e.g. 0 1 2 3 4 5 6 7 8 9 is a list of ten numbers), but different types cannot be mixed. The following shows successful attempts to append one character to another (using the verb ,) and one number to another, and an unsuccessful attempt to append a number to a character:

```
cd 'c', 'd'
3, 1
3 1 'c', 3
```

However, anything can be put into a box, and individual boxes then joined into a list. The verb ; (link) is particularly useful for displaying the results of simple J expressions: x. ; y. boxes x., also boxes y. if necessary, and links them together:

```
c 3 ("something in a box") ; 'a list of characters' ; 3 1 4 1 59.26
```

### 2.2.1 Characters

Most characters in J are specified by enclosing them in single quotes: ’a’, ’b’, ’ ’ (space), ’+’, ’{’ etc. Similarly, you give J a list of characters (a ‘text string’), by enclosing it in quotes:

```
'This is a text string'
This is a text string
'Within text, the single-quote (’) is indicated by double quotes (’’!’)'
Within text, the single-quote (’) is indicated by double quotes (’’)!
'Use ’’, ’’ (’append’) to join text strings together'
Use ’’, ’’ (’append’) to join text strings together
'Like’, ’ ’ This’’, ’!!!’
Like This!!!
```

As well as , (append), many other J verbs are useful for text manipulation. For example, (from) extracts specified characters from its right argument:

```
4 5 9 2 1 3 7 4 2 3 { 'qwertyuiop'
typewriter
5 _2 6 3 2 0 6 7 4 2 _1 _7 _8 _6 4 5 { 'qwertyuiop'
yourequitepretty
```

Note that J counts from 0, so here q is character 0, w is character 1, etc. This makes many algorithms much more natural, understandable and efficient [17]. Also note that _1 { y. is the last and _2 { y. the second last item in y., etc.
Dyadic i. (index of) is almost an inverse of dyadic }, as illustrated below:

```
'qweuryiop' i. 'typewriter'
4 5 9 2 1 3 7 4 2 3
'qweuryiop' i. 'yourequitepretty' NB. returns indices from start of x.
5 8 6 3 2 0 6 7 4 2 9 3 2 4 4 5
'qweuryiop' i. 'what if RHS omits some chars?'
1 10 10 4 10 7 10 10 10 10 8 10 7 4 10 10 10 10 3 10 10
```

Note that when evaluating x. i. y., a character in y. but not in x. is assigned the index [length of x.], i.e. [maximum possible index in x.] + 1.

All characters, including non-keyboard characters such as those for drawing boxes, can be extracted from J's full list a. For example, 10(a. produces a line-feed (new line in J), 179(a. draws a vertical line, and 196(a. draws a horizontal line:

```
218 194 196 191 10 196 196 180 10 179 179 65 179 10 192 193 196 217 { a.
```

```
+/ a. i. 'WINDOWS95' NB. remember J counts from 0 !?!
665
```

### 2.2.2 Numbers

J has a hierarchy of number types: *Boolean, integer, extended integer, rational, floating point* and *complex*. The result of combining two different types will have the type appearing later in this list. For example, adding the integer 2 to the floating point number 3.14159 results in the floating point number 5.14159. This is called *coercion* in some programming languages.

#### 2.2.2.1 Boolean

A *Boolean* (or *logical*) variable takes the value 1 representing ‘True’, or 0 representing ‘False’. For example, the following J dialogue verifies that \(12^3 + 1^3 = 10^3 + 9^3\):

```
(+/ 12 1 ^ 3) = +/ 10 9 ^ 3
1
```

J has all of the usual *relational functions*: < (less than), <= (less or equal), = (equal to), ~: (not equal), > (larger than), >: (larger or equal):

```
5 < 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9  NB. less than
0 0 0 0 0 1 0 1 0 0 0 1 1 1 1
5 <=: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9  NB. less than or equal to
0 0 0 0 1 1 0 1 1 0 1 1 1 1
5 =: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9  NB. equal to
0 0 0 0 1 0 0 0 1 0 1 0 0 0
5 -: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9  NB. not equal to
1 1 1 1 0 1 1 0 1 0 1 1 1 1
5 >: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9  NB. greater than or equal to
```
and the standard Boolean (or logical) functions -. (not), +. (or), *. (and), +: (not or), *: (not and):

- . 0 1
  1 0
  0 1 0 1 +. 0 0 1 1
  0 1 1 1
  0 1 0 1 *. 0 0 1 1
  0 0 0 1
  0 1 0 1 +: 0 0 1 1
  1 0 0 0
  0 1 0 1 *: 0 0 1 1
  1 1 1 0

Some more general arithmetic functions also have natural interpretations as logical functions when restricted to Boolean variables, for example:

0 1 0 1 ~.: 0 0 1 1
0 1 1 0
0 1 0 1 <: 0 0 1 1
1 0 1 1

Finally, note that some logical functions (=, -: can be applied to non-numeric nouns. However, others (e.g. <) don’t make sense, and you must also beware of mixing numeric and non-numeric arguments:

'a' = 'Canaan Banana'
0 1 0 1 1 0 0 0 1 0 1 0 1
'a' < 'Canaan Banana'
domain error
'a' < 'Canaan Banana'
5 = 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9
0 0 0 0 1 0 0 0 1 0 1 0 0 0
5 = '3 1 4 1 5 9 2 6 5 3 5 8 9 7 9'
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
5' = '3 1 4 1 5 9 2 6 5 3 5 8 9 7 9'
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

2.2.2.2 Integer

A number in J, as in most computer programming languages, has no imbedded spaces but can contain an e or E, which means ‘times ten to the power...’. Note however that in J a negative number begins with _ (negative sign—the underline character) rather than - (negate—the minus sign), which is a verb:

123 _123 12e5 12000e_2
123 _123 1200000 120
123 _123 12e5 12000e_2
0 _1199877 3

NB. a list of four integers
NB. 123 minus a list of three integers

NB. greater than
J has many monadic verbs for manipulating integers, including < (decrement), > (increment), * (signum), - (negate), | (magnitude) and ! (factorial):

```
<: 123 _123 _1 0 1 199999 200000 NB. decrement (by 1)
 122 _124 _2 _1 0 199998 199999
>: 123 _123 _1 0 1 199999 200000 NB. increment (by 1)
 124 _122 0 1 2 200000 200001
* 123 _123 _1 0 1 199999 200000 NB. signum (_1 0 1 for neg, zero, positive)
 1 _1 _1 _1 _1 _1
- 123 _123 _1 0 1 199999 200000 NB. negate
 _123 123 1 0 1 _1 _1 _1 _1 _1 _1 _1 _1 _1 _1 _1 _1
| 123 _123 _1 0 1 199999 200000 NB. magnitude (absolute value of)
123 123 1 0 1 199999 200000
! 1 2 6 24 120 720 5040 40320
! 20 100 NB. large factorials are coerced to floating point numbers
2.4329e18 9.33262e157
```

The verbs #. (base 2) and #: (antibase 2) provide links between integers and Booleans:

```
#: 123 NB. antibase 2 (i.e. base 2 representation of...)
 1 1 1 1 0 1 1
#: _123 NB. note representation of negative integers
 0 0 0 0 1 0 1
#: 123 _123 _1 0 1 199999 200000 NB. note how each row in result is padded
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1
 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 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 1 0 1 0 1 0 0 1 1 1 1 1
 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0
#. 1 1 1 1 0 1 1 NB. base 2 (i.e. interpret y. in base 2)
123
```

Dyadic verbs for manipulating integers include +. (GCD), * (LCM), | (residue—i.e. remainder after integer division of y. by x.), and ! (out of—i.e. x!y. is \binom{y}{x}\) in standard mathematical notation):

```
2299 +. 5225 NB. GCD (greatest common divisor)
209
2299 *. 5225 NB. LCM (least common multiple)
57475
10 | 123 _123 _1 0 1 199999 200000 NB. residue (on dividing y. by x.)
3 7 9 0 1 9 0
(i. 6) ! 5 NB. # ways of choosing x. out of y. (i.e. binomial coefficient)
1 5 10 10 5 1
```

Note how GCD and LCM are natural generalisations of the logical functions or & and respectively.
Dyadic #. (base) and #: (antibase) allow representation of integers in different bases, and have many applications including (perhaps surprisingly until you’ve verified the maths) polynomial evaluation:

\[
\begin{array}{l}
16 \#. \ 3 \ 14 \ 15 \ 9 \quad \text{NB. base 16 interpretation of } y.
\end{array}
\]

\[
\begin{array}{ll}
16121 & 16 \ 16 \ 16 \ 16 \#: \ 16121 \quad \text{NB. inverse of above operation}
\end{array}
\]

\[
\begin{array}{ll}
0 \ 3 \ 14 \ 15 \ 9 - 24 \ 60 \ 60 #: \ 1000000 \quad \text{NB. a million seconds = 11 days 13 hrs 46 mins 40 secs.}
\end{array}
\]

\[
\begin{array}{ll}
11 \ 13 \ 46 \ 40 & 1 \#. \ 1 \ 1 \ 41 \quad \text{NB. polynomial evaluation: } x^2 + x + 41 \text{ evaluated at } x = 1
\end{array}
\]

\[
\begin{array}{ll}
43 & 2 \#. \ 1 \ 1 \ 41 \quad \text{NB. } x^2 + x + 41 \text{ evaluated at } x = 2
\end{array}
\]

\[
\begin{array}{ll}
47 & 3 \#. \ 1 \ 1 \ 41 \quad \text{NB. } x^2 + x + 41 \text{ evaluated at } x = 3
\end{array}
\]

Note the use of an isolated _ to represent the noun infinity. Similarly __ represents minus infinity and __. represents indeterminate.

Other useful integer functions include p: (prime) and q: (prime factors). The following dialogue also illustrates the use of the verb ] (same or right), which returns its right argument. Here it can be read as ‘do the following, and print out the result’:

\[
\begin{array}{l}
] p= .. p: \ 1393 + i. \ 11 \quad \text{NB. 1393rd, 1394th, ... 1404th prime}
\end{array}
\]

\[
\begin{array}{l}
11587 \ 11593 \ 11597 \ 11617 \ 11621 \ 11633 \ 11657 \ 11677 \ 11681 \ 11689 \ 11699
\end{array}
\]

\[
\begin{array}{l}
4 \ p \quad \text{NB. 9 consecutive primes are of form } 4n+1, \text{ starting with 11593}
\end{array}
\]

\[
\begin{array}{l}
3 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 3
\end{array}
\]

\[
\begin{array}{l}
] y= .. 199 + 210 * i. \ 12 \quad \text{NB. arithmetic progression}
\end{array}
\]

\[
\begin{array}{l}
199 \ 409 \ 619 \ 829 \ 1039 \ 1249 \ 1459 \ 1669 \ 1879 \ 2089 \ 2299 \ 2509
\end{array}
\]

\[
\begin{array}{l}
\quad \text{NB. indices of } y \text{ in first 1000 primes}
\end{array}
\]

\[
\begin{array}{l}
45 \ 79 \ 113 \ 144 \ 174 \ 203 \ 231 \ 262 \ 288 \ 315 \ 1000 \ 1000
\end{array}
\]

\[
\begin{array}{l}
\quad \text{NB. i.e. 199 to 2089 are prime, 2299 and 2509 are composite}
\end{array}
\]

\[
\begin{array}{l}
\quad \text{NB. prime factors of 2299}
\end{array}
\]

\[
\begin{array}{l}
11 \ 11 \ 19
\end{array}
\]

\[
\begin{array}{l}
\quad \text{__ q: } 2299 \ 2509 \quad \text{NB. __ q: y. returns prime factors, multiplicities}
\end{array}
\]

\[
\begin{array}{l}
11 \ 19
\end{array}
\]

\[
\begin{array}{l}
2 \ 1
\end{array}
\]

\[
\begin{array}{l}
13 \ 193
\end{array}
\]

\[
\begin{array}{l}
1 \ 1
\end{array}
\]

15
2.2.2.3 Extended integer

By default, J will store large integers as floating point numbers, hence losing some precision. For most applications this is the appropriate action. However, J can also store and manipulate large integers exactly as 'extended integers', which may be useful in number theory or for certain precise evaluations.

Extended integers are given to J by appending an x (e.g. 1234567890123456789x); J prints extended integers out normally, without the appended x:

```
] a =. 1234567890123456789x
1234567890123456789
>: a NB. verbs that transform integers to integers work correctly
1234567890123456790
(10^x30) - a NB. 10^x30 is evaluated precisely since 30 is coerced to 30x
99999999999765432109876543211
q: a NB. prime factorisation of a
3 3 101 3541 3607 3803 27961
q: >: 2^x32 NB. Fermat wrongly hypothesised that 1 + 2^x32 is prime
641 6700417
341 | 2^x340x NB. residue (341 = 11*31 is a 'pseudoprime in base 2')
1
] b =. 2^x100
1267650600228229401496703205376
b +. 150x NB. GCD of two extended integers
140737488355328
2^x47
140737488355328
```

The verb x: (extend) creates extended precision integers from ordinary integers:

```
factors=. 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
multiplicities=. 47 22 12 8 4 3 2 2 2 1 1 1 1 1 1
*/ factors ` multiplicities NB. */ can be read as 'product over'
3.04141e64
*/ factors ` x: multiplicities NB. extended precision calculation
3041409320171337804361260816606476884437764156896051200000000000
```

J allows various other manipulations of extended integers. For example, the constructions <.@f (integer part of result of f) and >.@f (smallest integer not less than result of f) are closed on the extended integers, even for verbs f like %, o. and #: that generally return non-integer results:

```
(10^-16x) <.@% 89 98 997 998 1089 NB. integer part of (10^-16) % 89 98...
112359550561797 102040816326530 10030090270812 10020040080160 9182736455463
<.@. o. 10^x50 NB. 10^50 times pi
314159265358979323846264338327950288419716939937510
<.@%. ,. 2 3 4 5 6 7*10^x100 NB. 10^50 times square roots
14142135623730950480168872429096807856967187537694
173205080756887729352744634150587236694280525381.038
20000000000000000000000000000000000000000000000000000
2236067979499768869409173667873127623644061836961152
24494897427317808619728240740589139165947480865867
264575131106459059505016157536392604257102691830825
```
2.2.2.4 Rational

Rational numbers are represented as \([\text{numerator}] : [\text{denominator}]\), and are stored with positive denominator and with relatively prime numerator and denominator. Arithmetic on rationals is carried out to arbitrary precision:

\[
\begin{align*}
1r12 _4r123 & \ 16r1234 _64r12345 256r123456 & \text{NB. } J \text{ simplifies these} \\
1r12 _4r123 & \ 8r617 64r12345 4r1929 \\
1r20 * 2r19 & \ 3r18 * 4r17 \ 5r16 \ 6r15 & \text{NB. equivalent to } \% 6!20 \\
1r35760 & \text{ NB. roughly } \sqrt{2} \\
17677669526636851r12500000000000000000000 & \\
x^2 & 312499999999999992536822873553721r1562500000000000000000000000000000
\end{align*}
\]

The verb \(x:\) (extend) converts irrationals to rationals, with an inbuilt tolerance so that (for example) the floating point (inexact) representation of \(1/3\) is sensibly converted to \(1r3\). The inverse of extend, \(x:\ ^{-1}\), can be used to convert extended integers and rationals into floating point numbers. This is an example of the power conjunction, see Section 2.3.6.2. Finally, the dyadic form \(2 \times y\): returns separately the numerator and denominator of the rational approximation to \(y\).

\[
x: \begin{align*}
0.25 & 0.2 \ 0.3333333 \ 0.3333333 \ 0.3333333 \\
1r4 & \ 1r5 \ 3333333r1000000 \ 3333333r10000000 \ 1r3 \\
+/ x: >: i. 100 & \text{NB. sum of reciprocals from 1 to 100} \\
14466636279520351160221518043104131447711r278881509188499086581352357412492142272 & \\
+/ x: % >: i. 100 & \text{NB. this is also exact because of inbuilt tolerance} \\
14466636279520351160221518043104131447711r278881509188499086581352357412492142272 & \\
x: i1 1p1 1p4 & \text{NB. rational approximations to } e, \pi, \pi^{-4} \\
6157974361033r2265392166685 & 1285290289249r409120605684 & 24135758477270r247777268231 & \\
e: \% ! i. 20 & \text{NB. sum of reciprocals of factorial } 0, 1, \ldots, 19 \\
82666416490601r30411275102208 & \\
x: ^{-1} \ e & \text{NB. floating point representation of above} \\
2.71828 & \\
2 \times i1 1p1 1p4 & \text{NB. numerators, demoms. of approximations to } e, \pi, \pi^{-4} \\
6157974361033 & 2265392166685 & 1285290289249 & 409120605684 & \\
24135758477270 & 247777268231 & \\
\end{align*}
\]

2.2.2.5 Floating point

Floating point representation is used to store real non-integer values, to an accuracy of roughly 1 part in \(10^{15}\) in the MS Windows implementation of J. Note that a number cannot start with a dot: 0.618 is a valid number but .618 isn’t. Thus \(3f.3\ 1\ 4\ 1\ 5\ 9\) parses as \(3f.3\ 1\ 4\ 1\ 5\ 9\), not \(3f.3\ 1\ 4\ 1\ 5\ 9\). Also note that, as well as the common notation \(e\) or \(E\) meaning ‘times ten to the power...’, J allows \(p\) meaning ‘times \(\pi\) to the power...’, and \(x\) meaning ‘times \(e\) to the power...’:

\[
\begin{align*}
1e1 & 2e1 \ _{100}e1 \ 1e2 \ 1e1 & \text{NB. } 10, \ 2\times10, \ _{100}\times10, \ 10^{-2}, \ 1/10 \\
10 & 20 \ _{1000}100 \ 0 \ 1. \ 1p1 & 2p1 \ _{100}p1 \ 1p2 \ 1p1 & \text{NB. } \pi, \ 2\pi, \ _{100}\pi, \ \pi^{-2}, \ 1/\pi \\
3.14159 & 6.28319 \ _{314}159 \ 9.8696 \ 0.3183099 \\
1x & 2x \ _{100}x1 \ 1x2 \ 1x1 & \text{NB. } e, \ 2e, \ _{100}e, \ e^{-2}, \ 1/e \\
2.71828 & 5.43666 \ _{271}828 \ 7.38906 \ 0.3678794 \\
1r3p1 & 22r7p1 \ _{113}355p1 & \text{NB. } \pi/3, (22/7)/\pi, (113/355)\pi \\
1.0472 & 1.0004 \ 0.9999999 \\
\end{align*}
\]
J's default printing precision is about 6 significant figures (you can alter this—see 9.11 in Section 3.2).

Floating point numbers in J can be converted to integers using the monadic verbs <. (floor) and >. (ceiling). Floating point numbers will also be automatically rounded to integers if they are sufficiently close

```j
] v=. _123 _1p1 _1.5 0 1 1x1 1p1 1e1
_123 _3.14159 _1.5 0 1 2.71828 3.14159 10
<. v          NB. floor (i.e. integer part) of v
_123 _4 _2 0 1 2 3 10
> v
NB. ceiling of v (i.e. least integer at least as big as v)
_123 _3 _1 0 1 3 4 10
>. v - 0.5     NB. round v to nearest integer
_123 _3 _2 0 1 3 3 10
isinteger<. = >.    NB. floor = ceiling iff y. is an integer
isinteger v
1 0 0 1 1 0 0 1
] x=. 1 + 10^-i. 20     NB. default printing precision is about 6 sig. figs.
2 1 1 1.01 1.001 1.0001 1.00001 1 1 1 1 1 1 1 1
isinteger x
1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
```

Many of the monadic verbs already introduced, such as <:, -. and !, generalise naturally to non-integer arguments. Other important monadic verbs for manipulating floating point numbers include +: (double), -: (halve), *: (square), %: (square root), ^ (exponential) and ^.* (natural log):

```j
] v=. _123 _1p1 _1.5 0 1 1x1 1p1 1e1
_123 _3.14159 _1.5 0 1 2.71828 3.14159 10
< v          NB. decrement v (by 1)
_123 _4.14159 _2.5 _1 0 1.71828 2.14159 9
> v
NB. increment v (by 1)
_123 _2.14159 _0.5 1 2 3.71828 4.14159 11
~ v
NB. negate v
123 3.14159 1.5 _1 _2.71828 _3.14159 _10
* v
NB. signum v
_1 _1 _1 0 1 1 1 1
| v
NB. magnitude of v
123 3.14159 1.5 0 1 2.71828 3.14159 10
+: v
NB. double v
_246 _6.28319 _3 _3 _2.5 3.14656 6.28319 20
*: v
NB. square v
151299.8969 2.25 _5 1 7.38396 9.8968 100
~ v
NB. 1-v (generalises logical NOT)
124 4.14159 2.5 1 0 _1.71828 _2.14159 _9
~: v
NB. halve v
_61.5 _1.5708 _0.75 0.5 0.5 0.5 0.5 0.5 0.5
^ v
NB. reciprocal of v (note that 1/0 is infinite)
0.008130081 _0.3183099 _0.6666667 _1 0.3678794 0.3183099 0.1
^ v
NB. exp(v)
3.8175e54 0.04321392 0.2231302 1 2.71828 15.1543 23.1407 22026.5
o. v
NB. pi times v
^: v 0.5 1 1.5 2
NB. square root
0.7071068 1 1.22474 1.41421
^: 0.5 1 1.5 2
NB. natural log
_0.6931472 0.4054651 0.6931472
^! 0.5 1 1.5 2
NB. factorial 0.5 1 1.5 2 (i.e. gamma 1.5 2 2.5 3)
0.8862269 1 1.32934 2
```
Commonly used dyadic verbs include < (lesser of), > (larger of), ~ (log), % (root), and ^ (power), as well as the elementary arithmetic functions (+, -, *, ^), Boolean functions (,<, =: etc.) and others such as *, and #: whose definitions generalise naturally from integer arguments to floating point:

```
] v=._123 _1p1 _1.5 1 1x1 1p1 ge1
_123  3.14159 _1.5  1  2.71828 3.14159 10
  v = 355r113  NB. subtract 355/113 (excellent approximation to pi)
_126.142 _6.28319  _4.64159 _3.14159 _2.14159  _0.4233111  _2.66764e_7  6.85841
  v % 355r113  NB. v divided by 355/113
  _39.1521 _0.9999999 _0.4771268 0 0.3183099 0.8652559 0.9999999 3.1831
  0 < v  NB. is 0 less than v? (1 if true, 0 if false)
  0 0 0 0 1 1 1
  0 <=: v  NB. is 0 less than or equal to v? (1 if true, 0 if false)
  0 0 0 1 1 1 1
  0 <. v  NB. lesser of 0 and v
_123 _3.14159 _1.5 0 0 0 0
_0 >. v  NB. larger of 0 and v
  0 0 0 0 1 2.71828 3.14159 10
  10 ^._1 2 3 10 1000 5000  NB. logs to base 10
  0 0.30103 0.4771268 1 3 3.69997
  5 3 2 1 0.5 1r3 _1 _2 %: 100  NB. fifth root, cube root etc. of 100
  2.5189 4.64159 10 100 10000 1e6 0.01 0.1
  100 ^ 1r5 1r3 0.5 1 2 3 _1 _0.5 0.30103  NB. 100 to the power...
  2.5189 4.64159 10 100 10000 1e6 0.01 0.1 4
  123.45 *: 100  NB. least +ve integer that's an integer multiple of x. and y.
  246900
_1 #: _1p1 100p1 10000p1  NB. integer, fractional parts of pi, 100 pi, 10000 pi
  3 0.1415927
  314 0.1592654
  31415926536
_1 1760 3 12 2.54 #: 10^-9  NB. 10^-9cm = 6213 miles, 1252yd, 2ft, 11in + 1.02cm
  6213 1252 2 11 1.02
```

### 2.2.2.6 Complex

Complex numbers are represented in J by [real part]j[imaginary part]. For example, the following dialogue demonstrates that $\sqrt{-1} = i$, $1 + e^{i\pi} = 0$, $\log(i) = i\pi/2$, and that $-\frac{1}{2} + i\frac{\sqrt{3}}{2}$ is a complex cube root of 1:

```
%: _1
0j1
  1 + ^0j1p1
0
  ^ _0j1
0j1.5708
  w =. _0.5 + %: _0.75
  w ^ 1.23
_0.5j0.8660254 _0.5j0.8660254 1
```

Details of complex number arithmetic are given in the J Introduction and Dictionary[15], including the use of the monadic verbs + (conjugate), +. (real/imaginary), * (length/angle), j. (imaginary) & r. (angle), and the dyadic verbs j. (complex) & r. (polar).
2.2.3 Boxes

Any J object can be put in a box by monadic < (box), and lists (shelves?) of boxes created using dyadic ; (link), as illustrated on page 11. Verbs such as { and i. are as useful with lists of boxes as with lists of characters:

```
] scale =. 'A';'Bb';'B';'C';'Db';'D';'Eb';'E';'F';'F#';'G';'Ab'
A Bb B C Db D Eb E F F# G Ab
2 0 1 0 10 7 { scale
B A Bb AGE
scale i. <'Db'
4
] bluesE=. 'E';'E';'E';'A';'A';'E';'E';'B';'A';'E';'B'
E E E A A E E B A E B
scale i. bluesE
7 7 7 0 0 7 7 2 0 7 2
] bluesG=. (3 + scale i. bluesE) { scale  NB. transpose 3 semitones
G G G C C G D C G D
```

2.2.4 Arrays

Arrays are familiar in mathematics: a matrix is a 2-dimensional array, and a vector is a 1-dimensional array. To avoid confusion and possible ambiguity, rather than call a matrix a ‘2-dimensional array’ with rows as ‘dimension 1’ and columns as ‘dimension 2’, J refers to a matrix as a rank-2 array with rows as axis 1 and columns as axis 2.

A rank-1 array (vector) may also be called a list, and a rank-2 array (matrix) a table. Arrays of higher rank are similarly important, but usually don’t have particular names (except in statistics as ‘N-way tables’). A scalar or atom (i.e. a single number such as 163 or π, or in J any single character, number or box) is a rank-0 array.

J has many primitive verbs to create and manipulate arrays. An example of a verb that creates arrays is monadic i. (integers), which returns successive integers, starting from zero, filling up the shape specified by its argument:

```
i. 10   NB. a list of the first 10 non-negative integers
0 1 2 3 4 5 6 7 8 9
i. 1   NB. a list of length 1, containing just 0
0
i. 4 10 NB. a '4 by 10' matrix
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
```
\[ 1 \quad 2 \quad 3 \quad 8 \quad \text{NB. a 3-dimensional array} \\
0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \\
8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \\
16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 21 \quad 22 \quad 23 \\
24 \quad 25 \quad 26 \quad 27 \quad 28 \quad 29 \quad 30 \quad 31 \\
32 \quad 33 \quad 34 \quad 35 \quad 36 \quad 37 \quad 38 \quad 39 \\
40 \quad 41 \quad 42 \quad 43 \quad 44 \quad 45 \quad 46 \quad 47 \\
i. \quad 0 \quad \text{NB. an empty list} \\
\text{NB. J responded with a blank line!} \\
\]

The effect of \textit{i}. is conveniently illustrated using the verb \texttt{xrs}, which displays the results of several J expressions simultaneously, together with the shapes of the results (the way \texttt{xrs} works will be explained later in Section 2.4):

\[
\texttt{xrs=} \{, \emptyset ; \{ \text{do each} \} \} \quad \text{NB. expression, result, shape of result} \\
xrs 'i. \; 0'; 'i. \; 1'; 'i. \; 4'; 'i. \; 4 \; 3'; 'i. \; 1 \; 1'; 'i. \; 2 \; 3 \; 4'; 'i. \; 2 \; 3 \; 1 \; 4' \\
\]

<table>
<thead>
<tr>
<th>i. 0</th>
<th>i. 1</th>
<th>i. 4</th>
<th>i. 4 3</th>
<th>i. 1 1</th>
<th>i. 2 3 4</th>
<th>i. 2 3 1 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Comments:**

1. Note the way that J displays arrays: \textit{i. 4} shows a list of 4 numbers, \textit{i. 4 3} shows 4 successive 1-dimensional arrays, \textit{i. 2 3 4} shows 2 successive 2-dimensional arrays (separated by 1 space), \textit{i. 2 3 1 4} shows 2 successive 3-dimensional arrays (separated by 2 spaces), etc.

2. Although the results of \textit{i. 1} and \textit{i. 1 1} are displayed the same way, they are in fact different, as demonstrated by their shapes (\textit{1} and \textit{1 1} respectively).

3. Counting in J starts from 0 rather than from 1.

The first number in the shape of an array is the number of \textit{items} in the array. Thus \textit{i. 4} has 4 items each of which is an integer; \textit{i. 4 3} has 4 items, each being a list of 3 integers; \textit{i. 2 3 4} has 2 items, each being a \((3 \times 4)\) matrix, etc.
If \( A \) is an array with shape 2 3 4, then each of its 2 items is a subarray of rank 2, called a 2-cell, and has shape 3 4. Each item of a 2-cell is a subarray of rank 1, and is called a 1-cell, etc. Thus \( A \) may be thought of as a frame of shape 2 containing 2-cells of shape 3 4, or a frame of shape 2 3 containing 1-cells of shape 4, or a frame of shape 2 3 4 containing 0-cells (atoms), or finally as an empty frame containing one 3-cell of shape 2 3 4.

In general, if \( A \) is a rank-\( n \) array with shape \( S \), then, for each \( k = 0,1,\ldots,n \), \( A \) can be thought of as a frame of rank \( n - k \) (whose shape is the first \( n - k \) numbers of \( S \)), containing \( k \)-cells whose shape is the last \( k \) numbers \( S \). In particular, the items of \( A \) are its \((n - 1)\)-cells.

### 2.2.4.1 Array Shape (# and $)

Monadic \# (tally) returns the number of items in its argument, and monadic $ (shape of) returns the whole shape of its argument:

\[
\begin{array}{c}
\text{s=} 1 p 1 \ [ \text{v=} i . 3 \ [ \text{m=} i . 4 \ [ \text{A=} i . 2 3 4} \\
\text{xrs cut s v m A # s # v # m # A $ s $ v $ m $ A $ s s $ v $ m $ A $ s s s v m s s m s s m}}
\end{array}
\]

<table>
<thead>
<tr>
<th>s</th>
<th>v</th>
<th>m</th>
<th>A</th>
<th># s</th>
<th># v</th>
<th># m</th>
<th># A</th>
<th>$ s</th>
<th>$ v</th>
<th>$ m</th>
<th>$ A</th>
<th>$$ s</th>
<th>$$ v</th>
<th>$$ m</th>
<th>$$ A</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
<td>0 1 2</td>
<td>0 1</td>
<td>0 1 2 3</td>
<td>1</td>
<td>3 4 2</td>
<td>3 4 2</td>
<td>2 3 4</td>
<td>0 1 2 3 4 0</td>
<td>1 2 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 3 4</td>
<td>4 5</td>
<td>4 5 6 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 5 6 7</td>
<td>8 9</td>
<td>8 9 10 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12 13 14 15</td>
<td>16 17</td>
<td>16 17 18 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 17 18 19</td>
<td>20 21</td>
<td>20 21 22 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20 21 22 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Comments:**

1. \([ \text{left} \) is a dyadic verb that returns its left argument \( x \). In the above dialogue it can be read as a statement separator.

2. \text{cut} is a J utility (Section 4.1.1) that boxes chunks of text; by default those chunks between spaces.

3. Note that \# \( y \) returns a scalar but \$ \( y \) returns a vector (i.e. a list). In particular, if \( v \) is a list (e.g. \( v = i . 3 \) as above), then \# \( v \) and \$ \( v \) may look the same but are in fact different. Both display 3, but \# \( v \) is a scalar whereas \$ \( v \) is a list of length 1.

4. Note that the length of the list \$ \( y \) is the rank of \( y \), and may be obtained directly by \#\$ \( y \). (or, as a vector of length 1, by \$\$ \( y \)).

5. A scalar like \( 1 p 1 (\pi) \) has rank 0.
Dyadic # (copy) and $ (shape) are also useful verbs: $x$. $y$. returns $x$. copies of each item of $y$. , and $x$. $y$. returns a noun with $x$. items, copying the list of items of $y$. and repeating the list if necessary:

\[
\begin{align*}
\text{v} &= . i . 4 \ [' m' = . i . 2 . 4 \\
\text{xrs} &= 'v'; '3#v'; '2 3$v'; 'm'; '3#m'; '3$m'; '2 3$m'$
\end{align*}
\]

<table>
<thead>
<tr>
<th>v</th>
<th>3#v</th>
<th>2 3$v$ m</th>
<th>3#m</th>
<th>3$m$</th>
<th>2 3$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 0 0 1 1 1 2 2 2 3 3 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 0 1 4 5 6 7</td>
<td></td>
<td></td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>2 3 2 4 6 4 3 4</td>
<td>2 3 4</td>
<td>2 3 4</td>
<td>2 3 4</td>
</tr>
</tbody>
</table>

2.2.4.2 Reordering Axes (| and |:)

The expression $\text{v}$. $\text{y}$. (reverse $\text{y}$.) returns $\text{y}$. with the order of its items reversed, and $\text{x}$. $\text{y}$. ($\text{x}$. rotate $\text{y}$.) returns $\text{y}$. with its list of items rotated leftwards $\text{x}$. times:

\[
\begin{align*}
\text{v} &= . \text{abcde} \ [' m' = . i . 5 . 3 \\
\text{xrs} &= \text{cut} 'v m | .v 2|v 42|v _2|v | .m 2|m 42|m _2|m'$
\end{align*}
\]

| v  | m  | |v  | 2|v  | 42|v  | _2|v  | |m  | 2|m  | 42|m  | _2|m  |
|----|----|---|-----|----|-----|-----|----|-----|----|-----|-----|-----|-----|
| abcde | 0 1 2 | edcba | cdeab | cdeab | deabc | 12 13 14 | 9 10 11 | 6 7 8 | 12 13 14 | 6 7 8 | 9 10 11 | 12 13 14 |
|       | 3 4 5 |       |       |      |       | 9 10 11 | 9 10 11 | 12 13 14 | 12 13 14 | 0 1 2 | 3 4 5 | 3 4 5 |
|       | 6 7 8 |       |       |      |       | 12 13 14 | 12 13 14 | 0 1 2 | 3 4 5 | 3 4 5 | 6 7 8 |       |
|       | 9 10 11 |      |       |      |       | 6 7 8 | 9 10 11 | 12 13 14 | 0 1 2 | 3 4 5 | 3 4 5 | 6 7 8 |       |
|       | 12 13 14 |     |       |      |       | 9 10 11 | 12 13 14 | 0 1 2 | 3 4 5 | 3 4 5 | 6 7 8 |       |
Monadic \( \lbrack \) (transpose) reverses the order of the axes of \( y \). (matrix transpose, where \( y \) has rank 2, is just a special case). Dyadic \( \lbrack \) reorders the axes of \( y \), so that the \( x \).th axes come last, as illustrated below:

\[
xrs '1234'; '\lbrack :1234 \rbrack'; '0\lbrack :1234 \rbrack'; '1\lbrack :1234 \rbrack'; '0 1\lbrack :1234 \rbrack'; '0 2\lbrack :1234 \rbrack'
\]

<table>
<thead>
<tr>
<th>1234</th>
<th>:1234</th>
<th>0:1234</th>
<th>1:1234</th>
<th>0 1:1234</th>
<th>0 2:1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 12</td>
<td>0 12</td>
<td>0 4 8</td>
<td>0 4 8</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4 16</td>
<td>1 13</td>
<td>1 5 9</td>
<td>12 16 20</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>8 20</td>
<td>2 14</td>
<td>2 6 10</td>
<td>3 7 11</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>1 13</td>
<td>5 17</td>
<td>4 16</td>
<td>12 16 20</td>
<td>13 17 21</td>
</tr>
<tr>
<td>16 17 18 19</td>
<td>5 17</td>
<td>6 18</td>
<td>7 19</td>
<td>14 18 22</td>
<td>14 18 22</td>
</tr>
<tr>
<td>20 21 22 23</td>
<td>2 14</td>
<td>6 18</td>
<td>10 22</td>
<td>3 7 11</td>
<td>15 19 23</td>
</tr>
<tr>
<td>2 3 4</td>
<td>11 23</td>
<td>4 3 2</td>
<td>3 4 2</td>
<td>4 2 3</td>
<td>3 2 4</td>
</tr>
</tbody>
</table>

With a boxed left argument, dyadic \( \lbrack \) runs together the axes specified in \( x \). (imagine diagonal lines, planes etc. cutting through \( y \).):

\[
xrs '1234'; '<\lbrack 0 \rbrack :1234 \rbrack'; '<\lbrack 0 2 \rbrack :1234 \rbrack'; '<\lbrack 1 2 \rbrack :1234 \rbrack'; '<\lbrack 0 1 2 \rbrack :1234 \rbrack'
\]

<table>
<thead>
<tr>
<th>1234</th>
<th>&lt;\lbrack 0 \rbrack :1234</th>
<th>&lt;\lbrack 0 2 \rbrack :1234</th>
<th>&lt;\lbrack 1 2 \rbrack :1234</th>
<th>&lt;\lbrack 0 1 2 \rbrack :1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 16</td>
<td>0 13</td>
<td>0 5 10</td>
<td>0 17</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>1 17</td>
<td>4 17</td>
<td>12 17 22</td>
<td></td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>2 18</td>
<td>8 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>3 19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 17 18 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 21 22 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 4</td>
<td>4 2</td>
<td>3 2</td>
<td>2 3</td>
<td>2</td>
</tr>
</tbody>
</table>
2.2.4.3 Combining and Reshaping Arrays (, , , : and ;)

There are several J verbs to combine arrays in various ways, e.g., dyadic ; (link), (append) ,, (stitch) and ,: (laminate), whose effects are illustrated by the following dialogue:

```
i234=. i. 2 3 4
'x y'=. (97+i234){a.
xrs 'i234' ; 'x' ; 'y' ; 'x;y' ; 'x,y' ; 'x.,'y' ; 'x;:y'
```

```
i234   x   y    x;y    x,y    x.,y    x;:y
      0 1 2 3  abcd  mnop  abcd  abcd
      4 5 6 7  efgh  qrst   efgh   efgh
      8 9 10 11 ijkl  uvwx  ijkli  ijkli
    12 13 14 15
    16 17 18 19
    20 21 22 23
    2 3 4   3 4   3 4  2  6 4  3 8  2 3 4
```

Monadic , (ravel), ,, (ravel items), : (itemize) and ; (raze) are also useful. In particular, , y, returns a vector of the 0-cells of y.

```
a=. 2 2 2 $ 'abcdefh'
b=. 2 2 $ 1 ; (i. 2) ; (i. 3) ; 4
    xrs 'a'; 'a'; 'a'; 'a'; 'a'; 'b'; ; 'b'; ; 'b'; ; 'b'
```

```
a    ,a  ,a  ,a ;a   b    ,b  ,b  ;b
  ab   abcdefgh abcd  abcd  abcd  1 0 1 0 1 2 4
    cd  efgh    abcd  abcd  abcd  1 0 1 0 1 2 4
    ef    gh    0 1 2 4 1 0 1 0 1 2 4
    2 2 8   2 4 1 2 2 2 8 2 2 4 1 2 2 5 3
```

25
2.2.4.4 Subarrays ({ }, { }: { and })

The verbs {. (head—take), {(behead—drop), { (tail) and }; (curtail) variously return items from the
beginning or end of their right argument:

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
a & {.a} & {.a}.a & {.3.a} & {.3.a}.a & {.3.3.a} & {.3.3.3.a} & {.3.3.3.3.a} \\
\hline
0 & 1 & 14 & 5 & 0 & 1 & 0 & 1 \\
2 & 3 & 4 & 5 & 2 & 3 & 2 & 3 \\
4 & 5 & 6 & 7 & 4 & 5 & 4 & 5 \\
6 & 7 & 8 & 9 & 6 & 7 & 6 & 7 \\
8 & 9 & 10 & 11 & 8 & 9 & 8 & 9 \\
10 & 11 & 12 & 13 & 10 & 11 & 10 & 11 \\
12 & 13 & 14 & 15 & 12 & 13 & 12 & 13 \\
\hline
\end{array}
\]

The powerful dyadic verb { (from) extracts portions of an array. In its simplest form, { selects items x. from
array y. The following illustrates part of {’s syntax, making use of the rank conjunction " (Section 2.3.4.1).

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
c & {1c} & {2 0{1c}} & {1c"2c} & {(<1;0)2c} & {(<1;0;2;1)2c} \\
\hline
abcd & mnop & ge & ijk & uvw & om & sq \\
efgh & qrst & u & vi & w & ki & wu \\
ijkl & m & n & o & p & q & r & s \\
\hline
\end{array}
\]

2 3 4 3 4 2 3 2 2 4 2 4 2 2
The frighteningly powerful adverb \{ amend\} combines with its right argument to produce a dyadic verb, which then returns modified versions of arrays. The right argument to \} is usually in practice a noun, whose interpretation is similar to that of the right argument of \{, for example:

\[
\begin{array}{c|cc|cc|cc}
| a | \quad \text{'x'} \quad 1 \quad \text{a} | \quad \text{'xy'} \quad 2 \quad 0 \quad \text{1} \quad \text{a} | \quad \text{'x'} \quad \_1 \quad \text{1} \quad \text{a} | \quad \text{'x'} \quad \text{ (<1;0 2)} \quad \text{a} |
|-----------------|-----------------|-----------------|-----------------|
| 2 3 4 2 3 4 | 2 3 4 | 2 3 4 | 2 3 4 |
\end{array}
\]

Thus 1} is a verb: \text{x.} \text{ 1} \text{ y.} returns an array identical to \text{y.} except that item number 1 has been replaced by \text{x.}. If you actually want to modify \text{y.} then you must use a J expression like \text{y.=} \text{x.} \text{ 1} \text{ y.}, e.g.:

\[
a=. \quad 4 \quad 10 \quad $ \quad '-'
\]
\[
a
\]
\[
\text{--------}
\]
\[
\text{--------}
\]
\[
\text{--------}
\]
\[
\text{--------}
\]
\[
\text{--------}
\]
\[
\text{--------}
\]
\[
\text{--------}
\]

See the J documentation for further details of amend.

Although J's syntax for modifying arrays may seem convoluted, it turns out to have important advantages over the methods used in most programming languages (typically of the form 'A[indices] \leftarrow values'):

- J's notation for amend is consistent with that for other operations,
- the basic amendment syntax outlined above is easy to generalise, as described in the J Introduction and Dictionary[15].
2.2.4.5 Padding

Finally, note that J verbs that return arrays will, when necessary, automatically pad them out using 0 (zero), ' ' (space), or a: (ace—an empty box), as appropriate:

```
boxes=. 1;2 3 4
xrs '5{.42'; '5{.'foo'''; '5{.<'foo'''; 'boxes'; '>boxes'; '>::eh what??'

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>42 0 0 0 0</td>
<td>foo</td>
<td>[</td>
<td>1 2 3 4</td>
<td>boxes</td>
<td>&gt;boxes</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>20</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
```

Here the monadic verb > (open) opens its argument. If its argument isn’t a box or array of boxes, then > has no effect.

2.2.5 Data Structures

Arrays and boxes combine to give a flexible and powerful means of storing arbitrarily complicated data: Section 4.3 gives examples of database manipulation using J. As a simple example, the following might be part of a database record, and comprises two boxes:

```
Monster Movie

<table>
<thead>
<tr>
<th>Father cannot yell</th>
<th>7 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary, Mary so contrary</td>
<td>6 16</td>
</tr>
<tr>
<td>Outside my door</td>
<td>4 6</td>
</tr>
<tr>
<td>Yoo Doo Right</td>
<td>20 14</td>
</tr>
</tbody>
</table>
```

1. The first box contains the list of characters Monster Movie.

2. The second box contains a 4 × 2 array of boxes. Each of the four boxes in the first column contains a list of characters (not necessarily the same length each time). Each of the four boxes in the second column contains a list of two numbers.

If you are familiar with programming in ‘C’ [17], or a similar language, then you may prefer to think of the record as comprising two pointers: the first points to a string of characters, the second points to a list of pointers... etc.
The verb {:: (called map if monadic and fetch if dyadic), and the conjunctions L: (level) and S: (spread), are useful for manipulating complicated data structures:

\[
\begin{array}{c|c|c}
| & 0 & 1 2 0 1 2 3 4 5 \\
\hline
0 & 0 & 0 1 0 1 2 0 1 2 3 \\
\end{array}
\]

**:L:0 a
NB. square all leaves

\[
\begin{array}{c|c|c}
| & 0 & 1 4 0 1 4 9 16 25 \\
\hline
0 & 0 & 0 1 0 1 4 0 1 4 9 \\
\end{array}
\]

,\L:1 a
NB. ravel items at level 1 (‘stack innermost boxes in 2-D’)

\[
\begin{array}{c|c|c}
| & 0 & 1 2 3 4 5 \\
\hline
0 & 0 & 0 1 0 1 2 3 \\
\end{array}
\]

{:: a
NB. paths to each leaf

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
| & | & | & | & | & | & | \\
\hline
0 & 0 & 1 0 & 1 1 & 2 0 & 2 1 & 2 2 & 2 3 \\
\end{array}
\]

(2;1) {:: a
NB. what’s in box number 2, subbox 1?

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
| & | & | & | & | & | & | \\
\hline
0 & 1 & 2 0 1 2 3 4 5 & 0 & 0 1 1 2 3 \\
\end{array}
\]

\[
\text{pathleaves}:= \{:: ; S: 0 \}
\]

\[
\text{|:: pathleaves a}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
| & | & | & | & | & | & | \\
\hline
0 & 0 1 0 1 1 2 0 2 1 2 2 2 3 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
| & | & | & | & | & | & | \\
\hline
0 & 0 1 2 0 1 2 3 4 5 & 0 & 0 1 0 1 2 3 \\
\end{array}
\]

L. a

2
CHAPTER 2. THE J LANGUAGE

2.2. DATA

Comments:

1. The syntax of dyadic {:: (fetch) is similar to that of { from).

2. Levels are counted from the inside outwards. Thus L:0 (leaf) refers to the contents of the innermost boxes, L:1 refers to the innermost boxes themselves, and L:1 refers to the outermost boxes.

3. The expression ; S:1 0, used above in defining the verb pathleaves, applies ; (link) dyadically, at levels 1 (for the left argument) and 0 (for the right argument). Thus the innermost boxes of the left argument (unopened) are linked to the contents of the innermost boxes of the right argument.

4. The monadic verb L. (level) gives the maximum nesting of the boxes in its argument.
2.3 J Grammar

2.3.1 Order of Evaluation

A J expression is evaluated from right to left, as is natural in mathematics. Thus there is an implicit ‘of’ (or ‘the result of’) between successive verbs in a long expression. For example, $n \log \log n$, and the corresponding J expression $n * ^{-} ^{-} . ^{-} n$, can be read as ‘$n$ times the result of log of the result of log of $n$’. Similarly, $1 x 1 -+/ % i . n$ is evaluated in the order shown below:

\[
\begin{align*}
&i . 5 \quad \text{NB. first 5 non-negative integers} \\
&0 1 2 3 4 \quad \text{NB. factorials of previous result} \\
&! i . 5 \quad \text{NB. reciprocals of previous result} \\
&1 1 2 6 24 \quad \text{NB. sum of previous result} \\
&% ! i . 5 \quad \text{NB. e - previous result} \\
&1 x 1 - (+/ (% (! (i . 5)))) \quad \text{NB. order of evaluation made explicit} \\
&0.009948495 \\
&0.009948495
\end{align*}
\]

i.e. $1 x 1 -+/ % i . n$ reads as ‘$e$ minus the the sum of the reciprocals of the factorials of the first $n$ non-negative integers’.

The exceptions to this rule of right-to-left evaluation are:

1. Adverbs and conjunctions are applied before verbs (hence $+/$ acts as a single entity in the above expression),

2. The order of evaluation can be changed by parentheses, as illustrated in the following dialogue:

\[
\begin{align*}
&-/ 1 2 3 4 \quad \text{NB. equivalent to 1-2-3-4} \\
&-/ 2 1 \quad \text{NB. 1 - (2 - (3 - 4))} \\
&-/ 2 \quad (1 - 2) - 3 \quad 4 \\
&-/ 8 \quad (1 - 2) - (3 - 4) \\
&-/ 0 \quad (1 - 2) + (3 - 4) \\
&-/ 2 \quad y = .1 .7 0 .3 5 .4 8 .6 2 .3 \\
&+/ % # y \quad \text{NB. sum the reciprocal of the number of items in } y \\
&0.2 \quad (+/ % #) y \quad \text{NB. create the verb 'sum divided by number of', and apply to } y \\
&3.66
\end{align*}
\]
Note in particular that ] (left) is a verb, so if you use it as a statement separator, as on page 22, then you need to remember that the expression to the right of the ] will be evaluated before the statement left of the ], as in the following snippet:

```
a.='LEFTMOST evaluated LAST' ] b=. a ] a.='RIGHTMOST evaluated FIRST'
a
LEFTMOST evaluated LAST
b
RIGHTMOST evaluated FIRST
```

### 2.3.2 Trains and Hooks and Forks

A *train* is a list of J words that doesn’t evaluate to a noun. In particular, a list of two verbs is a called a *hook* and a list of three verbs is called a *fork*.

For compatibility with the on-line J documentation, nouns in this section will usually be represented simply x and y, and verbs by f, g and h (or occasionally V₁, V₂, V₃ etc.)

#### 2.3.2.1 Monadic Hooks

The expression (g h)y means y g(h y), i.e. the monadic hook (g h) applies h monadically to y, then dyadic g with left argument y and right argument h(y).

This interpretation of g h makes good sense when the hook is read in English. For example, the hook − mean can be read as ‘subtract the mean’ (assuming that you’ve already defined the verb mean), and (− mean)y reads as ‘apply “subtract the mean” to y’, i.e. calculate y − mean(y).

Further examples of monadic hooks are given below.

```
ltr=. < 4:  NB. 'less than 4' (1 if true, 0 if false)
ltr 3 1 4 1 5 9 _2 _6 _5 _3 _5
1 1 0 1 0 0 1 1 1 1
isint=. = <.  NB. 1 if y. equals floor of y. (i.e. if y. is an integer)
isint _123 _ip1 _1.5 0 1 _x1 _ip1 _1e1
1 0 0 1 1 0 0 1
addcol1=. ,. 1:  NB. append a column of 1's
addcol1 1.7 0.3 5.4 8.6 2.3
1.7 1
0.3 1
5.4 1
8.6 1
2.3 1
prz=. + %  NB. 'plus reciprocal'
prz 1 2 3 4 5 100 0.1 0.01 100
2 2.5 5.33333 4.25 5.2 100.01 10.1 100.01 100.01
issym=. 1:  NB. 'match transpose' i.e. is y. symmetric?
issym _1: b=: i. 3 3 [ c=: (d=. /* i.4),0 [ e=: (|/- i.5) { 4 1 0 0 0
xrs , (i: 'issym &.',)"0 'abcde'
```

<table>
<thead>
<tr>
<th>a</th>
<th>issym a</th>
<th>b</th>
<th>issym b</th>
<th>c</th>
<th>issym c</th>
<th>d</th>
<th>issym d</th>
<th>e</th>
<th>issym e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10 40</td>
<td>1</td>
<td>0 1 4</td>
<td>0</td>
<td>1 0 0</td>
<td>1</td>
<td>0 0 0</td>
<td>1</td>
<td>4 1 0 0</td>
</tr>
<tr>
<td>10</td>
<td>32 74</td>
<td>9</td>
<td>16 25</td>
<td>0</td>
<td>0 1 0</td>
<td>0</td>
<td>0 1 0</td>
<td>0</td>
<td>1 4 1 0</td>
</tr>
<tr>
<td>40</td>
<td>74 128</td>
<td>36</td>
<td>49 64</td>
<td>0</td>
<td>0 0 1</td>
<td>0</td>
<td>0 0 1</td>
<td>0</td>
<td>0 1 4 1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Hook
\[
\begin{array}{cc|cc}
\text{monadic} & \text{dyadic} & \text{monadic} & \text{dyadic} \\
(g \ h) \ y & x(g \ h) \ y & (f \ g \ h) \ y & x(f \ g \ h) \ y \\

g & g & g & g \\
/ \ \ \ \ / \ \ \ / \ \ \\
\text{Depiction:} & y & h & x & h & f & h & f & h & h & h \\
/ & | & / & | & / & / & / & / \\
y & y & y & y & x & y & x & y & y & x & y \\
i.e.: & y \ g(h \ y) & x \ g(h \ y) & (f \ y) \ g(h \ y) & (x \ f \ y) \ g(x \ h \ y) & g(h \ y) \ g(x \ h \ y) \\
\end{array}
\]

Figure 2.1: Hooks and Forks

```
new, i.e., \textit{NB. 'index of nub'}, i.e. first occurrences of new items}
new 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7
0 1 2 4 5 6 7 11 13 32
```

Comments:

1. 4: and 1: are examples of \textit{constant functions}. J defines the 19 verbs \texttt{:_9: _8: \ldots 9:}, together with the verb \texttt{_:} (\textit{infinity}). These each return the specified number, irrespective of the shape and value of the argument(s).

2. Dyadic \texttt{=:} (\textit{match}) returns 1 if and only if its left and right arguments are identical (Mnemonic: -: resembles \equiv. Well, it does a bit.)

3. Monadic \texttt{-:} (\textit{nub}) returns the distinct items of \texttt{y.} in the order in which they occur.

4. The name ‘hook’ comes (rather fancifully) from the depiction of its syntax in Figure 2.1.
2.3.2.2 Dyadic Hooks

The expression \( x(y \text{ h})y \) means \( x g(h, y) \), see Figure 2.1. For example, \( x(-1)y \) means ‘\( x \) minus the absolute value of \( y \)’, and the hook \(- | \) can be read simply as ‘minus the absolute value of’.

The following examples of dyadic hooks use J’s various verbs (/: and \:) for grading and sorting:

```
largest=: . \: - NB. ‘take (from) downwards sort’, i.e. largest \( x \) values in \( y \).
2 largest 1.7 0.3 5.4 8.6 2.3
8.6 5.4
flagrank=: /:/: - NB. ‘sort by grade up’, i.e. flag smallest \( y \) by 0(\( x \), etc.
(cut ‘smallest second third fourth largest’) flagrank 1.7 0.3 5.4 8.6 2.3

<table>
<thead>
<tr>
<th>second</th>
<th>smallest</th>
<th>fourth</th>
<th>largest</th>
<th>third</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 0 1 4 5 2 9 7 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sortabs=: . /: | NB. sort \( x \) in order of absolute values of \( y \).
sortabs= 3.1 4.1 5.9 _2.6 _5.3 _5.8 9.7 _9.3 2.3 8.4
2.3 _2.6 3.1 4.1 _5.3 _5.8 8.9 _9.3 9.7
(1.10) sortabs 3.1 4.1 _5.9 _2.6 _5.3 _5.8 9.7 _9.3 2.3 8.4
8 3 0 1 4 5 2 9 7 6

sortrev=: \: /"1 NB. sort \( x \) down by reversed items of \( y \). (useful for dates)
date=: 7 3 $ 4 6 1950 16 8 1944 2 3 1942 2 4 1939 9 9 1941 28 1 1945 9 2 1940
name=: 7 7*/'Dagmar Kevin Lewis Marvin Utis Robert William'
xrs 'dates'; 'names'; 'brep 'browse''; 'sortrev dates'; 'names sortrev dates'
```

<table>
<thead>
<tr>
<th>dates</th>
<th>names</th>
<th>brep 'sortrev'</th>
<th>sortrev dates</th>
<th>names sortrev dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 6 1950</td>
<td>Dagmar</td>
<td>[ 1 1 ]</td>
<td>4 6 1950</td>
<td>Dagmar</td>
</tr>
<tr>
<td>16 8 1944</td>
<td>Kevin</td>
<td></td>
<td>28 1 1945</td>
<td>Robert</td>
</tr>
<tr>
<td>2 3 1942</td>
<td>Lewis</td>
<td></td>
<td>16 8 1944</td>
<td>Kevin</td>
</tr>
<tr>
<td>2 4 1939</td>
<td>Marvin</td>
<td></td>
<td>2 3 1942</td>
<td>Lewis</td>
</tr>
<tr>
<td>9 9 1941</td>
<td>Otis</td>
<td>9 9 1941</td>
<td>9 9 1941</td>
<td>Otis</td>
</tr>
<tr>
<td>28 1 1945</td>
<td>Robert</td>
<td>9 2 1940</td>
<td>9 2 1940</td>
<td>William</td>
</tr>
<tr>
<td>9 2 1940</td>
<td>William</td>
<td>2 4 1939</td>
<td>2 4 1939</td>
<td>Marvin</td>
</tr>
<tr>
<td>7 3</td>
<td>7 7 2</td>
<td>7 3</td>
<td>7 7</td>
<td></td>
</tr>
</tbody>
</table>

Comments:

1. Monadic /: (grade up) gives the permutation of the indices of the items of \( y \). that would put them in increasing order. Similarly monadic \:\: (grade down) gives the permutation for decreasing order.

2. Dyadic /: (sort up) and \:\: (sort down) sort \( x \) according to the permutation /: \( y \). or \:\: \( y \). respectively. Thus, /: \( y \). sorts \( y \). from lowest to highest, and \:\: \( y \). from highest down to lowest.

3. The utility brep is automatically defined on starting J, and gives the boxed representation of the J object named by \( y \). For example, a hook \( gh \) is represented by two adjacent boxes containing the boxed representations of \( g \) and \( h \).
2.3.2.3 Forks

The expression \((f \; g \; h)\; y\) is a *monic fork*. It means \((f \; y)\; g(\; h \; y)\), i.e. apply \(f\) and \(h\) monadically to \(y\), and then dyadic \(g\) between the results. As with hooks, the interpretation makes good sense in English. For example, \((/ \; \% \; #)\; y\) could be read as ‘form the verb “sum-over divided by number-of”, and apply it to \(y\)’:

```
abslt4=. | < 4: NB. 'magnitude (i.e. absolute value) less than 4'
abslt4 3 1 4 1 5 9 2 6 _5 _3 _5
1 1 0 1 0 0 1 0 0 1 0
isint=. < = >. NB. 'floor' = 'ceiling', i.e. is \(y\) integral?
isint %: i. 20
1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
mean= / % # NB. 'sum over' divided by 'number of items', i.e. arithmetic mean
mean 1.7 0.3 5.4 8.6 2.3
3.66
gean= / % #: NB. 'no. of items'th root of 'product over', i.e. geometric mean
gean 1.7 0.3 5.4 8.6 2.3
2.22453
minmax= < / , > / NB. 'least over', 'largest over', i.e. min, max
minmax 1.7 0.3 5.4 8.6 2.3
0.3 8.6
range= > / - < / NB. 'largest over' minus 'least over', i.e. \((\text{max} - \text{min})\) = range
range 1.7 0.3 5.4 8.6 2.3
8.3
eqnext= }, = }: NB. 1 if current value = next value in list
eqnext 1 3 1 1 2 2 1 0
0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 1 0 0 1 0 1 1 0
```

Dyadic forks have a similar syntax to monadic forks: \(x\; (f \; g \; h)\; y\) means \((x \; f \; y)\; g(\; x \; h \; y)\), i.e. apply dyadic \(g\) to the results of applying \(f\) and \(h\) separately:

```
std=. + * - NB. sum times difference
1.0 std 0.6 NB. \((1.0 + 0.6) \times (1.0 - 0.6)\)
0.64
divides=. | = 0: NB. 1 iff \(x\) divides \(y\).
divides 12 divides 12 48 49 _132 1728 11999
1 1 0 1 1 0
symsdiff= -. , - NB. symmetric difference (items in \(x\), but not \(y\), or vice versa)
symsdiff 3 1 4 1 5 9 2 6 5 3 5
3 6 3 7 8 8 0
'odd mark on the doormat' symsdiff 'no drink or corks'
mathematics
sdsieve=. ; e. symsdiff NB. flag items of (\(x\), \(y\)) that form symsdiff
sdsieve 3 1 4 1 5 9 2 6 5 3 5
1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0
_1 ( , , , ) _3 i. 3 4 NB. display different effects of , and ,
-1 _1 _1 _1 _1 1 0 1 2 3
0 1 2 3 1 4 5 6 7
4 5 6 7 _1 1 8 9 10 11
8 9 10 11
```

Comments:

1. Dyadic -. (less) copies its right argument \(x\), but omitting items that are in \(y\).
2. -. (nub sieve) returns 1 for the first occurrence of each distinct item of \(y\), and 0 if an item has already appeared.
3. e. (member) returns 1 for each item of x. that also occurs as an item of y. and 0 for items of x. not in y.

4. The name ‘fork’ comes from the depiction of its syntax in Figure 2.1.

### 2.3.2.4 Trains

A train of n verbs is evaluated from the rightmost end as a succession of forks, culminating in a fork if n is odd or a hook if n is even. Thus (; ~ ; ~ ; =) parses as (; (~ ; (~ ; =))), i.e. a hook (V1V2) where V1 is ; and V2 is a fork (V3V4V5) in which V5 in turn is a fork (~ ; =):

\[
(\text{; ~ ; ~ ; =}) \text{ 'Canaan Banana'}
\]

<table>
<thead>
<tr>
<th>Canaan Banana</th>
<th>Can B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 0 1 1 0 0 0 0 0 1</td>
<td>1 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0 0 0 1 0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 1 0 0 0 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{dev} = - +/ \% # \quad \text{NB. deviations from mean}\]
\[\text{dev} = - 1.7 0.3 5.4 8.6 2.3 \]
\[-1.96 -3.36 1.74 4.94 -1.36 \]
\[\text{score} = +/ - > + <. / \quad \text{NB. sum excluding largest and smallest values}\]
\[\text{score} = 5.9 5.9 6.0 5.8 5.6 5.9 6.0 6.0 5.9 \]
\[41.4 \]
\[\text{isposint} = (> 0:)* . < . = >. \quad \text{NB. is (each atom of y.) a positive integer?}\]
\[\text{isposint} = 0 1 2 _1 1.0 1.2 1e2 1p2 1x2 \]
\[0 1 1 0 1 0 1 0 0 \]
\[3 1 4 1 5 9 2 6 5 3 5 (, , , e . , . , , .) 2 7 1 8 2 8 4 5 9 0 4 5 2 \]
\[3 1 4 1 5 9 2 6 5 3 5 2 7 1 8 2 8 4 5 9 0 4 5 2 \]
\[4 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 \]

The monadic verb = (self-classify), used in the first train above, returns a matrix \( M \) whose \((i,j)\) entry \( m_{ij} \) is 1 if the \( i \)th element of the nub of its argument \( y \). equals the \( j \)th element of \( y \)., otherwise \( m_{ij} = 0 \).

### 2.3.2.5 Cap ([i:])

A fork can be capped: \([i : g h] y \) means \( g(h(y)) \) and \( x \) \(([: g h] y \) means \( g(x h y) \), as depicted in Figure 2.1. This technique often allows J functions to be expressed as unbroken verb trains. The resulting J code avoids parentheses and conjunctions like \( @ \) (Section 2.3.6), and may be more readable:

\[\text{logit} = ^^\circ @ (\% -) \quad \text{NB. logit(p) = log(p/(1-p))}\]
\[\text{logit} = 0.1 0.5 0.9 0.99 \]
\[-2.19722 0 2.19722 4.59512 \]
\[\text{logit2} = [: ^^\circ ] \% -. \quad \text{NB. using cap and forks to create an unbroken train}\]
\[\text{logit2} = 0.1 0.5 0.9 0.99 \]
\[-2.19722 0 2.19722 4.59512 \]

However, don’t get carried away! For example, the following verb \( \text{fib} \), giving the \( y \).th Fibonacci numbers, is utterly incomprehensible:

\[\text{fib} = 20 \]
\[0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 \]
\[\text{fib} = 30 \]
\[832040 \]

36
Even if you know that the $n$th Fibonacci number $F_n$ is given by

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{\sqrt{5}},$$

(2.1)

and that the second term in the numerator of Equation 2.1 is close to zero, `fib` remains completely obscure and arguably obscene. It’s much better style to split a large J expression into smaller ones, with comments. I also usually avoid the passive tense (e.g. `%` and `~`), and might for teaching purposes define `fib` as follows:

```j
fib =. ] f3 [ : %: 5: NB. fib arg = f3 with (x.=arg) and (y.=sqrt(5))
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

2.3.3 Some Adverbs

2.3.3.1 Insert (Monadic u./)

The expression `fy.` inserts the dyadic verb `f` between the items of `y`, and evaluates the result. Many examples of the resulting ‘derived verbs’ `f/` have been met already:

```j
x=. 12 6 _6 42 _30
+/x  NB. sum
24
*/x  NB. product
544320
-./x  NB. alternating sum (12-6)+(_6-42)+_30
_72
<./ , >./x  NB. min, max
_30 42
+./ , */x  NB. HCF, LCM
6 420
```

Several verbs derived by insertion are useful when manipulating arrays. For example, monadic `,./` runs together the first two axes of `y`, forming an array of rank `(*/2.(y.), 2).y.`

```j
xrs ',/i.2 3 4'; ',./i.2 3 4'; '/i.2 3 4'
```

<table>
<thead>
<tr>
<th>,/i.2 3 4</th>
<th>,./i.2 3 4</th>
<th>;/i.2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3 12</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4 5 6 7 16</td>
<td>4 5 6 7 12</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>8 9 10 11</td>
<td>8 9 10 11</td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>20 21 22 23</td>
<td></td>
</tr>
<tr>
<td>16 17 18 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 21 22 23</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>6 4</td>
<td>3 8</td>
<td>2</td>
</tr>
</tbody>
</table>
Many mathematical objects can be defined naturally using insertion. For example, a continued fraction is an expression of the form

\[
a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}.
\]

In J this is just a_1 + a_2 % a_3 + a_4 + ..., or, rather prettily, (+%)/ a_1,a_2,a_3,a_4,.....:

\[
(+%)/ 1 1 2 1 1 4 1 1 6 1 1 8 1 1 10 \quad \text{NB. CF approximation to (e-1)}
\]
\[
1.71828
\]
\[
(+%)/ 1 2 2 2 2 2 2 2 2 2 2 2 2 2 \quad \text{NB. CF approximation to sqrt(2)}
\]
\[
1.41421
\]
\[
fibx=: \{@ (2&x:) @ ((+%)/ @ x:) @ #@1"0 \quad \text{NB. y.th Fibonacci number (extended)}
\]
\[
fibx 1 2 3 4 5 30 50 100
\]
\[
1 1 2 3 5 832040 12586269025 354224848179261915075
\]

### 2.3.3.2 Prefix (Monadic u.\_) and Suffix (Monadic u.\.)

The result of f \ y. has n = #y, items obtained by applying f to the first item, the first two items,...., all n items of y. (i.e. to all n prefixes of y.). This construction is particularly useful when f is itself derived by insertion, i.e. when f is of the form verb/ as illustrated below:

| record=. >./ | NB. largest so far |
| record 2 7 1 8 2 8 4 5 9 0 4 5 2 |
| 2 7 7 8 8 8 8 9 9 9 9 9 | NB. cumulative sum |
| cumsum. = +./ | NB. triangular numbers |
| 0 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 | NB. product so far |
| cumprod. = *./ | NB. factorials |
| cumprod 1x + i. 12 | |
| 1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600 | |
| (+%)\ 1 , 10^#2 | NB. successive continued fraction approxs. to %: 2 |
| 1 1.5 1.4 1.41667 1.41379 1.41429 1.41422 1.41422 1.41421 1.41421 1.41421 | |
| (+%)\ 1x , 10^#2 | NB. successive optimal rational approxs. to %: 2 |
| 1 3r2 7r5 17r12 41r29 99r70 239r169 577r408 1393r985 3363r2378 8119r5741 | |
| <' 'spoilage' | NB. box substrings |
| s | sp | spo | spoil | spoil a | spoilage | spoilage |

([: < -/)' 'spoilage' NB. arrange first 1, 2, 3,.... chars alphabetically

| s | p | s | p | o | s | i | o | s | a | g | i | o | s | a | g | i | o | s | a | g | i | o | s | a | g | i | o | s |

Many useful J phrases have the form f/\ y. where y. is a logical list:

| <\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1 | NB. i only for first 1 in y. |
| 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | |
| 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | |
| 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | |
| 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | |
| 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | |
| 1 1 1 0 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 | |
| 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | |
| */ 1 1 1 0 0 1 0 1 1 0 1 0 0 0 0 | NB. as for <\ |
Monadic \( \cdot \) (suffix) similarly applies its verb right argument to each of the suffixes of \( y \).

\[
\texttt{<\cdot 'zyxoma'}
\]

```
xymma yxomma yomma omma mma ma a
```

\textbf{OverRest} = \texttt{/.} \quad \textbf{NB. example of a user-defined adverb}
\textbf{<} . \textbf{OverRest} 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9
1 1 1 1 2 2 2 3 3 5 7 7 7 9
+ \textbf{OverRest} 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9
7 7 7 4 7 3 6 9 6 8 6 3 5 4 5 2 4 6 4 1 3 8 3 3 2 5 1 6 9
\textbf{rtb} = . ([: >/. (.-&))] \#) \quad \textbf{NB. remove trailing blanks}
\textbf{xrs} 'rtb' 'OK''; 'rtb' 'OK''; 'rtb'' only TRAILING blanks removed ''
\textbf{rtb} 'OK'\textbf{rtb} 'OK'\textbf{rtb} 'only TRAILING blanks removed'
\textbf{OK} \textbf{OK} \textbf{only TRAILING blanks removed}
2 2 31
```

2.3.3 Infix (Dyadic u./) and Outfix (Dyadic u.

Dyadic \( u./ \) acts on \textit{infixes} (sets of successive items of \( y \)). If \( x \) is positive, then \( x \cdot f \) \( y \) applies \( f \) to each infix of length \( x \) in \( y \). If \( x \) is negative, then \( x \cdot f \) \( y \) applies \( f \) to each non-overlapping infix of length \(-x\) in \( y \). (The last infix may have length less than \(-x\)).
y=. 3 1 4 1 5 9 2 6 5 3 5
3 <\ y  NB. box successive overlapping groups of 3
   3 1 4 1 4 1 5 | 1 5 9 5 9 2 9 2 6 2 6 5 | 6 5 3 5 3 5
5 <\ y
   3 1 4 1 5 1 4 1 5 9 4 1 5 9 2 1 5 9 2 6 5 9 2 6 5 9 2 6 5 3 2 6 5 3 5
5 (+/ % #)\ y  NB. running averages of length 5
2.8 4 4.2 4.6 5.4 5.4 5.2
3 (+/- < .-/ + .->)\ y  NB. running medians of length 3 (cute eh?)
3 1 4 5 5 6 5 5 5
diff=. 2: -/\ ]  NB. successive difference
diff y
2 _3 3 _4 _4 7 _4 1 2 _2
3 - +/-\ diff y
1 4 1 5 9 2 6 5 3 5
   _3 <\ y  NB. non-overlapping infixes (-ve right argument)
   3 1 4 1 5 9 2 6 5 3 5

Omitting an infix from y. leaves an outfix, and dyadic \. (outfix) acts on outfixes analogously to the way that dyadic \ acts on infixes:

   y=. 3 1 4 1 5 9 2 6 5 3 5
6 <\. y
   2 6 5 3 5 3 6 5 3 5 | 3 1 5 3 5 | 3 1 4 3 5 | 3 1 4 1 5 | 3 1 4 1 5
   _3 <\. y  NB. compare with _3 <\y
   1 5 9 2 6 5 3 5 | 3 1 4 2 6 5 3 5 | 3 1 4 1 5 9 3 5 | 3 1 4 1 5 9 2 6 5

1 (+/ % #)\. y  NB. means of each (# y)-1 subset of y
4.1 4.3 4.3 3.9 3.5 4.2 3.8 3.9 4.1 3.9

2.3.4 Actions on Arrays

J operates on complete arrays simultaneously, simplifying many computer programming tasks enormously. For example, verbs like %: (square root) and ^-. (natural log) work with arguments of any rank and shape:

 xrs 'ProbTab'; %: 'ProbTab'; ^-. 'ProbTab'  NB. ProbTab was created previously

<table>
<thead>
<tr>
<th>ProbTab</th>
<th>%: ProbTab</th>
<th>^-. ProbTab</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.181 0.111 0.078</td>
<td>0.4254409 0.3331666 0.2792849</td>
<td>1.70926 2.19823 2.55105</td>
</tr>
<tr>
<td>0.054 0.054 0.072</td>
<td>0.232379 0.232379 0.2683262</td>
<td>2.91877 2.91877 2.63109</td>
</tr>
<tr>
<td>0.024 0.024 0.032</td>
<td>0.1549193 0.1549193 0.1788854</td>
<td>3.7297 3.7297 3.44202</td>
</tr>
<tr>
<td>0.041 0.111 0.218</td>
<td>0.2024846 0.3331666 0.4669047</td>
<td>3.19418 2.19823 1.52326</td>
</tr>
<tr>
<td>2 2 3</td>
<td>2 2 3</td>
<td>2 2 3</td>
</tr>
</tbody>
</table>
Similarly, the left argument of \( \{ \text{(from) and the right argument of dyadic i. (index of) can have any shape:} \)

\[
\begin{array}{|c|c|c|}
\hline
k & ] a =. k \{ 'PATERNOSTER' i. a' & 'PATERNOSTER' i. a'
\hline
7 1 2 6 4 & SATOR & 83 65 84 79 82 7 1 2 6 4
1 4 3 0 6 & AREPO & 65 82 69 80 79 1 4 3 0 6
2 3 5 3 2 & TENET & 84 69 78 69 84 2 3 5 3 2
6 0 3 4 1 & OPERA & 79 80 69 82 65 6 0 3 4 1
4 6 2 1 7 & ROTAS & 82 79 84 65 83 4 6 2 1 7
\hline
\end{array}
\]

In general, \( x \{ y \) extracts the \( x \)th items from \( y \), so the result has shape \( (\{ x \}) \), \( } \). $ y.$

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{kings} & 6 7 & 'Edward Henry John Richard Stephen William'
\hline
\text{k} & 5 5 1 4 1 3 2 & 1 0 0 0 3 1 1
\hline
\text{colours} & (cut 'black blue green cyan red magenta yellow white'), \;. /2 2 2\$:i. 8
\hline
\text{c} & 2 1 3 \$ 4 7 1 7 0 7
\hline
\text{xrs 'kings'} & 'k'; 'k{kings'}; 'colours'; 'c'; 'c{colours}'
\hline
\end{array}
\]

2.3.4.1 Function rank

Every J verb has an associated rank: a monadic verb of rank \( k \) acts on the \( k \)-cells of its argument \( y \). For example, monadic i. (integers) has intrinsic rank 1, and interprets each 1-cell (vector) of \( y \) as the shape of a subarray to be filled with consecutive integers. The resulting subarrays, padded if necessary with zeros, constitute the cells of the result.

The rank conjunction " is used to change the rank of a verb: \( V^" k \) applies the verb \( V \) to the \( k \)-cells of its argument \( y \). Thus i. "0 creates integer lists whose lengths are given by the corresponding atoms of \( y \), and inserts them (padded if necessary) into a frame of shape \( $ y \).
xrs 'i.2 3' ; 'i.3 5' ; 'i.2 2$2 3 3 5' ; 'i."0 (2 2$2 3 3 5)'

<table>
<thead>
<tr>
<th>i.2 3</th>
<th>i.3 5</th>
<th>i.2 2$2 3 3 5</th>
<th>i.&quot;0 (2 2$2 3 3 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>0 1 2 3 4</td>
<td>0 1 2 0 0 0 0 0 0</td>
<td>0 1 2 0 0 0 1 2 0 0</td>
</tr>
<tr>
<td>3 4 5</td>
<td>5 6 7 8 9</td>
<td>3 4 5 0 0 0</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>10 11 12 13 14</td>
<td>5 6 7 8 9 10 11 12 13 14</td>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
</tr>
</tbody>
</table>

2 3 3 5  | 2 3 5  | 2 2 5  |

The effect of " on monadic verbs is easy to see using < (box). For example, <"1 has the same rank as i., and the following explicitly shows the arguments to i. in the above example:

```j
iargs = <" NB. same rank as monadic i.
xrs 'iargs 2 3'; 'iargs 3 5'; 'iargs 2 2$2 3 3 5'; 'iargs"0 (2 2$2 3 3 5)'
```

<table>
<thead>
<tr>
<th>iargs 2 3</th>
<th>iargs 3 5</th>
<th>iargs 2 2$2 3 3 5</th>
<th>iargs&quot;0 (2 2$2 3 3 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3</td>
<td>3 5</td>
<td>2 3 3 5</td>
<td>2 3 5</td>
</tr>
</tbody>
</table>

Some other examples of the intrinsic ranks of monadic verbs are given in the next example:

1. Simple mathematical functions such as monadic %: (square root) and ^. (natural log) have rank 0, so act on each atom of y. individually (see page 40).

2. Monadic #. (base 2) has rank 1, and interprets each 1-cell (vector) of y. as the base 2 representation of a number. Note that the atoms of y. are not restricted to 0 and 1, for example #. 10 3 is 23.

3. Monadic %.. (matrix inverse) has rank 2; each 2-cell of y. is treated as a matrix, and is inverted if possible (see Section 2.3.5.3).

   However, if y. is a nonzero vector then %. inverts it with respect to the unit circle/sphere/hypersphere, i.e. returns the vector (y. %+/ *: y.), which lies parallel to y. but which has length the reciprocal of y.s length.

4. Monadic $ (shape of) has unbounded rank (,), so returns the shape of y. as a whole. Similarly $"k returns the shapes of the k-cells of y., in a frame of shape (–k). $ y.).
A dyadic verb \( V \) has both a left rank \( k_1 \) and a right rank \( k_2 \), specified by \( V^{k_1 k_2} \). If the required left rank \( k_1 \) and right rank \( k_2 \) are identical, then \( V^{k_1 k_2} \) can be abbreviated to \( V^{k_1} \). For example, giving a dyadic verb \( V \) left rank 0 and right rank 1 will apply \( V \) to corresponding atoms of \( x \). and 1-cells of \( y \):

\[
xrs 'act3scene10' \; ; '1 1 1 2 1 1 {."0 1 act3scene10'
\]

You can check the ranks of a verb \( f \) by \( f \). 0, an example of the adverb \( b \). (basic characteristics). The result is a list of 3 elements: (monadic rank), (left rank), (right rank).

For example, \( %b.0 \) produces \(-1\)-, showing that dyadic \( $ \) (shape) has left rank 1 and right rank \(-\). More generally, \( $^{k_1 k_2} \) interprets each \( k_1 \)-cell of \( x \). as a shape to be filled by the items of the corresponding \( k_2 \)-cells of \( y \).

\[
x=.2 2$2 3 3 5
y-. 5 6
xrs ' $ b. 0' \; ; 'x'; 'y'; 'x$y'; 'x$'1 y'; 'x$'0 y'; 'x$'0 0 y'; 'x$'1 0 y'
\]
All three ranks of a verb \( V \) may be specified, using \( V^"k_1 k_2" \). Otherwise the omitted ranks are derived from the specified ones, as described in The J Introduction and Dictionary[15].

The action of verbs derived by insertion can be modified in numerous ways using the rank conjunction:

\[
c = (97 + i. 2 3 4) \{ a. \\
xrs \ 'c'; \ ','c'; \ "0/c'; \ "1/c'; \ "2/c'; \ "0 1/c'; \ "0 1/2 c'; \ :/c'; \ :"1/c'
\]

<table>
<thead>
<tr>
<th>c</th>
<th>/c</th>
<th>&quot;0/c</th>
<th>&quot;1/c</th>
<th>/&quot;2 c</th>
<th>&quot;0 1/c</th>
<th>&quot;0 1/2 c</th>
<th>:/c</th>
<th>:&quot;1/c</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcd efgh ijkl</td>
<td>abcd efgh ijkl</td>
<td>abcd efgh ijkl</td>
<td>abcd efgh ijkl</td>
<td>abcd efgh ijkl</td>
<td>abcd efgh ijkl</td>
<td>abcd efgh ijkl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mnop qrst uvwx</td>
<td>mnpqrsuvwx</td>
<td>mnpqrsuvwx</td>
<td>mnpqrsuvwx</td>
<td>mnpqrsuvwx</td>
<td>mnpqrsuvwx</td>
<td>mnpqrsuvwx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lu jv kw lx</td>
<td>iuvw juvx kuwx luvw</td>
<td>( iuvw juvx kuwx luvw )</td>
<td>( iuvw juvx kuwx luvw )</td>
<td>( iuvw juvx kuwx luvw )</td>
<td>( iuvw juvx kuwx luvw )</td>
<td>( iuvw juvx kuwx luvw )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This simplifies the statistical analysis of multi-way tables. For example, +/ has unbounded rank, so +/y. sums over the first axis of y.. Therefore +/"2 y. sums over the first axis of each 2-cell of y. and puts the results in a frame of shape \( \{ _2 \} \). $ y. \).

\[
xrs \ 'ProbTab'; \ '+/ b. 0'; \ '+/ProbTab'; \ '+/"1 ProbTab'; \ '+/"2 ProbTab'
\]

<table>
<thead>
<tr>
<th>ProbTab</th>
<th>+/ b. 0</th>
<th>+/ProbTab</th>
<th>+/&quot;1 ProbTab</th>
<th>+/&quot;2 ProbTab</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.181 0.111 0.078 0.054 0.054 0.072 0.024 0.024 0.032 0.041 0.111 0.218</td>
<td>_ _</td>
<td>0.205 0.135 0.11</td>
<td>0.37 0.18</td>
<td>0.235 0.165 0.15</td>
</tr>
<tr>
<td>0.095 0.165 0.29 0.08 0.37 0.065 0.135 0.25</td>
<td></td>
<td>0.37 0.18</td>
<td>0.235 0.165 0.15</td>
<td></td>
</tr>
<tr>
<td>2 2 3 3 2 3</td>
<td>2 2 2 3</td>
<td>2 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

44
Other phrases like (+/ % #)/2 act similarly:

\[
\text{mean} = +/ \% # \quad \text{A} = . \ i. 2 3 4 \\
xrs 'A'; 'mean b. 0'; 'mean A'; 'mean"1 A'; 'mean"2 A'
\]

<table>
<thead>
<tr>
<th>A</th>
<th>mean b. 0</th>
<th>mean A</th>
<th>mean&quot;1 A</th>
<th>mean&quot;2 A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>_ _ _</td>
<td>6 7 8 9</td>
<td>1.5 5.5 9.5</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>10 11 12 13</td>
<td>13.5 17.5 21.5</td>
<td>16 17 18 19</td>
<td></td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>14 15 16 17</td>
<td>16 17 18 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>16 17 18 19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 21 22 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 4</td>
<td>3</td>
<td>3 4</td>
<td>2 3</td>
<td>2 4</td>
</tr>
</tbody>
</table>

The arguments of a dyadic verb \( V \) must agree: if \( x \) is a frame \( F_1 \) of \( k_1 \)-cells, and \( y \) is a frame \( F_2 \) of \( k_2 \)-cells, where \( V \) has rank \((k_1 k_2)\), then the shapes of \( F_1 \) and \( F_2 \) must either be the same, or else one must be a prefix of the other.

Similarly the shapes of the \( k_1 \)-cells of \( x \) and the \( k_2 \)-cells of \( y \) must also agree—if necessary the shapes are implicitly extended by prepending 1s.

Formal definitions of verb rank and rank agreement are given in the J Introduction and Dictionary[15], Section II.B. Verbs.

\[
i234 = . \ i. 2 3 4 \\
xrs 'i234'; '2 i234'; '(i. 2 3)*i234'; '(i. 4)*"1 i234'; '(i. 2 4)*"1 i234'
\]

\[
i234 \quad 2 \text{ i234} \quad (i. 2 \text{ 3})*\text{i234} \quad (i. 4)*"1 \text{i234} \quad (i. 2 \text{ 4})*"1 \text{i234}
\]

<table>
<thead>
<tr>
<th>i234</th>
<th>2 i234</th>
<th>(i. 2 3)*i234</th>
<th>(i. 4)*&quot;1 i234</th>
<th>(i. 2 4)*&quot;1 i234</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 2 4 6</td>
<td>0 0 0 0</td>
<td>0 1 4 9</td>
<td>0 1 4 9</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>8 10 12 14</td>
<td>4 5 6 7</td>
<td>0 5 12 21</td>
<td>0 5 12 21</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>16 18 20 22</td>
<td>16 18 20 22</td>
<td>0 9 20 33</td>
<td>0 9 20 33</td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>12 13 14 15</td>
<td>36 39 42 45</td>
<td>0 13 28 45</td>
<td>48 65 84 105</td>
</tr>
<tr>
<td>16 17 18 19</td>
<td>16 17 18 19</td>
<td>64 68 72 76</td>
<td>0 17 36 57</td>
<td>64 85 108 133</td>
</tr>
<tr>
<td>20 21 22 23</td>
<td>20 21 22 23</td>
<td>100 105 110 115</td>
<td>0 21 44 69</td>
<td>80 105 132 161</td>
</tr>
<tr>
<td>2 3 4</td>
<td>2 3 4</td>
<td>2 3 4</td>
<td>2 3 4</td>
<td></td>
</tr>
</tbody>
</table>

(i. 4) * i234 \quad \text{NB. 4 is not a prefix of 2 3 4}

length error \( (i. 4) * i234 \)

length error \( (i. 2) *"1 i234 \quad \text{NB. shapes 2 and 4 of the 1-cells don’t agree}

length error \( (i.2) *"1 i234 \)
Finally, the rank conjunction can also take a noun left argument: $N^\# k$ is a verb of rank $k$, returning $N$ for each $k$-cell of $y$. (in the dyadic case, the arguments to $N^\# k$ must agree as described above). In particular, $N^\#$ is the constant function with value $N$:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 2 & 1 & 3 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 2 & 1 & 3 & 4 \\
\end{array}
\]

[2.3.5 Matrix Algebra]

2.3.5.1 Determinants etc.,

The determinant of a square matrix $A$ is given in J by $(-/ . *)A$. This expression uses the monadic conjunction and is defined recursively in terms of the minors of $A$, as explained under ‘Det’ in the J Introduction and Dictionary[15]. The Dictionary also describes the permanent as given by $(+/ . *)A$.

Other monadic expressions involving , might possibly be useful. For example, the trace $\sum a_{ii}$ of a matrix $A = (a_{ij})$ is produced by $(/ . +)A$ (this expression is, like the present author, cute and quite clever but remarkably inefficient). Similarly the odds ratio $a_{00} a_{11} / a_{01} a_{10}$ of a $(2 \times 2)$ matrix $A$ is given by $(%/ . *)A$

\[
A = 1+i.2 2 \text{ ] B = } . ? . 3 \text{ 3 } \text{ $20$}
\]

\[
\text{det} = -/ . * \text{ ] pm } . +/ . * \text{ ] tr } . / . + \text{ ] or } . \% / . *
\]

\[
xrs 'A' ; 'det A' ; 'pm A' ; 'tr A' ; 'or A' ; 'B' ; 'det B' ; 'pm B' ; 'tr B' ; '<.' . + B'
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 4 & 5 & 6 \\
10 & 10 & 10 & 10 & 13 & 13 \\
0.6666667 & 15 & 4 & 0 & 1854 & 24 \\
3 & 3 & 2 & 2 & 15 & 15 \\
\end{array}
\]

Comments:

1. The monadic verb , used above to create the matrix B, is called roll (fixed seed) and returns a reproducible sequence of pseudorandom numbers as detailed in Section 4.2.

2. The conjunction [, (lev) yields its left argument and is used above as a statement separator when defining several verbs on the same line. Note that using [, (left) instead would have incorporated ‘[ pm = . +/ . *’ in the definition of det, since [ is just a verb like any other:

46
Similarly, \( \) \( \) (\textit{dex}) is a conjunction that yields its right argument.

3. The expression \(<\/. * \text{B}\), or equivalently \((<\ /. +)\text{B}\), picks one atom from each row and each column of \( \text{B} \) such that the sum of these atoms is a minimum, and returns this minimal sum (in the above example, \( 15 = 2*13+0 = b_{00} + b_{21} + b_{12} \)). Such expressions could be useful in operational research.

Note also that the possible sets of atoms, one from each row and each column of \( \text{B} \), are given by the columns of \( , / . * \). Therefore \( <\/. */ . / . \). \( \text{B} \) is 15 in the above example, and \( */ . / . \). \( \text{B} \) is 4482, the permanent of \( \text{B} \).

### 2.3.5.2 Matrix Multiplication \((*/ . *)\)

Matrix multiplication in J is performed by dyadic \( */ . *\), an example of inner product (page 50).

The following example shows that matrix multiplication works in the obvious way with vectors, and also illustrates the use of \( !\): (transpose):

\[
A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad \text{mp} = */ . *
\]

\[x = \begin{bmatrix} 1.3 \\ 1.4 \end{bmatrix}, \quad y = \begin{bmatrix} 1 \end{bmatrix}\]

\[
A \text{ mp} \mid:\! A \mid \begin{bmatrix} 1 \end{bmatrix}, \quad (\mid:\! A) \text{ mp} A \mid \begin{bmatrix} 1 \end{bmatrix}, \quad x \text{ mp} A \mid \begin{bmatrix} 1 \end{bmatrix}, \quad A \text{ mp } y \mid \begin{bmatrix} 1 \end{bmatrix}, \quad y \text{ mp } y
\]

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
A & A \text{ mp} \mid:\! A & (\mid:\! A) \text{ mp} A & x \text{ mp} A & x \text{ mp} A & x \text{ mp} A & A \text{ mp } y & y \text{ mp } y \\
\hline
0 & 1 & 2 & 3 & 14 & 38 & 62 & 0 & 1 & 2 & 3 & 20 & 23 & 26 & 29 & 14 & 38 & 62 & 14 \\
3 & 4 & 3 & 3 & 4 & 4 & 3 & 4 & 4 & 3 \\
\hline
\end{array}
\]

A simple way to raise a square matrix \( A \) to a power \( n \) is to insert matrix multiplication \((*/ . *)\) in an array of rank 3, whose \( n \) items are each the matrix \( A \). For example, the relationship

\[
\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}^n = \begin{pmatrix} F_{2n-1} & F_{2n} \\ F_{2n} & F_{2n+1} \end{pmatrix},
\]

where \( F_k \) is the \( k \)th Fibonacci number, can be demonstrated in J as follows:
2.3.5.3 Matrix Inverse & Divide (%) 

Monadic % (matrix inverse) returns the inverse of a non-singular matrix:

$$B = . 4 4 \% 13$$

$$xrs 'B'; 'B'; 'B'$$

<table>
<thead>
<tr>
<th>% . B</th>
<th>(mp .) B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 9 5 6</td>
<td>0.0334728 0.06485356 0.07531381 0.1338912</td>
</tr>
<tr>
<td>2 0 8 10</td>
<td>0.1129707 0.03138075 0.0041841 0.04811715</td>
</tr>
<tr>
<td>0 0 6 8</td>
<td>0.0334728 0.4351464 0.07531381 0.3661088</td>
</tr>
<tr>
<td>4 4</td>
<td>0.3263598 0.05648536 0.3995816</td>
</tr>
</tbody>
</table>

Here and in the rest of this Section, mp=. +/ . * represents matrix product.

In general matrix inversion produces rounding errors ((mp .)B should be the identity matrix), but like all J's arithmetic operations, % can be carried out to arbitrary precision:

$$xrs 'B'; 'B'; 'B'; 'B'; 'B'; 'B'$$

<table>
<thead>
<tr>
<th>% . x: B</th>
<th>(mp .) x: B</th>
<th>% . x: B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 9 5 6</td>
<td>1 0 0 0</td>
<td>1 9 5 6</td>
</tr>
<tr>
<td>2 0 8 10</td>
<td>0 1 0 0</td>
<td>2 0 8 10</td>
</tr>
<tr>
<td>0 0 6 8</td>
<td>0 0 1 0</td>
<td>0 0 6 8</td>
</tr>
<tr>
<td>4 4</td>
<td>4 4</td>
<td>4 4</td>
</tr>
</tbody>
</table>

Dyadic % means matrix divide: if B is nonsingular then A%. B is B^(-1)A in standard notation. More generally, if y. is a matrix with r rows and c linearly independent columns (implying in particular that r ≥ c), and x. is a vector of length r, then x.%. y. is the least squares fit to x. in the space spanned by the columns of y..

For example, the least squares fit of the form ŷ_i = ᾱ_0 + ᾱ_1 x_i to the data

<table>
<thead>
<tr>
<th>i</th>
<th>1 2 3 4 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_i</td>
<td>1 3 4 5 7</td>
</tr>
<tr>
<td>y_i</td>
<td>8 8 4 4 1</td>
</tr>
</tbody>
</table>

48
is given by the following standard calculations (see for example DeGroot[7], section 10.5):

\[
y = \begin{pmatrix} 8 \\ 8 \\ 4 \\ 4 \\ 1 \end{pmatrix}, \quad X = \begin{pmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \\ 1 & 7 \end{pmatrix}, \quad \hat{\beta} = (X^T X)^{-1} X^T y = \begin{pmatrix} 10 \\ -1.25 \end{pmatrix}, \quad \hat{y} = X \hat{\beta} = \begin{pmatrix} 8.75 \\ 6.25 \\ 3.75 \end{pmatrix}.
\]

In J, given y and X as above, the fitted parameter vector \( \hat{\beta} = (\hat{\beta}_0, \hat{\beta}_1) \) is produced simply by y %. X, and other calculations related to least squares are also easy:

\[
y = 8 8 4 4 1\\x = 1 3 4 5 7\\xs = 1, x\\xrs 'y'; 'X'; 'b = y %. X'; 'Xt = . | : X'; '(% Xt mp X) mp Xt mp y'; 'X mp b'
\]

| y | X | b = y %. X | Xt = . | : X | (% Xt mp X) mp Xt mp y | X mp b |
|---|---|---|---|---|---|
| 8 8 4 4 1 | 1 1 13 1 4 1 5 1 7 | 10 -1.25 | 1 1 1 1 1 3 4 5 7 | 10 -1.25 | 8.75 6.25 5 3.75 1.25 |
| 5 | 5 2 2 | 2 5 | 2 | 5 |

Further examples of least squares fitting and the associated matrix manipulations are given in Section 4.2.

As detailed in the J Introduction and Dictionary[15], both monadic and dyadic % are more generally applicable than is implied above:

1. x %. y is defined for higher rank arguments. For example, if x is a matrix, then x %. y returns the least squares fit to each column of x.

2. If y is a non-square matrix with r rows and c linearly independent columns, then % y means I % y, where I is the \((r \times r)\) identity matrix.

This implies that % y is the pseudo-inverse of y: the pseudo-inverse of A is the unique matrix X satisfying the Moore-Penrose conditions:

(a) \(AXA = A\),
(b) \(XAX = X\),
(c) \((AX)^T = AX\) (i.e. AX is symmetric),
(d) \((XA)^T =XA\) (i.e. XA is symmetric),

see e.g. Golub & Van Loan[8], section 5.5.4.

\[
\text{xrs 'A = . 1, 1.4'; 'x = . X A'; 'A mp x mp A'; 'X mp A mp X'; 'A mp A'; 'X mp A'}
\]

<table>
<thead>
<tr>
<th>A = . 1, 1.4</th>
<th>X = . X A</th>
<th>A mp x mp A</th>
<th>X mp A mp X</th>
<th>A mp X</th>
<th>X mp A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>0.7 0.4 0.1 -0.2</td>
<td>1 0</td>
<td>0.7 0.4 0.1 -0.2</td>
<td>0.7 0.4 0.1 -0.2</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1</td>
<td>-0.3 -0.1 0.1 0.3</td>
<td>1 1</td>
<td>-0.3 -0.1 0.1 0.3</td>
<td>0.4 0.3 0.2 -0.1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 2</td>
<td>-1 2</td>
<td>1 2</td>
<td>0.1 2 0.3 0.4</td>
<td>0.1 2 0.3 0.4</td>
<td>0.1 2 0.3 0.4</td>
</tr>
<tr>
<td>1 3</td>
<td>-1 3</td>
<td>1 3</td>
<td>-0.2 0.1 0.4 0.7</td>
<td>-0.2 0.1 0.4 0.7</td>
<td>0.1 2 0.3 0.4</td>
</tr>
<tr>
<td>4 2</td>
<td>2 4</td>
<td>4 2</td>
<td>2 4</td>
<td>4 4</td>
<td>2 2</td>
</tr>
</tbody>
</table>
3. If \( y \) is a nonzero vector, then \( \% y \) inverts it with respect to the unit circle; see page 42.

2.3.5.4 Inner Product

In its clearest form, when \( x \) and \( y \) are matrices, \( x \).+/ . * \( y \) has entries obtained by taking each row of \( x \) and each column of \( y \), multiplying corresponding elements, and summing over the resulting vector.

More generally, \( f \) . \( g \) is an inner product: if its arguments \( x \) and \( y \) are matrices and \( g \) has rank zero, then \( x \). \( f \). \( g \) \( y \), takes each row of \( x \) and each column of \( y \), applies dyadic \( g \) between corresponding elements, and then monadic \( f \) to the resulting vector.

In practice, the verb \( f \) in the inner product \( f \).\( g \) is usually derived by insertion, for example

\[ x \cdot (+/ . =) \cdot y \] counts the matches between each row of \( x \) & column of \( y \).

\[ x \cdot (*/ . =) \cdot y \] returns 1 for each row of \( x \) and column of \( y \) that are identical, and

\[ x \cdot (+/ . ~:) \cdot y \] returns 1 for each row of \( x \) and column of \( y \) that differ.

However, the effects of inner products involving logical and relational functions are often more simply produced by other J verbs such as \(-:\) (match) or \( e\) . (member):

\[
\begin{array}{cccccccc}
 x & y & x +/ . = & y & x * . / . = & y & x +/ . ~: & y & x -/"1/ y" \cdot y \cdot e \cdot x \\
\hline
\text{cat} & \text{ant} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\text{bat} & \text{man} & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{man} & \text{bee} & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{pig} & \text{emu} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{5} & \text{3} & \text{3} & \text{5} & \text{3} & \text{5} & \text{3} & \text{5} & \text{3} & 3 & 5 & 3 & 3 \\
\end{array}
\]

Inner products can also be applied to vectors, as illustrated on page 47 for matrix product and in the following example:

\[
\text{xrs '2 3 37 */ .^ 1 3 1'; 'factors=. __ q: 666 1998 999999'; '(*/ .^)"2 factors'}
\]

\[
\begin{array}{cccccccc}
\text{2 3 37 */ .^ 1 3 1} & \text{factors=. __ q: 666 1998 999999} & \text{(*/ .^)"2 factors} \\
\hline
\text{1998} & 2 & 3 & 37 & 0 & 0 & 666 & 1998 & 999999 \\
 & 1 & 2 & 1 & 0 & 0 & & & \\
 & 2 & 3 & 37 & 0 & 0 & & & \\
 & 1 & 3 & 1 & 0 & 0 & & & \\
 & 3 & 7 & 11 & 13 & 37 & & & \\
 & 3 & 1 & 1 & 1 & 1 & & & \\
3 & 2 & 5 & 3 & 3 & 5 & 3 & \\
\end{array}
\]

The following illustrates how various inner products may be useful in summarising a table of data, for example to show which students have passed all their exams, what was the lowest percentage mark for each student, and which was each student’s best subject (0, 1, 2 or 3):

50
max=. 100 50 60 100 NB. maximum marks possible in each exam
pass=. 40 25 30 35 NB. pass mark for each exam
M=. 60 35 40 50 "'1(5 1 2 l)'" 1 -/ ? . 2 7 4$8 15 10 40 NB. simulate marks
xrs 'M';'max,:pass';'M */. >: pass';'100 * M <./ . % max';'M (i.>.)/ . % max'

<table>
<thead>
<tr>
<th>M</th>
<th>max,:pass</th>
<th>M */. &gt;: pass</th>
<th>100 * M &lt;./ . % max</th>
<th>M (i.&gt;.)/ . % max</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 33 34 61</td>
<td>100 50 60 100</td>
<td>100 48 52 25</td>
<td>40 64 47</td>
<td>1 2 3 2 1 2 2</td>
</tr>
<tr>
<td>65</td>
<td>24 46 52</td>
<td>40 25 30 35</td>
<td>1 0 1 0 0 1 1</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>26 44 74</td>
<td>25 25 36 50</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>31 24 56</td>
<td>70 32 50 74</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>31 50 47</td>
<td>70</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7 4</td>
<td>2 4</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

The final example of inner product illustrates the more general case \( x \cdot f \cdot g \cdot y \) where neither \( x \) nor \( y \) is a matrix, \( f \) isn't derived by insertion, and \( g \) has non-zero rank:

\[
x = (2+i, 3, 1, 2) \div 0 0 0 0 0 0 \\
y = #: 1 2 3 4 6; 7 8 17 31 \\
xrs 'x'; 'y'; '#. b. 0'; 'x < . #. y'; 'x +/ . #. y'; '2 3#."0 1 y'
\]

\[
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>#. b. 0</th>
<th>x &lt; . #. y</th>
<th>x +/ . #. y</th>
<th>2 3#.&quot;0 1 y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 2</td>
<td>2</td>
<td>2 2 2 2 2</td>
<td>0 0 0 0 0 1</td>
<td>1 1 1</td>
<td>14 29 85 125</td>
</tr>
<tr>
<td>3 3</td>
<td>3 3 3 3 3</td>
<td>0 0 0 0 0 1 0</td>
<td>0 0 0 1 1</td>
<td>1 2 3 4</td>
<td>32 129 631 797</td>
</tr>
<tr>
<td>4 4</td>
<td>4</td>
<td>4 4 4 4 4</td>
<td>0 0 0 0 1 1</td>
<td>1 4 5 16</td>
<td>58 349 2409 2837</td>
</tr>
<tr>
<td>5 5</td>
<td>5 5 5 5 5</td>
<td>0 0 0 0 0 0 0</td>
<td>1 0 0 0 0</td>
<td>31 125 626 781</td>
<td></td>
</tr>
<tr>
<td>6 6</td>
<td>6 6 6 6 6</td>
<td>1 1 1 1 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 7</td>
<td>7 7 7 7 7</td>
<td>1 1 1 1 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here #: has dyadic rank 1 1, so

1. For \( x f . #. y \) to work, the 1-cells of \( x \) must agree with the 1-cells of \( y \) (both in fact have shape 5).
2. The last axis of the frame of 1-cells in \( x \) must agree with the first axis of the frame of 1-cells in \( y \) (the frames here have shapes 3 1 2 and 2 4 respectively).
3. The inner product \( x f . #. y \) calculates \( f(z #. y) \) for each 2-cell \( z \) in \( x \), and returns the results in a frame of shape 3 1.

More generally, if \( g \) in \( x . f . g . y \) has dyadic rank \( k_1 k_2 \), then

1. The \( k_1 \)-cells of \( x \) must agree with the \( k_2 \)-cells of \( y \).
2. The last axis of the frame of \( k_1 \)-cells in \( x \) must agree with the first axis of the frame of \( k_2 \)-cells in \( y \).
3. The inner product \( x . f . g . y \) calculates \( f(z g y.) \) for each \((k_1+1)\)-cell \( z \) in \( x \), and returns the results in a frame of the same shape as that of the \((k_1+1)\)-cells in \( x \).
In the simplest case such as matrix product, \( g \) has dyadic rank 0 0, so

1. The 0-cells of \( x \) automatically agree with the 0-cells of \( y \), both being scalars.
2. The last axis of \( x \) must agree with the first axis of \( y \).
3. The inner product \( x \, f \, g \, y \) calculates \( f(z \, g \, y) \) for each 1-cell \( z \) in \( x \), and returns the results (each of shape \( y \)) in a frame of shape \( y \).

### 2.3.5.5 Miscellaneous Matrix Functions

One simple way to produce the \( n \times n \) identity matrix is \((=@i.) n\), using the verb \textit{self-classify}. Many other highly-structured matrices, such as triangular matrices, Hilbert matrices and Toeplitz matrices\[8\], are easily produced via the \textit{table} adverb (page 58):

\[
\begin{align*}
\text{ident} &= = \@ i. \\
\text{lowertri} &= >:/ @ i. \\
\text{hilbert} &= \% @ >: @ (+/\sim) @ i. \\
\text{toeplitz} &= (\langle - i. -/ i.)@-,@-@#) \{ \}
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{ident 5} & \text{lowertri 5} & \text{hilbert 5x} & \text{toeplitz 31 41 59 26 53 58 97 93 23} \\
\hline
1 0 0 0 0 & 1 0 0 0 0 & 1 1 2 1 3 & 1 5 3 5 8 97 93 23 \\
0 1 0 0 0 & 1 1 0 0 0 & 1 2 3 4 5 & 2 6 53 58 97 93 \\
0 0 1 0 0 & 1 1 1 0 0 & 1 3 4 5 6 & 59 26 53 58 97 \\
0 0 0 1 0 & 1 1 1 1 0 & 1 4 5 6 7 & 41 59 26 53 58 \\
0 0 0 0 1 & 1 1 1 1 1 & 1 5 6 7 8 & 31 41 59 26 53 \\
\hline
5 5 & 5 5 & 5 5 & 5 5
\end{array}
\]

Several other examples are given in J Phrases\[2\], chapter 5.

More advanced matrix computations, such as eigenstructure analysis and SVD, LU, \& QR decompositions, can be produced using foreign conjunction (Section 3.2.1.16) or J’s distributed script files (Section 3.3.1).

### 2.3.6 More Conjunctions

#### 2.3.6.1 Bonding, Function Composition etc. (\&: \& and \&:\)

Between a dyadic verb and a noun, the conjunction \& (\textit{bond}) glues the noun to the verb as one of its arguments, giving a new (monadic) verb:

\[
\begin{align*}
\text{log10} &= 10^\sim \\
\text{cube} &= ^\&3 \\
\text{cuberoot} &= 3^\%/ \\
\text{twopower} &= 2^\& \\
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{log10 2 3 4 100} & \text{cube 2 3 5} & \text{cuberoot 10 125} & \text{twopower i. 6} \\
\hline
0.30103 0.4771213 0.60206 2 & 8 27 125 & 2.15443 5 & 1 2 4 8 16 32 \\
4 & 3 & 2 & 6 \\
\hline
\end{array}
\]

52
Between two monadic verbs, & denotes composition: f&g y. means f(g(y)), i.e. 'apply g to y., then f to the result'. If f is dyadic, then f&g means 'apply g to each argument, then f between the results'.

The following dialogue illustrates many uses of & while checking for anagrams and palindromes:

```
isaanag=: -: & (/:~) NB. (dyadic) is x. an anagram of y.?
'integral calculus' isaanag 'calculating rules'
1 'eleven + two' isaanag 'twelve + one'
1 'punishment' isaanag 'nine thumps'
0 noblanks=: -.&' NB. (monadic) remove blanks from y.
isaanag1=: isaanag & noblanks
'punishment' isaanag1 'nine thumps'
1 'schoolmaster' isaanag1 'the classroom'
1 'the aristocracy' isaanag1 'a rich tory caste'
1 'Anagrams' isaanag1 'Ars magna'
1 'Horatio Nelson' isaanag1 'Honor est a Nilo'
1 'incomprehensible' isaanag1 'problem in Chinese'
0 a. i. 'aAZ'
97 65 90
isupper=: >:&65 *. <:&90
lfu=: + (32&e & isupper) NB. lower-case index from upper-case index
lowercase=: lfu &. (a.&i.) NB. convert upper-case to lower
isaanag2=: isaanag1 & lowercase
'incomprehensible' isaanag2 'problem in Chinese'
1 'Ivanhoe by Sir Walter Scott' isaanag2 'A novel by a Scottish writer'
1 'Nessiteras Rhombopteryx' isaanag2 'Monster hoax by Sir Peter S.'
0 islower=: >:&97 *. <:&122
lowercase=: ((islower #.) & lfu) &. (a.&i.) NB. redefined
'Nessiteras Rhombopteryx' isaanag2 'Monster hoax by Sir Peter S.'
1 'King’’s Lead Hat' isaanag2 'Talking Heads'
1 ispalin=: (:-: |.) & lowercase NB. (monadic) is y. a palindrome?
ispalin 'A man, a plan, a canal - Panama!'
1 ispalin 'Dennis and Edna sinned'
1 ispalin 'Marge lets Norah see Sharon’’s telegram'
```

Comments:

1. /:- y. can be read as ‘sorted items of y.’ (in the case of character data y., sorted into alphabetical order according to a.).

2. The first anagram-checking function (isaanag) fails if the number of spaces in the left and right arguments don’t agree. The second attempt (isaanag1) corrects this, but takes no account of upper- and lower-case letters. The third attempt (isaanag2) then fails if there is any non-matching punctuation, but finally
works correctly after one of the verbs it refers to (lowercase) has been rewritten to omit everything but letters.

3. The above J dialogue was written in a way that demonstrates &, but there are many other (and perhaps clearer) ways of defining the verbs used. For example, (32^n * isupper) could be used instead of (32^n & isupper).

4. ‘Ars magna’ means ‘great art’, ‘Nesiteras Rhombopteryx’ was the name given by Sir Peter Scott to the Loch Ness Monster when some alleged underwater photos of her were published many years ago, and yes, this example was included partly for a bit of light relief.

Another J conjunction between successively applied verbs is @ (atop). In the monadic case, @ has the same effect as &, i.e. f@g y. and f&g y. both mean f(g(y)). However, f@g differs from f&g when used dyadically.

x. f&g y. means (g x.) f (g y.), i.e. apply g monadically, then f dyadically.
Thus 2 -% 5 means \( \frac{1}{2} - \frac{1}{5} \).

x. f@g y. means f(x. g y.), i.e. apply g dyadically, then f monadically.
Thus 2 -% 5 means \( -\frac{5}{5} \).

Examples of dyadic & and @ follow:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a , b</th>
<th>a , b</th>
<th>a , b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>0.4</td>
<td>1.5</td>
<td>0.1</td>
<td>0.123012301</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The conjunctions @: (at) and &: (appose) resemble @ and & respectively, except that the ranks of the resulting function are infinite, whereas the ranks of f@g and f&g are those of g. In my experience, if you’re unsure which of @, &; or &: you need, then you probably need @:, as illustrated below:

+/@*: i. 5 NB. * has rank 0, so +/@*: also has rank 0 & sums over each atom
0 1 4 9 16
+/@*: i. 5 NB. this is probably what was intended: the sum of squares
30

23.6.2 Power (^:)

Often one wishes to apply a verb repeatedly. The power conjunction (^:) does this: the expression f ^: n applies the verb f, n times:

\[
\begin{array}{|c|c|}
\hline
\text{(^: 2) 0 1 2 3} & \text{NB. ((0^2)-2), ((1^2)-2), ((2^2)-2), ((3^2)-2)} \\
0 1 16 81 & \\
\hline
\text{(^: 3) 0 1 2 3 2} & \text{NB. 2, (2^2)-2, ((2^2)-2), ((2^2)-2)^2) } \\
2 4 16 256 & \\
\hline
\text{(^: 2) 2x 3x} & \text{NB. ((2^2)-(2^2)), ((3^2)-(3^2))} \\
256 44342648824303776994852496306194892603 & \\
\hline
\end{array}
\]

54
J recognises when iterated applications of a verb converge: the expression \( f^\infty \cdot y \), literally means ‘apply \( f \) infinitely often with starting value \( y \)’, but the process will be stopped as soon as successive results are equal, since the result would be unchanging thereafter. Another special case of the power conjunction is \( f^\infty \cdot 1 \) (the inverse of \( f \), if it’s defined). Both these special cases appear in the following example of Newton’s method, employing the iteration \( y \mapsto \frac{1}{2}(y + 3/y) \) to find \( \sqrt{3} \):

\[
\text{nxt} = \text{nxt} \cdot:\text{0}(+3\%)
\text{NB. next approximation to square root of 3}
\text{xxs} \quad \text{nxt} \cdot 1'; \text{nxt} 1.5'; \text{nxt} \cdot 2(1)'; \text{nxt} \cdot 3(1)'; \text{x} = \text{nxt} \cdot _{-} (1)'; \text{x} \cdot 2'; 'x' \cdot ^{\infty} \cdot _{-} (1(3))
\]

<table>
<thead>
<tr>
<th>nxt 1</th>
<th>nxt 1.5</th>
<th>nxt \cdot 2(1)</th>
<th>nxt \cdot 3(1)</th>
<th>x = nxt \cdot _{-} (1)</th>
<th>x \cdot 2</th>
<th>*' = ^{\infty} \cdot _{-} (1(3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.75</td>
<td>1.75</td>
<td>1.73214</td>
<td>1.73205</td>
<td>3</td>
<td>1.73205</td>
</tr>
</tbody>
</table>

This is easily, if unnecessarily, generalised to a function to find the square root of any number. The following example finds \( \sqrt{2} \) and \( \sqrt{3} \) to 14 significant figures, and shows that \( 9^2 + 19^2/22 \) is approximately \( \pi^4 \):

\[
\text{pp} = \text{(9!}:10) 0 \quad \text{NB. save current print precision}
(9!):11 14 \quad \text{NB. increase print precision}
\text{nxt} = \text{nxt} \cdot:\text{0}(\% + \[) \quad \text{NB. next approximation from \( y \) to sqrt(x.)}
\text{sqrt} = \text{J nxt} = _{-} 1: \quad \text{NB. create verb ‘sqrt’ equivalent to \%:}
\text{xxs} \quad '3' \text{nxt} 1'; '3(nxt} = _{-} (1)'; 'sqrt 2'; 'sqrt} = 2 (\%9)+(\%19)%22'; '1p1'
\]

<table>
<thead>
<tr>
<th>3</th>
<th>nxt 1</th>
<th>3(nxt = _{-} (1)</th>
<th>sqrt 2</th>
<th>sqrt \cdot 2 (%9)+(%19)%22</th>
<th>1p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.7320508075689</td>
<td>1.4142135623731</td>
<td>3.1415926525826</td>
<td>3.1415926535898</td>
<td></td>
</tr>
</tbody>
</table>

9!\cdot11 pp \quad \text{NB. reset print precision}

Finally, the expression \( f^{\infty} \cdot n \) where \( n \) is a negative integer means ‘apply the inverse of \( f \), \( n \) times’:

\[
\text{xxs} \quad '3' \cdot 1_2 -3' \cdot 3'; '3^2 4 8'; '1 x = . ^{\infty} \cdot 3 '' \text{Russian Doll''}'; ^{\infty} = _{-} x'
\]

<table>
<thead>
<tr>
<th>(3 \cdot 1_2 -3) 3</th>
<th>3^2 4 8</th>
<th>J x = . ^{\infty} \cdot 3 'Russian Doll'</th>
<th>^{\infty} = _{-} x</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 81 6561</td>
<td>9 81 6561</td>
<td>Russian Doll</td>
<td>Russian Doll</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>
2.3.6.3 Under (\&.)

The expression f&.g (‘f under g’) is like f@g, except that g¹_ is applied to each cell of the result. Note that transformations of the form g⁻¹f@g are common throughout mathematics.

J knows the inverses of many primitive operations such as ^ (log), so &.^ means ‘under log’, i.e. ‘on a log scale’. Another example is prime: p: y. returns the y.th prime(s). This is a partial inverse to p:^_1, which J defines to mean ‘index of smallest prime greater than or equal to y’, or equivalently ‘number of primes less than y’:

| xrs '*:13'; '*:I 169'; '+:&._. 13'; 'p:i.10'; 'p:I 1000000'; '<:&.(p:I) 1000000 |
|---|---|---|---|---|
| 169 | 13 | 169 | 2 3 5 7 11 13 17 19 23 29 | 78498 999983 |

The above dialogue shows that +:&._. (‘double on a log scale’) is equivalent to *: (square), and that the largest prime below 1000000 is 999983.

A common use of &. is f&.@, i.e. open each box, apply f, and close the box again. The standard J startup script defines the helpful mnemonic each=. &.@ as used below (and as used also in the definition of xrs, Section 2.4).

y=. (i.5); (i.2 5); ('first';'second'); 'lamina'; 'Able was I ere I saw Elba'

| 0 1 2 3 4 | 0 1 2 3 4 | first | lamina | Able was I ere I saw Elba |
| 5 6 7 8 9 | 5 6 7 8 9 | second |

| 4 3 2 1 0 | 5 6 7 8 9 | second | animal | ableE was I ere I saw elBA |
| 0 1 2 3 4 | 0 1 2 3 4 | first |

| Able was I ere I saw Elba | lamina | first | 0 1 2 3 4 | 5 6 7 8 9 |
| | second | 0 1 2 3 4 |

J can deduce the inverses of many other bijective operations, implicitly using rules like (fg)⁻¹ = g⁻¹f⁻¹. Sometimes, however, you may need to specify the inverse of an operation explicitly, using &: (obverse). For example, if you have defined a verb N01F that returns (an accurate approximation to) the standard Normal cumulative distribution function, and a verb N01q that approximates the inverse c.d.f., then you could define the probit transformation by probit=: N01q &. N01F, and carry out statistical analysis on a probit scale using &.probit. A simpler illustration follows:

iflchar=: 'MDCLXVI'&i. { 1000 500 100 50 10 5 1"_ NB. int from Latin char
isheadmax=: */ @ ((. >: )@.)) NB. take account of IV, IX, XL etc.
iflaf=: _1: + ++@isheadmax \ NB. multiplying factor for each char
ifi=: +/ @ (* iflaf) @ iflchar NB. Integer from Latin numerals
if1 'MCMLIX'
1959
if1=: 2 5 2 5 2 5&#: NB. partial inverse (IIII not IV etc.)
if1 1959
MDCCCLVIII
if1 # 'Josephus Agrestis Piscesque'
XXVII
IFL=: if1 ;. if1
inLatin=: &. IFL
'III' *inLatin 'DCLIII'
MDCCCLVIII
3 * 653
1959

2.3.6.4 Fit (!.)

Many J verbs can have their action modified by the fit conjunction (!.).

For example, dyadic $ (shape), (append) and {. (take) all use padding if needed. The customised verb f!.a applies f, padding out with the atom a when necessary:

```
   i23=: i. 1 2 3 [ i5=: i. 1 5 [ a23=: 2 3$'abcdef' [ a5=: 5$pqrs' 
xrs 'i23,i5';'i23 ,!._1 i5';'a23,a5';'a23 ,!._1 i5';'3 6$!._1 i5'
```

```
   +-------+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>pq</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
</tr>
</tbody>
</table>
+-------+-------+-------+-------+-------+

   +-------+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>a23</th>
<th>a23</th>
<th>a23</th>
<th>a23</th>
<th>a23</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>abc</td>
<td>abc</td>
<td>abc</td>
<td>abc</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>def</td>
<td>def</td>
<td>def</td>
<td>def</td>
<td>def</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>pqrs</td>
<td>pqrs</td>
<td>pqrs</td>
<td>pqrs</td>
<td>pqrs</td>
</tr>
</tbody>
</table>
+-------+-------+-------+-------+-------+

   +-------+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>
+-------+-------+-------+-------+-------+

Another use of the fit conjunction is to specify zero tolerance. For example, J treats x=y. as true if x. and y. are sufficiently close ('tolerably equal'), whereas x. =!.0 y. makes the comparison exact (up to available computer accuracy). Several other verbs like dyadic | (residue) can be similarly customised:

```
   y=. 1 + 10 - _10 - i.10
   y = 1 0 0 0 0 1 1 1 1 1
   y =!._0.1 0 0 0 0 0 0 0 0 0 0
   y = exactly 1 0 0 0 0 0 0 0 0 0 0
   y = exactly 1 0 0 0 0 0 0 0 0 0 0
   y = exactly 1 0 0 0 0 0 0 0 0 0 0
   y = exactly 1 0 0 0 0 0 0 0 0 0 0
   y = exactly 1 0 0 0 0 0 0 0 0 0 0
```

The J Introduction and Dictionary[15] seems to suggest other tolerances can be defined (using e.g. =!._t), but my current attempts to do so just produce domain errors.

57
2.3.7 Miscellaneous Constructions

2.3.7.1 Table (Dyadic u. /) and Outer Products

The following tables show all 16 possible dyadic Boolean functions:

```
btt: (; ; 1"_" , ; (2 1$0 1)"_" ; do@, /"- 0 1"_" ) NB. Boolean table
do each ('bt''","1 ,&''",")each cut '0:"0 *, ["0 < ]"0 - : +'
```

```
<table>
<thead>
<tr>
<th></th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
do each ('bt''","1 ,&''",")each cut '+: = -.@]"0 >: -.@"0 <: *: 1:"0'
```

```
<table>
<thead>
<tr>
<th></th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

See also the adverb b. (Boolean) in the J Introduction and Dictionary[15].

The following table shows sin(θ), cos(θ) & tan(θ) for θ = 0°, 30°, 45°, 60°, 90°, 135°, 180°, 270°, 360°:

```
theta=. 0 30 45 60 90 135 180 270 360
rfd=. *1r180p1 NB. radians from degrees
(4 3$'degsincostan') ; 1 2 3 (] , (o. rfd)^"0/) theta
```

```
<table>
<thead>
<tr>
<th>deg</th>
<th>0 30 45 60 90 135 180 270 360</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>0 0.5 0.7071068 0.8660254 1 0.7071068 0 1 -1 2.44921e_16</td>
</tr>
<tr>
<td>cos</td>
<td>1 0.8660254 0.7071068 0.5 6.12303e_17 -0.7071068 -1 0 1</td>
</tr>
<tr>
<td>tan</td>
<td>0 0.5773503 1 1.73205 1.63318e16 _1 0 _1 2.44921e_16</td>
</tr>
</tbody>
</table>
```

Many familiar mathematical objects, for example Pascal's triangle, identity matrices & Hilbert matrices, can be simply expressed using the table adverb reflexively:

```
xrr '!/- i.5 ; =/- i.6 ; '%'@@:(+/-) i.4 ; </- i.5'
```

```
<table>
<thead>
<tr>
<th>!/- i.5</th>
<th>=/- i.6</th>
<th>'%'@@:(+/-) i.4</th>
<th>&lt;/- i.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
<td>1 0 0 0 0</td>
<td>1 0.5 0.3333333 0.25 1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>0 1 2 3 4</td>
<td>0 1 0 0 0</td>
<td>0.5 0.3333333 0.25 0.2 0 1 1 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 0 1 3 6</td>
<td>0 0 1 0 0</td>
<td>0.3333333 0.25 0.2 0.1666667 0 0 1 1</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 0 0 1 4</td>
<td>0 0 0 1 0</td>
<td>0.25 0.2 0.1666667 0.1428571 0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>0 0 0 0 1</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>5 5</td>
<td>6 6</td>
<td>4 4</td>
<td>5 5</td>
</tr>
</tbody>
</table>
```

58
mt=. (* \@p) \@p: NB. multiplication table for prime-order Finite Field
xrs 'p: 2 3 4' ; 'mt 2' ; 'mt 3' ; 'mt 4'

<table>
<thead>
<tr>
<th>p:</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>mt 2</th>
<th>mt 3</th>
<th>mt 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 7 11</td>
<td>1 2 3 4</td>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 6 7 8 9 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 4 13</td>
<td>2 4 6 1 3 5</td>
<td>2 4 6 8 10 1 3 5 7 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 1 4 2</td>
<td>3 6 2 5 1 4</td>
<td>3 6 9 1 4 7 10 2 5 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 3 2 1</td>
<td>4 1 5 2 6 3</td>
<td>4 8 1 5 9 2 6 10 3 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.7.2 Catalogue (monadic {) and Cartesian Products

2.3.7.3 Gerunds

A gerund is a verb acting as a noun; the idea is implemented in J using the primitives '{ (tie), ': (evoke gerund), / (insert) and }. (agenda). The following simple example implements the factorial function as a verb f by defining a gerund (f0‘fn); f0 is invoked if y=0, and the recursive function fn is invoked if y>0:

```
f0=: 1:
fn=: * f0 <::
fa=: *
   NB. factorial 0
   NB. factorial y.=n>0 defined recursively
   NB. 0 if y=0, 1 if y>0
f=: f0'fn@.fa"0
   NB. factorial y. using gerund and agenda
f i. 6
1 1 2 6 24 120
```

Comments:

1. The verb f fails (with limit error) if its argument y. isn’t a non-negative integer.
2. I have hardly ever used gerunds, preferring instead the more down-to-earth programming methods of Section 2.4.
3. Details and further examples of gerunds are given in the J Introduction and Dictionary[15].
2.4 J Programming

". (do)
xrs
[ and each

2.4.1 Assignment

2.4.1.1 Local (=.) and Global (=:) Assignment

=:. is (local)
=: is (global)
(debugging)

2.4.1.2 Indirect Assignment

2.4.2 Tacit Definition

each=: &..>
inv=: ^:_1
limit=: ^:_
exactly=: !.0
Section 4.1.2 gives several further examples of tacit definitions.

2.4.3 Explicit Definition

monadic : (explicit)
dyadic : (monad/dyad)
one-line explicit
polynomials, rational functions
matrix to power
The script reproduced in Section 4.3 gives several examples of explicit definitions.

2.4.4 Recursion and Self-Reference ($:)

factorials
Fibonacci
Viete
tower of Hanoi
adjacency matrix, connectedness, ergodicity
2.4.5 Examples

Suppose that you’re researching the lengths of intervals between successive primes, and you want a J verb that returns the distance between the two primes ‘bracketing’ y. The following dialogue shows several tacit definitions, culminating with the verb `lenpint`:

```j
npl=: p: inv       NB. number of primes less than y.
sub10=: -&1 0      NB. subtract 1, 0 from y.
pnp=: sub10 &. npl  NB. previous, next prime (i.e. primes bracketing y.)
from=: --          NB. x. from y. (i.e. y. minus x.)
lenpint=: from/ @ pnp
lenpint 1000
12
pnp 1000
997 1009
```

This shows that the length of the interval is 12, and that in fact the two primes bracketing 1000 are 997 and 1009.

The number of intermediate definitions and comments in the above snippet is (to me) excessive. At the other extreme, the required verb could be defined in one go:

```j
1pi=: --/ @ (-&1 0 &. p: ^: -: 1)
1pi 1000
12
```

This is hard to read, but the computer helps if you ask what it understands by `1pi`:

```j
1pi
```

Note that by default J responds with the boxed representation of any expression that doesn’t evaluate to a noun. This helps interpretation:

1. `1pi` is shown to be of the form `V1 @ V2`.
2. Looking into the box representing `V2` shows that `V2` has the form `'V21` under the transformation `V22` (where `V21` and `V22` are further verbs).
3. `V21` means ‘subtract 1 0’ and `V22` means ‘smallest prime ≥’. Therefore, with a composite argument, `V2` means ‘previous, next prime’ (you can confirm these with a little experimentation).
4. Finally, `V1` means ‘insert --’. In particular, if `V1`’s argument `y` is a list of two numbers, then take the first away from the second.

You could similarly check the definition of `lenpint`. However, to resolve all intermediate definitions, you need the adverb `f.` (fix):

61
You could equally have created an explicit definition of the required verb:

\[
\text{lpi}=: 3: 0
\text{inp}=: (p:^:-1) y \text{ NB. index of next prime}
\text{pnp}=: p: inp - 1 0
\text{--/ pnp}
\)
\text{lpi} 1000
12
\]

None of these definitions is fully satisfactory, however. Stylistically, \text{lempint} is overexuberant in making tacit definitions of everything, \text{lpi} is obscure for inexpert J users, and \text{lpi} hides some of the mathematical structure (e.g. the use of \&.). More importantly, they only work on scalar arguments:

\[
\text{lpi} 1000 2000 3000
\]

A simple way to generalise to lists or higher-rank arrays is to apply your verb with rank 0, so that it acts separately on each atom of \text{y}:

\[
(\text{lpi}^"0) 1000 2000 3000
12 4 2
\]

The zero-rank verbs \text{lempint}^"0 and \text{lpi}^"0 would work similarly.

Alternatively, a little experimentation (or a little insight!) shows that the problems start when trying to subtract 1 0 from the array of indices of the next primes. You should instead subtract 1 0 from each index separately, i.e. use rank 0:

\[
\text{sub10 nplt 1000 2000 3000}
\text{length error: sub10}
\text{sub10 nplt 1000 2000 3000}
\text{sub10^"0 nplt 1000 2000 3000}
167 168
302 303
429 430
\]

Similarly, you need to change the rank of \text{--/}. My preferred definition of \text{lpi}, which uses another possibly useful verb (\text{pnp}) and incorporates terse but helpful comments, would be:
Further examples of tacit definitions are shown in the dialogue below, which uses extended precision integers and rationals to produce an accurate approximation to $e$:

```j
rfactab=: 1: ,. !@x:@i. NB. 1st column 1, 2nd column e.p. factorials
xfrfr2=: (2&x:^:._1)^:1 NB. e.p. rational from numerator, denom.
erat=: */ @ (xfrfr2 @ rfactab) NB. rational approx to $e$ using $y$. terms
e20=. erat 20
(; x:^:._1) e20 NB. rational, real approx. to $e$
```

$$8266416490601r304112751022082.71828$$

```j
e50=. erat 50
ft=: ":: @ (.< @ (* 10x&")) NB. format extended integer x. * 10^y.
dp=: -[ @ (tel ; (.) ft= ) ] NB. represent y. to x. decimal places
!50
3.04141e64
60 dp e50 NB. previous line shows that e50 is correct to > 60 d.p.
```

$$2 718281828459045235360287471352662497757247093699959574966967$$

### 2.4.6 Control Structure

```j
if. do. end.
else.
elseif.
while. do. end.
whilst. do. end.
for. do. end.
for_name. do. end.
break.
continue.
select. case. end.
fcase.
try. catch. end.
goto_name.
label_name.
return.
```
Chapter 3

J Implementation

J is currently implemented on Windows 95/NT, Windows 3.1, DOS 386, Linux, Mac, RS/6000 and Sparc. The Windows and Mac versions also come with an application development environment.

This Chapter primarily describes the Windows 3.1 Professional edition. A freeware version for Windows 3.1 is downloadable from http://www.jsoftware.com/. This freeware version lacks DDE and is slower (having no coprocessor support, and missing certain code optimizations), but otherwise is identical to the Professional edition.

3.1 J for Windows

J is supplied with the ISIJ font, which includes the box-drawing characters.

Profile + configuration (J Latest)

3.1.0.1 J for Windows 95/NT

The Windows 95/NT version is similar to the Windows 3.1 version, but has added support for Windows 95/NT-specific features:

- OLE (Object Linking and Embedding) and OCX (Microsoft Custom Control) allow two processes to intercommunicate. J comes with an example linking J to Excel for Windows 95; similarly J can communicate with software such as Delphi, Visual Basic and Visual C++.
- Java is well supported. In particular, ‘J Automation objects’ can be used by Java products like MS J++ and Net browsers.
- Windows 95/NT specific controls. For example, these are used in the Regular Expression Demo, which is therefore unavailable under Windows 3.1.
- 32-bit DLL rather than 16-bit.
- The OpenGL graphics system, allowing 3-D graphics[25].
- Runtime (jr) and locked (j1) scripts larger than 64k can be created (important for people building and distributing large applications).
3.2 Foreign Conjunction

The conjunction (!: (foreign) ...
3.2.1.9 PC DOS Facilities (8 :)

3.2.1.10 Global Parameters (9 :)

Random link

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlq=</td>
<td>9!:0</td>
</tr>
<tr>
<td>rls=</td>
<td>9!:1</td>
</tr>
<tr>
<td>rlg</td>
<td>0</td>
</tr>
<tr>
<td>16807</td>
<td>? 5</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>282475249</td>
<td>(7^5) ; (2^31) ; (2^31)</td>
</tr>
</tbody>
</table>

Print precision

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppq=</td>
<td>9!:10</td>
</tr>
<tr>
<td>pps=</td>
<td>9!:11</td>
</tr>
<tr>
<td>ppq</td>
<td>0</td>
</tr>
<tr>
<td>dummy=</td>
<td>pps 14</td>
</tr>
<tr>
<td></td>
<td>(%: 1r3) ; - %: 328r1835</td>
</tr>
<tr>
<td>dummy=</td>
<td>pps 6</td>
</tr>
</tbody>
</table>

3.2.1.11 Windows (11 !:)

3.2.1.12 Debug (13 !:)

3.2.1.13 Data Driver (14 !:)

3.2.1.14 Dynamic Link Library (15 !:)

3.2.1.15 Socket Driver (16 !:)

Only available under Windows 95/NT.
3.2.1.16 Numerical Functions (128 !:)

Currently this family is relatively empty. It includes the QR decomposition of a complex matrix (128!:0) and inversion of a square upper-triangular matrix (128!:1).

3.2.2 List of Foreign Conjunctions

The file main/xenos.js, reproduced below, gives recommended names to most of the verbs formed by foreign conjunction, and provides a useful reference to them.

```
NB. xenos.js - suggested names for external conjunctions
NB. This file is not required on start up.
NB. Names given here are referenced in J system libraries stdlib.js  
NB. and winlib.js only if they are also defined there.
NB. Debug/Data Driver/DLL/socket names are also in files:
NB. debug.js/dd.js/dll.js/socket.js
NB. For file access, the definitions in files.js are recommended.
NB. Names are not given for conjunctions specific to Mac or PC DOS, or
NB. for conjunctions 128!:x

script= 0!:0  NB. script
scriptn= 0!:0  NB. noisy script
scriptx= 0!:100  NB. execute noun as script
scriptnx= 0!:100  NB. execute noun as noisy script
fdir= 1!:0  NB. file directory
fread= 1!:1  NB. file read
fwrite= 1!:2  NB. file write
fappend= 1!:3  NB. file append
fsize= 1!:4  NB. file size
fcd= 1!:5  NB. create directory
fatr= 1!:6  NB. query/set attributes
fperm= 1!:7  NB. query/set permissions
freadx= 1!:11  NB. indexed file read
fwritex= 1!:12  NB. indexed file write
fmsx= 1!:20  NB. open file numbers/names
fopen= 1!:21  NB. file open
fclose= 1!:22  NB. file close
flock= 1!:30  NB. file locks
flockx= 1!:31  NB. lock file
funlock= 1!:32  NB. unlock file
ferase= 1!:55  NB. erase file
host= 2!:0  NB. host
hosts= 2!:1  NB. host spawn
off= 2!:55  NB. exit from J

ntype= 3!:0  NB. noun storage type
ntrep= 3!:1  NB. noun internal representation
ntrepx= 3!:2  NB. convert from internal representation
ic= 3!:4  NB. convert integer/character
fc= 3!:5  NB. convert float/character

nameclass= 4!:0  NB. name class
namelist= 4!:1  NB. name list
scriptlist= 4!:3  NB. loaded script list
scriptindex= 4!:4  NB. index into script list
nameset= 4!:5  NB. names set 1=start counting, 0=stop
erase= 4!:55  NB. erase
```
fixrep= 5!0 NB. fix from 5!1 representation
atom1= 5!1 NB. atomic representation
display= 5!2 NB. display representation
tree= 5!4 NB. tree representation
linear= 5!5 NB. linear representation
paren= 5!6 NB. parenthesized representation
time= 6!0 NB. time	
times= 6!1 NB. time since session start
times= 6!2 NB. time expression
timed= 6!3 NB. time delay
space= 7!0 NB. space in use
spaces= 7!1 NB. space used since session start
space= 7!2 NB. space used to execute expression
memq= 7!3 NB. memory manager query
memrel= 7!4 NB. memory manager release
rlq= 9!0 NB. query random link
rls= 9!1 NB. set random link
prompts= 9!4 NB. query input prompt
boxdraw= 9!6 NB. query box drawing characters
boxdraw= 9!7 NB. set box drawing characters
errormsg= 9!8 NB. query error messages
errormsg= 9!9 NB. set error messages
pp= 9!10 NB. query print precision
pp= 9!11 NB. set print precision
sysq= 9!12 NB. query system
verq= 9!14 NB. query version
psq= 9!15 NB. query positioning and spacing
pes= 9!16 NB. set positioning and spacing
to= 9!18 NB. query comparison tolerance
to= 9!20 NB. set comparison tolerance
wd= 11!0 NB. windows driver
dbr= 13!0 NB. reset, set suspension mode
dbs= 13!1 NB. display stack
dbsq= 13!2 NB. stop query
dbs= 13!3 NB. stop set
brun= 13!4 NB. run again
brun= 13!5 NB. run next
br= 13!6 NB. exit and return argument
brjmp= 13!7 NB. jump to line number
bsig= 13!8 NB. signal error
dberr= 13!11 NB. last error number
dber= 13!12 NB. last error message
dnetk= 13!13 NB. call stack
dbxml= 13!14 NB. latent expression query
dbxml= 13!15 NB. latent expression set
dbtrace= 13!16 NB. trace control
ddcon= 14!0 NB. connect
dddis= 14!1 NB. disconnect
ddsq= 14!2 NB. SQL
ddfet= 14!3 NB. fetch
ddcol= 14!4 NB. columns
ddcol= 14!5 NB. columns selected
ddsrc= 14!6 NB. data source names
ddsel= 14!7 NB. selection
ddend= 14!8 NB. end sql statement
dderr= 14!9 NB. error info
ddtrans= 14!10 NB. begin transaction
ddcons= 14!11 NB. commit transaction
ddrbk= 14!12 NB. rollback transaction
ddtbl= 14!13 NB. tables
ddfetch= 14!14 NB. fetch in columns
ddcnt= 14!15 NB. rowcount of last ddsq
3.3 Associated Scripts and Packages

3.3.1 Mathematics

3.3.2 Regular Expressions

main\regex.js
Contains the main definitions for forming and using regular expressions.

packages\regex\regbuild.js
Contains further verbs for building regular expressions.

Regular expression searching is also available from the J menu (Edit | Find in Files).


3.4 Lab Sessions

The J Windows menu has an entry Studio, which provides on-line ‘labs’ (interactive tutorials) on using J.

You can create your own labs with the Studio | Author menu command, and package them with any software you distribute.

List of tutorials
3.5 Package Development

3.5.1 Locales

- avoid name conflicts
- protect users from unimportant details of an application
- separate out different parts of a large application

3.5.2 Debugging

3.5.3 Distribution

A J application can be freely distributed, with a free run-time version of J. Script files can be encoded.

3.6 Graphics Interface

wd from release notes.

2d and 3d graphics (J Latest) mouse and character events (J Latest)
Plot (J Latest)

3.7 J Support

3.7.1 Help Files

3.7.2 J Books and Papers

3.7.3 J on the Net

news:comp.lang.apl (Usenet newsgroup for J and APL)
http://www.jssoftware.com (Iverson Software Inc.)
http://www.watserv1.uwaterloo.ca/languages/j/ (Waterloo J archives)
http://www.acm.org/sigapl/ (ACM Special Interest Group on APL)
http://www.torontoapl.org/ (Toronto Special Interest Group on APL)
http://www.vector.org.uk/ (Vector: quarterly publication of the British APL Association)
http://www.cs.trinity.edu/About/The_Courses/cs301/ (Trinity College course CS301)
http://www.warwick.ac.uk/Statsdept/Staff/JEHS/ (My official homepage)
http://web.cs.ualberta.ca/pubs/~smillie/ (Ken Smillie's homepage)
http://www.apl.demon.co.uk

70
Chapter 4

J Examples

4.1 Utilities

4.1.1 Standard J Utilities

4.1.2 My Utilities

The following extracts from my own myutil.js file show J verbs and other definitions that I have found generally useful. Note the importance of including examples of each utility within myutil.js: documented code induces feelings of security, spiritual well-being and insufferable smugness.

4.1.2.1 cart

```j
rcart=: *## $ ]
lcart=: #* #
cart=: 11 : 'lcart x. rcart'
```

NB.
NB. ### Example 1 ###
NB. (i.2 3) -cart i.3 3
NB. 1 1 4
NB. 0 1 32
NB. 0 1 256
NB. 1 4 25
NB. 27 256 3125
NB. 729 16384 390625
NB.
NB. ### Example 2 ###
NB. (i.3 5) ,.cart 10+i.2 2
NB. 0 1 2 10 11
NB. 0 1 2 12 13
NB. 3 4 5 10 11
NB. 3 4 5 12 13
NB. 6 7 8 10 11
NB. 6 7 8 12 13
NB.
NB. ### Alternatives ###
NB. cartbox=: , @ { @ (, & (0<1)))
NB. (i.3 3) >&((@&.) @ cartbox) 10+i.2 2
NB. 0 1 2 10 11
NB. 0 1 2 12 13
```

71
4.1.2.2 copyshape

\[
\text{(i. 15) copyshape i. 3 4}
\]

\[
\text{0 1 2 3 4 5 6 7 8 9 10 11 0 1 2}
\]

\[
\text{(i. 15) copyshape- i. 3 4}
\]

\[
\text{0 1 2 3}
\]

\[
\text{4 5 6 7}
\]

\[
\text{8 9 10 11}
\]

4.1.2.3 expand

\[
\text{expand= } /:/0;0[ \{ #@[ \{. \}]}
\]

\[
\text{(from Ken Iverson)}
\]

\[
\text{### Example ###}
\]

\[
\text{1 0 0 1 0 1 expand 2 3 4}
\]

\[
\text{2 0 0 3 0 4}
\]

4.1.2.4 expprod

\[
\text{expprod= (\#-\#) ((\$@+(\$/0)@,0 @ [0,0+/0,0[@] ,0]}
\]

\[
\text{### Example + Comparison ###}
\]

\[
\text{expprod2=} \text{shape2 } @ (\text{expand}'1')}
\]

\[
\text{x;y; (x expprod y); x expprod2 y}
\]

\[
\begin{array}{cccc}
\text{1 1 0 0 5 6} & \text{5 6 0 0 5 6 0 0} \\
\text{1 0 1 0 7 8} & \text{7 8 0 0 7 8 0 0} \\
\text{1 0 0 1} & \text{5 0 6 0 5 0 6 0} \\
\text{0 1 1 0} & \text{7 0 8 0 7 0 8 0} \\
\text{0 1 0 1} & \text{5 0 0 6 5 0 0 6} \\
\text{0 0 1 1} & \text{7 0 0 8 7 0 8 0} \\
\text{0 5 6 0} & \text{0 5 6 0 5 6 0} \\
\text{0 7 8 0} & \text{0 7 8 0 7 8 0} \\
\text{0 0 5 6} & \text{0 0 5 6 0 5 6} \\
\text{0 0 7 8} & \text{0 0 7 8 0 7 8}
\end{array}
\]

\[
\text{100 timex 'x expprod y'}
\]

\[
\text{0.0038}
\]

\[
\text{100 timex 'x expprod2 y'}
\]

\[
\text{0.0176}
\]

4.1.2.5 kronprod

\[
\text{kronprod=} \text{#@$ } ,0; ("0 1 cart) f}
\]
NB.
NB. ### Example ###
NB. a
NB. 1 2
NB. 3 _1
NB. b
NB. 0 1 2
NB. 3 4 5
NB. 6 7 8
NB. a kronprod b
NB. 0 1 2 0 2 4
NB. 3 4 5 6 8 10
NB. 6 7 8 12 14 16
NB. 0 3 6 0 _1 _2
NB. 9 12 15 _3 _4 _5
NB. 18 21 24 _6 _7 _8
NB. b kronprod a
NB. 0 0 1 2 2 4
NB. 0 0 3 _1 6 _2
NB. 3 6 4 8 5 10
NB. 9 _3 12 _4 15 _5
NB. 6 12 7 14 8 16
NB. 18 _6 21 _7 24 _8

4.1.2.6 rowprod

rowprod=: ,"2/ @ (*"1/)

NB.
NB. ### Example ###
NB. (i.3 4); 10+i.2 4

<table>
<thead>
<tr>
<th>0 1 2 3</th>
<th>10 11 12 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 6 7</td>
<td>14 15 16 17</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td></td>
</tr>
</tbody>
</table>

NB. (i.3 4) rowprod 10+i.2 4
NB. 0 11 24 39
NB. 0 15 32 51
NB. 40 55 72 91
NB. 56 75 96 119
NB. 80 99 120 143
NB. 112 135 160 187

4.1.2.7 shape2

shape2=: ((*/@) : ,@) @ $ @ ,: $ ,

NB.
NB. ### Examples ###
NB. ($;]) shape2 4
NB. 1 1 4
NB. ($;]) shape2 i. 4
NB. 1 4 0 1 2 3
NB. ($;]) shape2 i. 3 4
NB. 3 4 0 1 2 3
NB. 4 5 6 7
NB. 8 9 10 11
NB.  ($;]) shape2 i. 2 3 4
NB.
\begin{verbatim}
NB. 6 4  0 1 2 3
NB. 4 5  6 7
NB. 8 9 10 11
NB. 12 13 14 15
NB. 16 17 18 19
NB. 20 21 22 23

Note that there are many other ways to write \texttt{shape2}, but they all seem slower, e.g.:

\begin{verbatim}
(shape2 :: ,.&.(|:"_10[:])) i. 2 3 4 5 6 7
100 timex 'shape2' i. 2 3 4 5 6 7'
0.0115
100 timex ',.&.(|:"_10[:]) i. 2 3 4 5 6 7'
0.0434
\end{verbatim}
\end{verbatim}

4.1.2.8 diffs

diffs=: }. - }:

\begin{verbatim}
NB. ### Example ###
NB. diffs 1 3 6 10
NB. xy ; diffs xy
NB. 0 1  2
NB. 1 2  1
NB. 3 3  0
NB. 6 2  1
NB. 10 1
NB. diff2=: 2: -/\]
NB. diff2 1 3 6 10
NB. 2 3 4
NB. 100 timex 'diffs 1 3 6 10'
NB. 0.0017
NB. 100 timex 'diff2 1 3 6 10'
NB. 0.0038
\end{verbatim}

4.1.2.9 iotav

iotav=: [: ; i. each

\begin{verbatim}
NB. (from Roger Hui)
NB. ### Example ###
NB. iotav 3 1 4 1 5
NB. 0 1 2 0 1 2 3 0 1 2 3 4
NB. ### JEHS unidiomatic alternative - was faster in J2! ###
NB. time x '[: ; i.&.>] i.1000' NB. J2 timing
NB. 3.46
NB. time x '[: ; <@i."0" i.1000' NB. J2 timing
NB. 3.08
NB. 10 time x '[: ; i.&.>] i.1000' NB. J3
NB. 0.653
NB. 10 time x '[: ; <@i."0" i.1000' NB. J3
NB. 1.912
\end{verbatim}
4.1.2.10 nubfreq

nubfreq=: diffs 0 ($ , - (~/:~/) # i.@$)

NB.
NB. ### Example ###
NB. (/:-@~ ; nubfreq) 3 1 4 1 5 9 2 6 5 3 5
NB. 1 2 3 4 5 6 9 2 1 2 1 3 1 1
NB.

4.1.2.11 nubrank

nubrank=: i.- (/:-@~.)

NB.
NB. ### Example ###
NB. (; nubrank) %-/ 1 2 3 4
NB. 1 0.5 0.3333333 0.25 5 2 1 0
NB. 2 1 0.6666667 0.5 8 5 3 2
NB. 3 1.5 1 0.75 9 7 5 4
NB. 4 2 1.33333 1 10 8 6 5
NB.

4.1.2.12 countint

countint=: <: 0 (nubfreq @ (/:- @ ((i.@>:@>:@.))/,])])

NB.
NB. ### Example ###
NB. countint 3 1 4 1 5 9 2 6 5 3 5
NB. 0 2 1 2 1 3 1 0 0 1

4.1.2.13 ordercols

ordercols=: ([ , i.@#] -. [ ] { ])"1

NB.
NB. ### Example ###
NB. 1 2 ordercols i.7 5
NB. 1 2 0 3 4
NB. 6 7 5 8 9
NB. 11 12 10 13 14
NB. 16 17 15 18 19
NB. 21 22 20 23 24
NB. 26 27 25 28 29
NB. 31 32 30 33 34

4.1.2.14 round

round=: <. 0 (0.5&+)

NB.
NB. ### Example ###
NB. round ((~: >: %@: 5)~: 1.20) %: 5
NB. 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

4.1.2.15 v2

v2=: (v2false * -.0*v2test) + v2true * v2test
NB.  ### Example 1 ###
NB.  \[ v2test =: > \]
NB.  \[ v2true =: ] \]
NB.  \[ v2false =: [ \]
NB.  \[ min =: v2 f. \]
NB.  \[ 3 1 4 1 5 9 \]
NB.  \[ 2 7 1 8 2 8 \]
NB.  \[ 2 1 1 1 2 8 \]
NB.  
NB.  ### Example 2 ### (double or halve; cf agenda in dictionary)
NB.  \[ v2test =: >x9 \]
NB.  \[ v2true =: -. \]
NB.  \[ v2false =: +: \]
NB.  \[ dorh1 =: v2 f. \]
NB.  \[ Quicker than the following \]
NB.  \[ dorh =: +:/"-\-@. (?)>8: \]
NB.  \[ x =: 55000@20 \]
NB.  \[ timex 'dorh"0 x' \]
NB.  \[ 8.85 \]
NB.  \[ timex 'dorh1 x' \]
NB.  \[ 0.38 \]
NB.  \[ (dorh"0 -: dorh1) x \]
NB.  1
4.2 Statistics

[1] [26]
cdf for a contingency table using :. (cut)
tagulation using / . (key)
monadic ? (roll)
dyadic ? (deal)
monadic ?. (roll (fixed seed))
dyadic ?. (deal (fixed seed))
J Phrases[2], chapter 10.
[20] [21] [22] [23] [24]

```j
gamma =: ! & <:
beta =: *@gamma % gamma@+
beta2=: % @ ((* % !) * ([ ! +))
0.7 beta 0.3
3.88322
0.7 beta2 0.3
3.88322
(! _0.3) * (! _0.7) % ! 0
3.88322
1p1 % 1 o. 0.3p1
3.88322
```

NB. beta function B(x,.y.) from gamma function
NB. one way to get beta function from dyadic !
NB. explicit calculation of B(0.7,0.3)
NB. B(x,1-x) = pi / sin(pi x) for 0<x<1
4.3 Timetabling

This example illustrates

1. Creation and manipulation of a database in J,
2. Incorporating J output in a \LaTeX{} document.

4.3.1 J Timetabling Script

The following J script (timab.js) sets up a database of timetabling information, including course codes, names, lecturers involved and student numbers, timetabling slots, locations and rooms, and course requirements for each Statistics degree.

```
NB. J.E.H. Shaw Timetabling in J
NB. ---------------------------------
NB. Created 18-Sept-1997
NB. Last modified 11-Dec-1997
NB.
NB. Type 'alltt 0' to produce \LaTeX{}2e timetabling file '\temp\alltt.tex'
NB.
NB. require 'convert dates files format misc strings'
NB. ---------------------------------------------------------------
NB. Staff teaching duties
NB.

STAFFCOURSES=: do @:> chop 0 : 0
  'HPW' ; < chop 'ST305'
  'JBC' ; < chop 'ST323 ST329 ST332'
  'JEMS' ; < chop 'seminar ST104 ST104p ST217/b ST217a/b ST329 ST952'
  'JQS' ; < chop 'ST114 ST301'
  'JW' ; < chop 'ST318'
  'PAE' ; < chop 'ST217/a ST217a/a ST304 ST329'
  'RJR' ; < chop 'ST113 ST215 ST327'
  'SBJ' ; < chop 'Stock ST108 ST111/a ST111/b ST112/a ST112/b ST208 ST213'
  'WSK' ; < chop 'ST202 ST333'
)
STAFF=: '{."1 STAFFCOURSES
isstaff=: < e. STAFF&[

NB. Course code ; name; abbreviation ; size
NB. Course name will only be printed up to '/' (if present)
NB.

COURSES=: do @:> chop 0 : 0
  'Stoch' ; 'Stochastic Calculus + Control' ; 'Reading Course' ; 15
  'Stsem' ; 'Statistics Departmental Seminar' ; 'STsem' ; 20
  'ST104' ; 'Statistical Laboratory I' ; 'Stat Lab I' ; 210
  'ST104p' ; 'Stat Lab practical' ; 'Stat Lab prac' ; 210
  'ST105' ; 'Applications of Algebra and Analysis' ; 'AAA' ; 80
  'ST111' ; 'Probability (Part A)' ; 'Prob A' ; 380
  'ST111/a' ; 'Probability (Part A; Maths + M/S)' ; 'Prob A' ; 280
  'ST111/b' ; 'Probability (Part A; M/ISE)' ; 'Prob A' ; 70
  'ST112' ; 'Probability (Part B)' ; 'Prob B' ; 300
  'ST112/a' ; 'Probability (Part B; Maths + M/S)' ; 'Prob A' ; 230
  'ST112/b' ; 'Probability (Part B; M/ISE)' ; 'Prob A' ; 70
  'ST113' ; 'Statistical Computing' ; 'Stat Comp' ; 75
```

78
<... Approximately 100 lines omitted ...>

'MA3F4'; 'Linear Analysis'; 'Lin An'; 140  
'MA300'; 'Modern Control Theory'; 'Cont Th'; 90  
'PH201'; 'History of Modern Philosophy'; 'Mod Phil'; 60  
)

NB. ==============================================================
NB. Main allowed courses for each (course,year);
NB. "core", 'major', 'minor', 'tiny' in order of importance
NB.
MORSE1=: do 0.:> chop 0 : 0  
'CS117'; 'minor'  
'CS123'; 'tiny'  
'CS126'; 'tiny'  
'CS128'; 'minor'  
'EC106'; 'core'  
'GE111'; 'tiny'  
'IB104'; 'core'  
'MA105'; 'core'  
'MA112'; 'tiny'  
'MA113'; 'tiny'  
'MA125'; 'minor'  
'MA128'; 'tiny'  
'MA129'; 'core'  
'MA130'; 'minor'  
'MA131'; 'core'  
'MA246'; 'minor'  
'ST104'; 'major'  
'ST105'; 'minor'  
'ST106'; 'core'  
'ST111/b'; 'core'  
'ST112/b'; 'core'  
'ST113'; 'core'  
'ST114'; 'major'  
)

MORSE2=: do 0.:> chop 0 : 0  
'CS201'; 'tiny'

<... Approximately 200 lines omitted ...>

MSC= do 0.:> chop 0 : 0  
'Stoch'; 'core'  
'Stein'; 'core'  
'ST301'; 'major'  
'ST304'; 'core'  
'ST305'; 'core'  
'ST313'; 'major'  
'ST318'; 'major'  
'ST323'; 'core'  
'ST327'; 'major'  
'ST329'; 'major'  
'ST332'; 'core'  
'ST333'; 'major'  
'ST952'; 'core'  
)

NB. ==============================================================
NB. Lecture ; term ; day ; time ; room ; weeks
NB.
LECTURES= do 0.:> chop 0 : 0  
'CS117'; 1 ; 'mon' ; 13 ; 'L3' ; 'w:2-10'  
'CS117'; 1 ; 'thur' ; 15 ; 'R0.21' ; 'w:1-10'  
'CS128'; 3 ; 'mon' ; 15 ; 'CS-local' ; 'w:1-5'  
'CS128'; 3 ; 'tues' ; 9 ; 'CS-local' ; 'w:1-5'  
'CS128'; 3 ; 'thur' ; 17 ; 'CS-local' ; 'w:1-5'
'EC106' ; 1 ; 'mon' ; 14 ; 'L4' ; 'w:1-10'
'EC106' ; 1 ; 'tues' ; 16 ; 'F107' ; 'w:1-10'
'EC106' ; 2 ; 'thur' ; 16 ; 'SO21' ; 'w:1-10'
'EC106' ; 2 ; 'fri' ; 16 ; 'SO21' ; 'w:1-10'

<... Approximately 450 lines omitted ...>

'ST333' ; 1 ; 'mon' ; 12 ; 'RM68' ; 'w:2-10'
'ST333' ; 1 ; 'wed' ; 11 ; 'RM68' ; 'w:1-10'
'ST333' ; 1 ; 'fri' ; 11 ; 'B212' ; 'w:1-10'

'ST952' ; 1 ; 'fri' ; 14 ; 'RM68' ; 'w:2-5'
'ST952' ; 1 ; 'fri' ; 15 ; 'RM68' ; 'w:2-5'

NB. ================================================================
NB. Days + times to appear in timetable (as coded in 'LECTURES')
NB.

DAYS: 'mon' ; 'tues' ; 'wed' ; 'thur' ; 'fri'
TIMES: 9 + i. 9

NB. ==============================================================
NB. Codelist 0 Returns codes of lectures whose times are known
NB. Lectlist 'MS21' Returns lecture list required for M&S year 2 term 1
NB. Maketable Make timetable (boxed entries) from lecture list
NB. Tt 'MS21' Returns J version of timetable for M&S year 2 term 1
NB.

codelist= : 3 : 0
codes= . /* ( & { } . each { . . 1 LECTURES
list= . ( , i . { . 1 COURSES e . "0 1 codes ) # COURSES
/ : ( . ( . 1 list ) , "1 , ' , ' , ' 1 ( > i { . "1 list )
)

lectlist= : 3 : 0
'start term' = . ( ; y . ; do { ; y .
if . isstaff start do.
. codes= , > (( < start = STAFF ) # { . . 1 STAFFCOURSES
else.
. do 'codes= . ( , . "1 , start , ' , )
end.
lect= . "1 LECTURES
i= . lect e . "0 1 codes
lists= . i # LECTURES
lts. > 1 ( . "1 list
(lt = term ) # list )

maketable= : 3 : 0
key= . TIMES i . > 3 ( . "1 y.
}. each key < / . y .

formatentry= : 3 : 0
'c y d t r w ' = . y .
( i . & / ( { . & / ) c ) , ( ' , ' ) , : w

maketable= : 3 : 0
dummy= . ( 1 ; 2 ) , "1 ( ; DAYS ; < "0 TIMES ) , "1 ( 5 ; 6
list= . dummy , y .
key= . DAYS i . 2 ( "1 list
table= . key < / . list
table= . > ( maketable ) each table

ttj= : 3 : 0
table= . maketable lectlist y.
table= ($ $ (formatentry"1 each @ ,))
table= DAYS,"0 table
t= "(' : TIMES) , '-' , (" : TIMES+1) , '"
table= (a, ("i t), table

NB. =========================================================================
NB. alltt Produce LaTeX2e file to print all timetables
NB. cctab Return LaTeX2e commands for Course Code table
NB. tt Return LaTeX2e commands for a single timetable
NB. tentry Process entry, shape 1 6 (course,term,day,time,room,weeks)
NB.

TEXSEP= LF, ('%,75#='), LF, LF
setboldlist= 3 : 0
l= ('core') = {"1 y.
1 # {"1 y.
}
tthead= 3 : 0
'start term' = . (}; ; {); y.
l= 'begin{center}', LF, '{Large\bf ', l3= ' Term ',term, '} \par \bigskip', LF
if (isstaff start) do.
  boldlist= i. 0
12= start
else.
do 'boldlist= setboldlist', start
select. start
case. 'MSC' do. 12= 'M.Sc.'
case. do.
  'course year' = . (}; ; {); y.
slect. course
  case. 'MS' do. 12= 'Maths & Stats'
  case. 'MORSE' do. 12= 'MORSE'
case. do. 12= course
end.
13= ' Year ', year, 13
end.
l= 'begin{tabular}|c|*{', (' : TIMES), '}p{1.55cm}|}', LF
l1= ', \tttimes', LF, '\\ \hline', LF
l2= ,12,13,14,15

TTEND= 'end{tabular}', LF, 'end{center}', LF, 'newpage', LF, TEXSEP

tentry= 3 : 0
course term day time room weeks' = y.
a= ' \ttentry', ("<course> e. boldlist) # 'bf ',
a= a, ("i.'/ "{} course), '{', room, '}', " weeks, '"
break'

ttslot= 3 : 0
'&', (O<#y.) # LF, (_? ). vfm tentry"1 y.), LF

)
trow= 3 : 0
t= "(\&>/ ttslot each y.)
t, (LF "{:t} # LF), '\\ \hline', LF
)
ttbody= 3 : 0
tday= 'Mon' ; 'Tues' ; 'Wed' ; 'Thurs' ; 'Fri'
text= .
for_d. i. 5 do.
text= . text, ' \ttstrut ', (d pick ttdays), LF, trow d[y.
end.


This script creates various data structures. For example the noun ‘STAFFCOURSES’ is shown in Figure 4.1.
<table>
<thead>
<tr>
<th>STAFFCOURSES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HPW</strong></td>
</tr>
<tr>
<td><strong>JBC</strong></td>
</tr>
<tr>
<td><strong>JEHS</strong></td>
</tr>
<tr>
<td><strong>JQS</strong></td>
</tr>
<tr>
<td><strong>JW</strong></td>
</tr>
<tr>
<td><strong>PJH</strong></td>
</tr>
<tr>
<td><strong>RJR</strong></td>
</tr>
<tr>
<td><strong>SDJ</strong></td>
</tr>
<tr>
<td><strong>WSK</strong></td>
</tr>
</tbody>
</table>

Figure 4.1: Noun STAFFCOURSES created by timetab.js
Timetables for each lecturer, course, room or degree can be produced, and then formatted using \TeX. In particular, `alltt 0' produces the file `\texttt{\textbackslash temp\textbackslash alltt.tex}' containing timetables for each member of staff and for each degree course.
4.3.2 \TeX\ Timetables Produced by J

An abbreviated listing of the output file `\temp\alltt.tex` follows; it can be processed by \LaTeXe\ in the usual way \cite{\LaTeX}, finally producing tables like that shown in Figure 4.2.

```
\begin{document}
\begin{center}
\begin{tabular}{|c|}
\hline
\ttentry{MA125}{GLT1}{w:6-10} \\
\hline
\end{tabular}
\end{center}
```

<... Approximately 1000 lines omitted ...>

```
\begin{center}
\begin{tabular}{|c|}
\hline
\ttentry{PH201}{S020}{w:1-10} \\
\ttentry{\bf ST202}{ACCR}{w:2-10} \\
\ttentry{IB215}{ACCR}{w:1-10} \break
\ttentry{MA241}{L3}{w:2-10} \\
\end{tabular}
```

85
\ttentry{bf ST208}\{F110\}{w:2-10} & \\
\ttentry{bf IB106}\{R0.21\}{w:1-6,8-10} & \\
\ttentry{bf MA231}\{L3\}{w:1-10} & \\
\ttentry{bf ST217}\{PLT\}{w:2-10} & \\
// \hline

\ttstrut Tues

\ttentry{bf MA241}\{H051\}{w:1-10} & \\
\ttentry{bf IB207}\{S013\}{w:1-5} \break
\ttentry{bf IB207}\{L4\}{w:6-10} & \\
\ttentry{bf MA242}\{GLT1\}{w:1-10} & \\
\ttentry{bf ST208}\{rm68\}{w:1-10} & \\
\ttentry{bf MA244}\{GLT1\}{w:1-10} & \\
\ttentry{bf IB207}\{S021\}{w:1-10} & \\
// \hline

\ttstrut Wed

\ttentry{bf ST202}\{ACCR\}{w:1-10} & \\
\ttentry{bf MA241}\{L3\}{w:1-10} & \\
\ttentry{bf ST217}\{H052\}{w:1-10} & \\
\ttentry{bf IB215}\{H051\}{w:1-10} \break
\ttentry{bf MA244}\{R0.21\}{w:1-10} & \\
\ttentry{bf EC213}\{S021\}{w:1-10} & \\
// \hline

\ttstrut Thurs

\ttentry{bf ST217}\{ACCR\}{w:1-10} & \\
\ttentry{bf EC213}\{LIB2\}{w:1-10} & \\
\ttentry{bf MA231}\{ACCR\}{w:1-10} & \\
\ttentry{bf MA242}\{GLT1\}{w:1-10} & \\
\ttentry{bf IB106}\{R0.21\}{w:1-6,8-10} \break
\ttentry{bf ST208}\{F110\}{w:1-10} & \\
\ttentry{bf EC213}\{L5\}{w:1-10} & \\
\ttentry{bf IB207}\{L4\}{w:1-10} & \\
// \hline

\ttstrut Fri

\ttentry{bf MA244}\{L3\}{w:1-10} & \\
\ttentry{bf ST202}\{PLT\}{w:1-10} & \\
\ttentry{bf MA242}\{L3\}{w:1-10} & \\
\ttentry{bf MA231}\{R0.21\}{w:1-10} \break
\ttentry{bf ST208}\{F110\}{w:1-10} & \\
// \hline
\end{tabular}
\begin{center}
\end{center}
\newpage

\begin{center}
{\Large\bf Maths \& Stats Year 2 Term 2} \par \bigskip
\begin{tabular}{\tt|*{9}{p{1.5cm}}}{$\times$} \\
\hline
\ttstrut Mon \\
\entry{MA205}{R0.21}{w:1-10} \break 
\entry{PH201}{S020}{w:1-10} \\
\end{tabular}
\end{center}

<... Approximately 1400 lines omitted ...>

\ttstrut Fri \\
\ttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
Maths & Stats Year 2 Term 1

<table>
<thead>
<tr>
<th></th>
<th>9-10</th>
<th>10-11</th>
<th>11-12</th>
<th>12-13</th>
<th>13-14</th>
<th>14-15</th>
<th>15-16</th>
<th>16-17</th>
<th>17-18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>PH201</td>
<td>ST202</td>
<td>IB215</td>
<td>ST208</td>
<td>IB106</td>
<td>MA231</td>
<td>ST217</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S020</td>
<td>ACCR</td>
<td>ACCR</td>
<td>F110</td>
<td>R0.21</td>
<td>L3</td>
<td>PLT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w:1-10</td>
<td>w:2-10</td>
<td>w:1-10</td>
<td>w:2-10</td>
<td>w:1-6,8-10</td>
<td>w:1-10</td>
<td>w:2-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tues</td>
<td>MA241</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H051</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w:1-10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td>ST202</td>
<td></td>
<td>IB215</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACCR</td>
<td>MA241</td>
<td>H052</td>
<td>H051</td>
<td></td>
<td></td>
<td>IB207</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td></td>
<td>w:1-10</td>
<td>w:1-10</td>
<td></td>
<td>w:1-10</td>
<td>S021</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w:1-10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>w:6-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thurs</td>
<td>ST217</td>
<td>EC213</td>
<td>MA231</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACCR</td>
<td>LIB2</td>
<td>ACCR</td>
<td></td>
<td></td>
<td></td>
<td>IB207</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>w:1-10</td>
<td></td>
<td></td>
<td></td>
<td>L4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>w:1-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri</td>
<td>MA244</td>
<td>ST202</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>PLT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>w:1-10</td>
<td>w:1-10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: Example Timetable produced by `timetab.js`
References


<table>
<thead>
<tr>
<th><strong>Index</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>each (&amp;.), 47</td>
</tr>
<tr>
<td>xrs, 18, 51</td>
</tr>
<tr>
<td>ace (a:), 23</td>
</tr>
<tr>
<td>adverbs, 7</td>
</tr>
<tr>
<td>agenda (@), 50</td>
</tr>
<tr>
<td>alphabet (a.), 5</td>
</tr>
<tr>
<td>amend ({}), 22</td>
</tr>
<tr>
<td>and (.*), 10, 12</td>
</tr>
<tr>
<td>angle (r.), 16</td>
</tr>
<tr>
<td>antibase 2 (#.), 11</td>
</tr>
<tr>
<td>antibase (#.), 12, 16</td>
</tr>
<tr>
<td>append (,), 8, 9, 21, 48</td>
</tr>
<tr>
<td>appose (&amp;.), 45</td>
</tr>
<tr>
<td>arrays, 17, 33</td>
</tr>
<tr>
<td>at (@), 45</td>
</tr>
<tr>
<td>atoms, 8, 17</td>
</tr>
<tr>
<td>atop (@), 45</td>
</tr>
<tr>
<td>axis, 17</td>
</tr>
<tr>
<td>base 2 (#.), 11, 34</td>
</tr>
<tr>
<td>base (#.), 12</td>
</tr>
<tr>
<td>basic characteristics (b.), 35</td>
</tr>
<tr>
<td>behead (.), 21</td>
</tr>
<tr>
<td>beta function, 67</td>
</tr>
<tr>
<td>bond (&amp;), 44</td>
</tr>
<tr>
<td>Boolean (b.), 49</td>
</tr>
<tr>
<td>box (&lt;), 5, 17, 34</td>
</tr>
<tr>
<td>boxed representation, 27, 52</td>
</tr>
<tr>
<td>cap ([), 27, 29</td>
</tr>
<tr>
<td>catalogue ([), 50</td>
</tr>
<tr>
<td>ceiling (&gt;.), 15</td>
</tr>
<tr>
<td>cells, 18, 34</td>
</tr>
<tr>
<td>circular functions (o.), 49</td>
</tr>
<tr>
<td>coercion, 10, 11</td>
</tr>
<tr>
<td>comment (NB.), 5</td>
</tr>
<tr>
<td>complex (j.), 16</td>
</tr>
<tr>
<td>composition (&amp;), 44</td>
</tr>
<tr>
<td>conjugate (+), 16</td>
</tr>
<tr>
<td>conjunction, 7</td>
</tr>
<tr>
<td>constant functions (9: to 9:), 26, 28, 29</td>
</tr>
<tr>
<td>continued fractions, 30</td>
</tr>
<tr>
<td>control structure</td>
</tr>
<tr>
<td>break., 54</td>
</tr>
<tr>
<td>continue., 54</td>
</tr>
<tr>
<td>else., 54</td>
</tr>
<tr>
<td>elseif., 54</td>
</tr>
<tr>
<td>fcase., 54</td>
</tr>
<tr>
<td>for. do. end., 54</td>
</tr>
<tr>
<td>for_name. do. end., 54</td>
</tr>
<tr>
<td>goto_name., 54</td>
</tr>
<tr>
<td>if. do. end., 54</td>
</tr>
<tr>
<td>label_name., 54</td>
</tr>
<tr>
<td>return., 54</td>
</tr>
<tr>
<td>select. case. end., 54</td>
</tr>
<tr>
<td>try. catch. end., 54</td>
</tr>
<tr>
<td>while. do. end., 54</td>
</tr>
<tr>
<td>whilst. do. end., 54</td>
</tr>
<tr>
<td>copy (#), 19</td>
</tr>
<tr>
<td>cosine (2 o.), 49</td>
</tr>
<tr>
<td>CRLF, 3</td>
</tr>
<tr>
<td>curtail (;), 21</td>
</tr>
<tr>
<td>cut, 19</td>
</tr>
<tr>
<td>cut (:), 67</td>
</tr>
<tr>
<td>deal (fixed seed) (?), 67</td>
</tr>
<tr>
<td>deal (?), 67</td>
</tr>
<tr>
<td>decrement (&lt;:), 11, 15</td>
</tr>
<tr>
<td>derivative (D.), 7</td>
</tr>
<tr>
<td>derived verbs, 30</td>
</tr>
<tr>
<td>determinant (-/ . *), 38</td>
</tr>
<tr>
<td>dex (1.), 38</td>
</tr>
<tr>
<td>divided by (%), 2, 5</td>
</tr>
<tr>
<td>do (*.), 51</td>
</tr>
<tr>
<td>dot product (.), 41</td>
</tr>
<tr>
<td>double (+), 15, 47</td>
</tr>
<tr>
<td>drop (.), 21</td>
</tr>
<tr>
<td>dyadic, 5, 35</td>
</tr>
<tr>
<td>equal to (=), 10</td>
</tr>
<tr>
<td>evoke gerund (‘’), 50</td>
</tr>
<tr>
<td>evoke (-), 7</td>
</tr>
<tr>
<td>explicit definition, 51</td>
</tr>
<tr>
<td>explicit (;), 51</td>
</tr>
<tr>
<td>exponential (-*), 2, 15</td>
</tr>
<tr>
<td>extend (x:), 13, 14</td>
</tr>
</tbody>
</table>
factorial (!), 11, 15
fetch (!::), 23
Fibonacci numbers, 29, 30, 39
fit (!.), 48
fix (f.), 52
floor (<.), 15
foreign (!.), 14, 55
forks, 27, 28
frames, 18
from (!), 9, 22, 33
gamma function, 15, 67
GCD (+.), 12
gerunds, 50
grade down (/), 27
grade up (/), 27
half (-:/), 15
head (!.), 21
hooks, 26, 27
identity matrix, 43
imaginary (j.), 16
increment (>:/), 11, 15
indeterminate (-.), 12
index of (i.), 9, 33
infinity (verb) (-), 26
infinity (-), 12
infix (\), 32
infixes, 32
inner product, 41
insert (/), 2, 7, 30, 50
integers (i.), 17, 34
is (global) (=/), 51
is (local) (=/), 51
itemize (/), 21
items, 18
key (/), 67
labs, 60
laminant (/.), 21
larger of (>:/), 16
larger or equal (>:=), 10
larger than (>), 10
LCM (*), 12, 16
leaf (L::), 24
left ([], 19, 25, 38
length/angle (*), 16
less or equal (<:=), 10
less than (<), 5, 10
less (-), 28
lesser of (<), 16
lev ([.), 38
level conjunction (L:), 23
level (L.), 24
link (·), 8, 17, 21, 24
lists (vectors), 9, 17
log (·), 2, 8, 16, 47
logical functions, 10–12, 31
magnitude (!), 11, 15
map (!::), 23
match (-:), 26, 41
matrix divide (\%), 40
matrix inverse (\%), 2, 34, 39
matrix multiplication (+/ *), 39
member (e.), 28, 41
minus infinity (-), 12
minus (-), 2
monad/dyad (:), 51
monadic, 5
names, 3
natural log (·), 15, 33, 34
negate (-), 11, 15
negative sign (-), 11
Newton’s method, 46
not and (*), 10
not equal (-=), 10
not or (+=), 10
not (-), 2, 10
noun, 3
nouns, 5
nub sieve (-:), 28
nub (-.), 26
obverse (·), 47
open (>), 23
or (+), 10, 12
order of evaluation, 5, 25
out of (!), 12
outfix (\), 32
outfixes, 32
paddings, 11, 23, 34, 48
parentheses, 25
passive (-), 7
permanent (+/ *), 38
pi times (o.), 15
plus (+), 2
polar (r.), 16
polynomials (p.), 2
polynomials, 12
power conjunction (·), 7, 14, 46
power (-), 2, 16
precision, 13, 14, 46
prefix (\), 31
prefixes, 31
prime factors (q::), 12
prime (p::), 12, 47
primitives, 8
pseudo-inverse, 41
rank, 7, 19, 34, 45, 53
rank agreement, 37
rank conjunction (*), 22
ravel items (.,.), 21
ravel (.), 21
raze (;), 21
real/imaginary (+.), 16
reciprocal (%), 5, 15
recursion, 51
reflexive (-), 7
relational functions, 10
residue (\), 12, 48
reverse (\.), 20
right (\), 12
roll (fixed seed) (?), 38, 67
roll (?), 67
root (\%), 2, 16
rotate (\.), 20
same (\), 12
scalars, 17, 19
scripts, 3
self-classify (=), 29, 43
self-reference (\$:), 51
shape of ($), 19, 34
shape ($), 19, 35, 48
signum (*), 11, 15
sine (1 o.), 49
sort down (\:\), 27
sort up (\:/), 27
spread (S::), 23
square root (\%), 2, 15, 33, 34
square (*), 15, 47
stdlib, 3
stitch (.,.), 21
suffix (\:\), 32
table (/), 43, 49
tables (matrices), 17
tacit definition, 51–53
tail (\::), 21
take (\::), 21, 48
tally (#), 19
tangent (3 o.), 49
Taylor series approximation (T.), 2
tie (\'), 50
times (*), 2
tolerance, 14, 48
trains, 26, 29
transpose (\::), 20, 39
type
  Boolean, 10
  boxed, 17
  character, 9
  complex, 16
  extended integer, 13
  floating point, 14
  integer, 11, 13
  logical, 10
  rational, 14
under (\&.), 47
verbs, 5
word formation (\::), 7